Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

MANTA NETWORK

Manta-ZK Lib

Veridise Inc.

May 1, 2023

► **Prepared For:**

Manta Network
https://manta.network/

► **Prepared By:**

Kostas Ferles
Benjamin Sepanski
Alp Bassa
Daniel Domínguez Álvarez
Shankara Pailoor

► **Contact Us:** contact@veridise.com

► **Version History:**

Mon. 1, 2023	V1
Wed. 19, 2023	Initial Draft

Contents

1 Executive Summary

From Feb. 27, 2023 to April. 18, 2023, Manta Network engaged Veridise to review the security of their Manta-ZK Lib. The review covered several crucial arkworks circuits that implement Manta Network's L1 zero-knowledge protocol, whose goal is to enable on-chain privacy for applications. Veridise conducted the assessment over 35 person-weeks, with 5 engineers reviewing code over 7 weeks on commit `ba1a4b7`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Manta Network developers provided the source code of the Manta-ZK Lib contracts for review. To facilitate the Veridise auditors' understanding of the code, the Manta Network developers provided adequate documentation that formally specified the expected behavior of the system. The documentation came in a multitude of forms such as web documents with a high-level description of the system, Manta Network's whitepaper, a paper with the circuits formal specification, as well as READMEs and function comments in the codebase.

The source code contained a test suite, which covered all critical paths of the application. The Veridise team studied the suite extensively to understand the expected way of using each API in Manta Network's codebase.

Overall, Manta Network's codebase was of very high quality. The accompanied documentation was clear and easy to follow, and the test suite was extensive and exercised all security critical flows of the system.

Summary of issues detected. The audit uncovered 7 issues, none of which were assessed to be of high or critical severity by the Veridise auditors. The Veridise team assessed two issues as medium severity. Both of these issue were related to the configuration of cryptographic protocols (e.g., Schnorr signatures).

Recommendations. Even though our audit did not uncover any critical issues, we would still recommend fixing the medium severity issues at some point in the future. This recommendation is warranted by the fact that the sophistication of blockchain attackers is increasing monotonically. Therefore, a codebase that is not exploitable at this point of time might become exploitable in the future.

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
Manta-ZK Lib	`ba1a4b7` - `ba1a4b7`	Rust	Polkadot

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Feb. 27 - April. 18, 2023	Manual & Tools	5	35 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	0	0
Medium-Severity Issues	2	0
Low-Severity Issues	1	0
Warning-Severity Issues	2	1
Informational-Severity Issues	2	0
TOTAL	7	1

Table 2.4: Category Breakdown.

Name	Number
Cryptographic Vulnerability	3
Data Validation	2
Maintainability	2

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Manta Network's arkworks circuits. In our audit, we sought to answer the following questions:

- Do all circuits implement the expected behavior?
- Are all circuits properly constrained?
- Are all cryptographic protocols implemented and configured properly?
- Does the project utilize the arkworks framework properly?

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:

- *Fuzzing/Property-based Testing.* We leveraged fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalized the desired behavior of several components as assertions and then used the AFL fuzzing framework to determine if a violation of the specification can be found.
- *Differential Fuzzing.* We also employed differential fuzzing techniques in cases where encoding the desired behavior of a component as an assertion is infeasible, e.g., hash functions. For such cases, we found an existing (and audited) implementation of the same component and used it as an oracle for our differential fuzzer.

In total, we fuzz tested the API of 5 sub-components of Manta-ZK Lib. These components included the Manta Network's Poseidon hash implementation, digital signatures, Merkle trees, etc. We ran our fuzzers for a total of 182 hours combined across all components. Our tests did not uncover any bugs or crashes, which is an additional confirmation of the high quality of code produced by Manta Network developers.

Scope. The scope of this audit is limited to the following packages provided by the Manta Network developers:

- manta-crypto: includes cryptographic primitives used throughout the codebase.
- manta-accounting/transfer: defines a generic version of Manta Network's protocol.
- manta-pay: an instantiation of the above protocol that will be deployed.

The Manta Network developers also provided a detailed breakdown on which individual files must be audited for each of the above packages. For brevity, we omit listing each of the files in this document. We are happy to provide a complete list of the documents we audited upon request.

