Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

.arianee

Full-Privacy Extension ZK Circuits

Veridise Inc.

July 17, 2024

► **Prepared For:**

Arianee
https://www.arianee.org/

► **Prepared By:**

Jon Stephens
Ian Neal
Mark Anthony

► **Contact Us:**

contact@veridise.com

► **Version History:**

July 17, 2024	V2
July 16, 2024	V1
July 10, 2024	Initial Draft

Contents

1 Executive Summary

From July 8, 2024 to July 10, 2024, Arianee engaged Veridise to review the security of their Full-Privacy Extension ZK Circuits. The review covered both the smart contracts and zero-knowledge circuits that allow issuers to privately interact with the Arianee protocol. Note, however, that this report will focus on the ZK Circuit review.

Veridise conducted the assessment over 9 person-days, with 3 engineers reviewing code over 3 days on commit `b7da01e`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual code review.

Project summary. Arianee allows brands to issue and manage NFTs for a variety of purposes. This review did not cover the entire protocol, but rather is limited to their full-privacy extension. Specifically, this report will focus on the associated ZK Circuits. The review covered the implementation of three zero-knowledge circuits that implement: (1) proof of ownership, which allows the proxy contract to associate a commitment with a specific token ID and a user to prove ownership of that commitment; (2) proof of credit registration, which allows the smart contracts to prove the valid issuance of credit; and (3) proof of credit, so that users can spend credits for the Arianee's SmartAsset store via a privacy-preserving proxy contract.

Code assessment. The Arianee developers provided the source code of the Full-Privacy Extension ZK Circuits for review. The source code appears to be mostly original code written by the Arianee developers. It contains some documentation in the form of READMEs and documentation comments within the provided circom code. To facilitate the Veridise auditors' understanding of the code, the Arianee developers shared a design document that provided an overview of the full privacy extension and were responsive to the Veridise auditors when asked questions about the design or intent of the code. The source code contained a test suite, which the Veridise auditors noted tested the core methods in which the verifiers would be used from Arianee's smart contracts.

Summary of issues detected. The audit uncovered 4 issues, 1 of which are assessed to be of critical severity by the Veridise auditors. Specifically: the commitments used by the `OwnershipVerifier` proofs may be repeated across tokens, and will therefore leak all brand information should a single token become identified (V-ARIA-VUL-001); and that credit notes may be stolen since the `CreditVerifier` only verifies credit existence and not credit ownership, meaning that, once created, anyone can use an existing `CreditNoteProof` by frontrunning a transaction that provides one (V-ARIA-VUL-002). The Veridise auditors also identified 2 warnings. The Full-Privacy Extension ZK Circuits developers have been notified of the issues and have indicated an intent to fix them.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Full-Privacy Extension ZK Circuits.

Use output signals. A number of public input signals could be declared as output signals in the circom circuits. For example, in the `OwnershipVerifier` circuit, the `pubCommitmentHash` input could be declared as an output signal, and the `commitmentHasher.out === pubCommitmentHash` constraint rewritten as `pubCommitmentHash <== commitmentHasher.out`. Since output signals are effectively public inputs in circom, this change does not affect the functionality of the circuit, but it would make the circuit construction more intuitive and make the circom-generated witness generator program easier to use.

Simplify array assignments. The version of circom used by the Full-Privacy Extension ZK Circuits developers (version 2.1.8) supports array assignments in place of for-loops. We recommend using this simplified syntax where applicable (e.g., in the `CreditVerifier`'s assignment of path elements to the `MerkleTreeChecker` subcomponent).

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Full-Privacy Extension ZK Circuits	`b7da01e`	Circom	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
July 8–July 10, 2024	Manual & Tools	3	9 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	0	0	0
Medium-Severity Issues	1	1	0
Low-Severity Issues	0	0	0
Warning-Severity Issues	2	2	2
Informational-Severity Issues	0	0	0
TOTAL	4	4	3

Table 2.4: Category Breakdown.

Name	Number
Frontrunning	1
Information Leakage	1
Constraint Optimization	1
Data Validation	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Full-Privacy Extension ZK Circuits's zero-knowledge circuits. In our audit, we sought to answer questions such as:

- Are the ZK Circuits properly constrained?
- Do the ZK Circuits leak their private inputs?
- Are the ZK Circuits properly used by the smart contracts?
- Do the ZK Circuits ensure privacy for their users?
- Are the cryptographic primitives in the ZK Circuits used properly?
- Is the checker implemented correctly?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom ZK Circuit static-analysis tool, Vanguard. This tool is designed to find instances of common zero-knowledge circuit vulnerabilities, such as underconstrained signals and unconstrained subcomponent signals.
- *Automated verification.* To verify safety properties of the zero-knowledge circuits, we leveraged our custom automated verification tool, Picus. This tool is designed to prove or find violations of determinism, which is an important safety property for zero-knowledge circuits.

Scope. The scope of this audit is limited to the `packages/privacy-circuits/src/circom` folder of the source code provided by the Full-Privacy Extension ZK Circuits developers in the `https://github.com/Arianee/arianee-sdk` repository, which contains the three main circuits of the Full-Privacy Extension ZK Circuits and all shared utilities.

Methodology. Veridise auditors reviewed the Full-Privacy Extension ZK Circuits design documentation provided by the Full-Privacy Extension ZK Circuits developers, inspected the provided tests, and read the Full-Privacy Extension ZK Circuits documentation. They then began a manual review of the code assisted by Veridise's custom automated tooling. At the beginning of the audit, the Veridise auditors met with the Full-Privacy Extension ZK Circuits developers to ask questions about the code. The Veridise developers asked further questions during the course of the audit over Slack.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-ARIA-VUL-001	CreditNoteProofs can be stolen	Critical	Fixed
V-ARIA-VUL-002	OwnershipProofs could identify issuers	Medium	Acknowledged
V-ARIA-VUL-003	1-indexed values should be 0-indexed	Warning	Fixed
V-ARIA-VUL-004	No check that commitment is not the zero leaf	Warning	Fixed

4.1 Detailed Description of Issues

4.1.1 V-ARIA-VUL-001: CreditNoteProofs can be stolen

Severity	Critical	**Commit**	b7da01e
Type	Frontrunning	**Status**	Fixed
File(s)	`creditVerifier.circom`		
Location(s)	Multiple		
Confirmed Fix At	095a366, f95260c		

The `ArianeeStore` contract requires that issuers pay credits to perform certain actions such as creating a SmartAsset, updating a SmartAsset, creating an event and creating a message. To allow private issuers to do so without revealing their identity, the developers created the `CreditNoteProof`, which acts as a proof of credit ownership that may be spent (defined and verified by the `CreditVerifier` circuit). It does so by proving that a given credit commitment is stored within a given Merkle tree of credit owners that is updated upon a credit purchase. In addition to proving that the commitment is in the given tree, the `CreditNoteProof` also contains a public nullifier that uniquely identifies a credit such that the credit commitment (and issuer's identity) is not revealed.

The code shown below uses the `CreditVerifier` to determine whether a `CreditNoteProof` may be spent. Notably, it first verifies if the given root is contained in the commitment Merkle tree's history. It then ensures the note is for the given credit type and that the credit has not been previously spent (by checking the nullifier). Finally, if the ZK proof verifies, the user will be allowed to spend the note. Importantly, there is no indication of the spender's identity.

```
function _verifyProof(CreditNoteProof calldata _creditNoteProof, uint256
    _zkCreditType) internal view {
    bytes32 pRoot = bytes32(_creditNoteProof._pubSignals[0]);
    require(isKnownRoot(pRoot), 'ArianeeCreditNotePool: Cannot find your merkle root'
    ); // Make sure to use a recent one

    uint256 pCreditType = _creditNoteProof._pubSignals[1];
    require(
        pCreditType == _zkCreditType,
        'ArianeeCreditNotePool: Proof credit type does not match the function
    argument `_zkCreditType`'
    );

    bytes32 pNullifierHash = bytes32(_creditNoteProof._pubSignals[2]);
    require(!nullifierHashes[pNullifierHash], 'ArianeeCreditNotePool: This note has
    already been spent');

    // We don't check the intent hash in the `ArianeeCreditNotePool` contract because
     it is already checked in the `ArianeeIssuerProxy` contract
    // and the `ArianeeIssuerProxy` contract is the only one allowed to call the `
    spend` function.

    require(
        creditVerifier.verifyProof(
            _creditNoteProof._pA,
            _creditNoteProof._pB,
            _creditNoteProof._pC,
            _creditNoteProof._pubSignals
        ),
        'ArianeeCreditNotePool: CreditNoteProof verification failed'
    );
}
```