Limitations. Due to the scope of our audit, the recommendations provided in this report are limited to the functional specification provided by the Manta Network developers. The overall security of the system can be compromised if any component outside the scope of the audit is vulnerable. For Manta-ZK Lib, such components include, but are not limited to, the following:

- **Circuit deployment**: If the circuits are not deployed according to industry standards, i.e., following a secure trusted setup ceremony, the whole protocol can be at risk in case the information used in the creation of the common reference string (CRS) is leaked.
- **Front-ends**: Certain components in the codebase assume that the front-end uses Manta-ZK Lib's API correctly. Since the front-end is not in scope of the current audit, our team cannot provide any guarantees to that extent.

Methodology. Veridise auditors reviewed the reports of previous audits for Manta-ZK Lib, inspected the provided tests, and read the Manta-ZK Lib documentation. They then began a manual audit of the code assisted by the tools described above. During the audit, the Veridise auditors regularly met with the Manta Network developers to ask questions about the code and report progress of the audit.

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., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-MANZ-VUL-001	Schnorr signature scheme is vulnerable to bad r...	Medium	Open
V-MANZ-VUL-002	Repeated domain tags across Poseidon specs	Medium	Open
V-MANZ-VUL-003	Hash Function Bias	Low	Open
V-MANZ-VUL-004	Public Asset of opaque UTXOs can have non-defau	Warning	Open
V-MANZ-VUL-005	Unconstrained address_partition	Warning	Intended
V-MANZ-VUL-006	Use of RngCore trait without CryptoRng	Info	Open
V-MANZ-VUL-007	field_try_into! assumes field is larger than ta...	Info	Open

4.1 Detailed Description of Issues

4.1.1 V-MANZ-VUL-001: Schnorr signature scheme is vulnerable to bad randomness attacks

Severity	Medium	**Commit**	ba1a4b7
Type	Crypto Vulnerability	**Status**	Open
File(s)	`manta-crypto/src/signature/mod.rs`		
Location(s)	Schnorr::sign		

Textbook Schnorr signatures (and the Manta implementation) typically use a random nonce. However, if a bad source of randomness is used (e.g., using the same nonce twice or dependent randomness), this can leak the secret key.

Impact Leaked secret keys would compromise security for end users, since others can use their secret key to impersonate them.

Recommendation In practice, it is recommended to de-randomize the nonce. For instance, a hash of the message and the secret key (or xored with the secret key) can be used instead of the nonce. You can find more details under Section 5.1 in this paper.

Developer Response The developers acknowledged this issue and they are planning on fixing it in a future version of their circuit.

4.1.2 V-MANZ-VUL-002: Repeated domain tags across Poseidon specs

Severity	Medium	**Commit**	ba1a4b7
Type	Crypto Vulnerability	**Status**	Open
File(s)	`manta-pay/src/config/utxo.rs`		
Location(s)	All implementations of poseidon::hash::DomainTag		

All implementations of `poseidon::hash::DomainTag` in the affected file return the same domain tag (i.e., zero value). Domain tags should be distinct for each use of cryptographic hash function (see here). The Manta developers are clearly aware of this issue, as they have marked all these locations with a `FIXME` comment.

Impact Malicious users can reuse hash values across domains with identical configurations of the Poseidon hash.

Recommendation Please change the domain tag for each implementation as recommended by this RFC.

Developer Response The developers acknowledged this issue and they are planning on fixing it in a future version of their circuit.

4.1.3 V-MANZ-VUL-003: Hash Function Bias

Severity	Low	**Commit**	ba1a4b7
Type	Cryptographic Vulnera	**Status**	Open
File(s)	manta-pay/src/config/utxo.rs		
Location(s)	ViewingKeyDerivationFunction		

When obtaining the viewing key from the authorization key, the x and y coordinates of the authorization key are hashed to an element of the base field of the elliptic curve. This is reduced modulo the order of the large prime subgroup of the elliptic curve to obtain an element of the scalar field.