Snippet 4.1: Definition of the `_verifyProof` function (defined in ArianeeIssuerProxy.sol) that is used to check if a `CreditNoteProof` may be spent.

Now, let us consider how these tokens are typically spent. The below snippet shows the `updateSmartAsset` function which will spend a `CreditNoteProof`. As can be seen, it is passed in as an argument to the function which invokes `trySpendCredit` to spend the credit. This function will simply invoke the above function and update state so that the Credit Note's nullifier cannot be spend again. As a result, anyone who has access to this proof *after it is generated* will be able to spend it.

```
function updateSmartAsset(
  OwnershipProof calldata _ownershipProof,
  CreditNoteProof calldata _creditNoteProof,
  address _creditNotePool,
  uint256 _tokenId,
  bytes32 _imprint,
  address _interfaceProvider
) external onlyWithProof(_ownershipProof, true, _tokenId) {
  trySpendCredit(_creditNotePool, ZK_CREDIT_TYPE_UPDATE, _creditNoteProof);
  store.updateSmartAsset(_tokenId, _imprint, _interfaceProvider);
}
```

Snippet 4.2: Definition of the `updateSmartAsset` function (defined in ArianeeIssuerProxy.sol) which spends `_creditNoteProof`.

Impact As the proof will be observable once it is posted to the mempool, anyone monitoring the pool could front-run the transaction to spend the note first.

Recommendation As users always provide a `CreditNoteProof` with an `OwnershipProof`, consider combining these into a single ZK proof as an `OwnershipProof` can only be spent in the call specified by the prover. This would require modifying the `CreditVerifier` to accept inputs for the `OwnershipVerifier` circuit and verify an `OwnershipProof` internally in addition to verifying the `CreditNoteProof`.

Developer Response We have added the same `pubIntentHash` signal in the **creditVerifier.circom** as in the **ownershipVerifier.circom**, and we are passing the originator's `msg.data` from the **ArianeeIssuerProxy** to the **ArianeeCreditNotePool**. This way, the **ArianeeCreditNotePool** is able to verify that the **CreditNoteProof** is legitimately spent with a matching intent.

4.1.2 V-ARIA-VUL-002: OwnershipProofs could identify issuers

Severity	Medium	**Commit**	b7da01e
Type	Information Leakage	**Status**	Acknowledged
File(s)	`ownershipVerifier.circom`		
Location(s)	Multiple		
Confirmed Fix At	N/A		

The privacy module is intended to protect the identity of SmartAsset issuers so that brand information is not exposed publicly. As such, rather than directly administering SmartAssets, issuers can instead perform the administration via the `ArianeeIssuerProxy`. They do so by submitting an `OwnershipProof` which the `OwnershipVerifier` circuit uses to prove that they know how to create the commitment associated with the asset. While Arianee includes the Token ID as a private input to the commitment, there is no enforcement in the circuit or contracts to do so. Private issues could therefore re-use commitments across tokens which is functionally extremely similar to a scheme where brands are assigned random addresses that they could use to administer their assets.

```
function tryRegisterCommitment(uint256 _tokenId, uint256 _commitmentHash) internal {
  require(
    commitmentHashes[_tokenId] == 0,
    'ArianeeIssuerProxy: A commitment has already been registered for this token'
  );
  commitmentHashes[_tokenId] = _commitmentHash;
}
```

Snippet 4.3: Definition of the `tryRegisterCommitment` function, defined in ArianeeIssuerProxy.sol), which associates tokens with commitments.

Impact If all tokens are associated with a single public commitment, anyone can track information about tokens associated with that commitment. If at any point in the future, information is leaked that links a brand to a token or commitment, anyone will have full knowledge of the activities of the issuer. Additionally, since it is likely that the owner of a SmartAsset (at the very least) knows information about the brand associated with the token, it is very likely that they will have full knowledge of the brand's activities once they purchase a token.

Recommendation The `OwnershipVerifier` circuit creates ownership proofs from 3 secrets that will likely remain static:

```
template OwnershipVerifier() {
    // Private inputs
    signal input sig[3];
    // VERIDISE: ...
```

Snippet 4.4: Snippet of the `OwnershipVerifier` definition.

Rather than creating commitments from these 3 secrets alone, also include additional information that must change with every token to ensure that different commitments are used. Note that

this information does not have to be private. In fact, we would recommend adding the token ID to the commitment hash and to validate the token ID when an ownership proof is checked.

```
template OwnershipVerifier() {
    // Private inputs
    signal input sig[3];

    // Public inputs
    signal input pubCommitmentHash;
    signal input pubIntentHash;
    signal input pubNonce;
    // VERIDISE: recommended changes:
    signal input tokenId;

    component commitmentHasher = Poseidon(4);
    commitmentHasher.inputs[0] <== sig[0];
    commitmentHasher.inputs[1] <== sig[1];
    commitmentHasher.inputs[2] <== sig[2];
    commitmentHasher.inputs[3] <== tokenId;

    // VERIDISE: ...
}
```

Snippet 4.5: An example of the recommended change to the `OwnershipVerifier` circuit.

This allows brands to maintain a single secret, but ensures that they must use unique commitments. Information associated with a single commitment therefore cannot reveal information about the brand in general. Instead, someone would need to compromise their secret.

Developer Response In the current scheme, all tokens are associated with a unique commitment. Each token commitment is a Poseidon hash of a unique (per-token) signature that depends on 3 non-static parameters, resulting in: **`Poseidon(EthSig(ChainId.SmartAssetContractAddress.TokenId))`**.

However, it is true that the current scheme does not enforce at the circuit level that a commitment is unique for a token. We enforce this in the upper layer (the @arianee/privacy-circuits SDK) of our solution. When an issuer uses our SDK to generate a commitment by calling `computeCommitmentHash`, we generate a unique signature according to the context `${chainId}.${smartAssetContractAddress}.${tokenId}`. It is very unlikely that an issuer will use an in-house solution to generate a commitment.

The `signal input sig[3];` represents the values (R, S, V) of this unique signature.

4.1.3 V-ARIA-VUL-003: 1-indexed values can be optimized if 0-indexed instead

Severity	Warning	**Commit**	b7da01e
Type	Constraint Optimization	**Status**	Fixed
File(s)	`creditRegister.circom`, `creditVerifier.circom`		
Location(s)	Multiple		
Confirmed Fix At	963b056, b6e7583		

The nullifier index is required to be in range $[1, 1000]$ and the credit type is required to be in range $[1, 4]$ by the `CreditVerifier` and `CreditRegister` circuit. According to the documentation on `ArianeeCreditNotePool.purchase` (defined in ArianeeCreditNotePool.sol), this 1-indexing scheme is done to make circuit implementation easier.

```
// VERIDISE: ...
// Ensure that nullifierDerivationIndex is in range 1-1000
// One nullifier can have up to 1000 different nullifierHashes
// In others words, one CreditNote can be spent up to 1000 times
component nulDerIdxGeqt = GreaterEqThan(16);
nulDerIdxGeqt.in[0] <== nullifierDerivationIndex;
nulDerIdxGeqt.in[1] <== 1;
nulDerIdxGeqt.out === 1;

component nulDerIdxLeqt = LessEqThan(16);
nulDerIdxLeqt.in[0] <== nullifierDerivationIndex;
nulDerIdxLeqt.in[1] <== 1000;
nulDerIdxLeqt.out === 1;

// Ensure that pubCreditType is in range 1-4
component creTypGeqt = GreaterEqThan(8);
creTypGeqt.in[0] <== pubCreditType;
creTypGeqt.in[1] <== 1;
creTypGeqt.out === 1;

component creTypLeqt = LessEqThan(8);
creTypLeqt.in[0] <== pubCreditType;
creTypLeqt.in[1] <== 4;
creTypLeqt.out === 1;
// VERIDISE: ...
```

Snippet 4.6: Snippet from the `CreditVerifier` template.

However, the circuit is easier to implement if the values are 0-indexed, as the `GreaterEqThan` components can be removed or in the case of the credit type, `Num2Bits(2)` could be used instead which is more efficient.

Impact This 1-indexing scheme makes the circuits more complex than needed and necessitates an additional conversion step in the smart contracts, adding additional complexity there as well.


```
// VERIDISE: ...
// Ensure that nullifierDerivationIndex is in range [0, 999]
// One nullifier can have up to 1000 different nullifierHashes
// In others words, one CreditNote can be spent up to 1000 times
component nulDerIdxLeqt = LessEqThan(16);
nulDerIdxLeqt.in[0] <== nullifierDerivationIndex;
nulDerIdxLeqt.in[1] <== 999;
nulDerIdxLeqt.out === 1;

// Ensure that pubCreditType is in range [0, 3]
component creTypLeqt = LessEqThan(8);
creTypLeqt.in[0] <== pubCreditType;
creTypLeqt.in[1] <== 3;
creTypLeqt.out === 1;
// VERIDISE: ...
```

Snippet 4.7: Snippet from `template CreditVerifier` changed to be zero-indexed.

Recommendation Use 0-indexed nullifier indices and credit types in both the smart contracts and zero-knowledge circuits.

Developer Response The 1-indexing scheme was originally used when the `creditType` was added to the commitment; the `creditType` therefore needed to be at least 1 to impact the final commitment number. However, since this is no longer used, the 1-indexing scheme has been removed and simplified to be 0-indexed across both the smart contracts and ZK circuits. The 1-indexing scheme has also been removed from the `nullifierDerivationIndex`.

4.1.4 V-ARIA-VUL-004: No check that commitment is not the zero leaf

Severity	Warning	**Commit**	b7da01e
Type	Data Validation	**Status**	Fixed
File(s)	`creditVerifier.circom`		
Location(s)	CreditVerifier		
Confirmed Fix At	41ef695, 4812488		

The `CreditNoteProof` allows users to spend credits without revealing their identity by proving their knowledge of a commencement that is stored in a given incremental Merkle tree. This incremental Merkle tree is initially populated entirely with a so-called zero leaf which is as new entries are added to the tree. As a zero-leaf is always in the Merkle tree unless it is full (and has not been inserted at any point), should someone learn a commitment associated with this value they could spend tokens without buying them.

```
template CreditVerifier(levels) {
    // VERIDISE: ...

    component commitmentHasher = CommitmentHasher();
    commitmentHasher.nullifier <== nullifier;
    commitmentHasher.secret <== secret;
    commitmentHasher.creditType <== pubCreditType;

    // VERIDISE: ...

    component tree = MerkleTreeChecker(levels);
    tree.leaf <== commitmentHasher.commitment;
    tree.root <== pubRoot;
    for (var i = 0; i < levels; i++) {
        tree.pathElements[i] <== pathElements[i];
        tree.pathIndices[i] <== pathIndices[i];
    }
}
```

Snippet 4.8: Snippet from the `CreditVerifier` template.

Impact Arianee uses a custom zero-leaf rather than 0, which is a common default. As such, it is very unlikely that this would occur but should it happen credits could be stolen.

Recommendation Consider checking that the given commitment does not correspond to the zero leaf.

Developer Response The `CreditVerifier` now asserts that the commitment hash is not equal to the zero leaf.

Glossary

Merkle Tree A cryptographic commitment to a list of values which can be opened at individual entries in the list. See https://en.wikipedia.org/wiki/Merkle_tree to learn more. 5

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 1

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See https://en.wikipedia.org/wiki/Zero-knowledge_proof for more. 1, 5, 17

ZK Circuit zero-knowledge circuit. 1, 5