Snippet from ViewingKeyDerivationFunction

```
    fn viewing_key(
        &self,
        proof_authorization_key: &Self::ProofAuthorizationKey,
        compiler: &mut (),
    ) -> Self::ViewingKey {
        Fp(rem_mod_prime::<ConstraintField, EmbeddedScalarField>(
            self.0
                .hash(
                    [
                        &Fp(proof_authorization_key.0.x),
                        &Fp(proof_authorization_key.0.y),
                    ],
                    compiler,
                )
                .0,
        ))
    }
```

The size of the base field (ContraintField) is not divisible by the size of the scalar field (EmbeddedScalarField). This will introduce a modulo bias, as in the reduction elements can have 7 or 8 inverse images. As the ratio of the field sizes is small (around 8, the cofactor of the prime subgroup), the bias will be computationally noticeable. In general, to ensure the outcome to be indistinguishable from random, the ratio in bits (here 3) should be around the targeted security level (see Section 5 here)

Impact Bias acts like a side-channel and will pose a vulnerability if it can be exploited. It can leak secret information or impair privacy.

Recommendation The root cause of this issue is that the viewing key derivation function obtains only the first element of the Poseidon hash state which is then reduced modulo the order of the embedded scalar field (see snippet below).

```
    fn hash(&self, input: [&Self::Input; ARITY], compiler: &mut COM)
    -> Self::Output {
        self.hash_untruncated(input, compiler).take_first()
    }
```

We suggest the following key derivation scheme, which yields an insignificant bias and has the desired security level. Use the first two elements of the vector returned by `hash_untrucated`, say a_0 and a_1, forming the big integer $a_0 + a_1 \cdot p$ (where p is the size of the base field) and then reducing modulo the prime of the scalar field, let's say ℓ, as follows:

$$\Big((a_0 \mod \ell) + (a_1 \mod \ell) \cdot (p \mod \ell)\Big) \mod \ell$$

Developer Response The developers acknowledged this issue and they are planning on fixing it in a future version of their circuit.

4.1.4 V-MANZ-VUL-004: Public Asset of opaque UTXOs can have non-default value

Severity	Warning	Commit	ba1a4b7
Type	Data Validation	Status	Open
File(s)		N/A	
Location(s)		N/A	

For opaque transfers, `utxo.public_asset` can be set to any value.

For example, given a variable `to_public_pre` of type `Transfer` with an opaque UTXO, we may set the `public_asset` of its receiver to any value before generating the `TransferPost`:

```
// Set receiver utxo to an arbitrary asset
to_public_pre.receivers[0].utxo.public_asset = Asset::sample((), &mut rng);

// Generate a TransferPost from the modified transfer succeeds
let to_public = to_public_pre
    .into_post(
        FullParametersRef::new(&PARAMETERS, utxo_accumulator.model()),
        &PROVING_CONTEXT.to_public,
        Some(&spending_key),
        Vec::from([ALICE.into()]),
        &mut rng,
    )
    .expect("Unable to build TO_PUBLIC proof.")
    .expect("");
```

Impact Buggy front-ends may reveal secret information via the `utxo.public_asset` variable. In the worst case, a buggy frontend may set the public asset equal to the secret asset. However, subtler errors (such as setting the `public_asset` to an insecure hash of the secret asset) may be difficult to identify.

Other bugs may arise from frontends which incorrectly assume that a `utxo` is transparent when the `utxo.public_asset` is non-zero.

Finally, this gives some control to users over the nullifiers of opaque UTXOs. This doesn't seem to be exploitable with the current transaction shapes, but as the system evolves such freedom can be potentially exploitable.

Recommendation This can be easily prevented by requiring that `utxo.public_asset` is set to some fixed value for opaque transfers.

Developer Response The developers informed us that this was actually a feature of the system as they are planning to use the unconstrained public asset asset to store metadata. However, they did acknowledge that giving control to users over the nullifiers does expose some risk to the system. So, they will consider restricting the value of the public asset for opaque transactions.

4.1.5 V-MANZ-VUL-005: Unconstrained address_partition

Severity	Warning	Commit	ba1a4b7
Type	Data Validation	Status	Intended Behavior
File(s)	protocol.rs		
Location(s)	MintSecret::well_formed_asset		

The `FullIncomingNote` class contains a field `address_partition: u8` which is used to optimize the wallet synchronization process—reducing the number of notes it must open by a factor of 256.

However, this field is unconstrained. For example, given a `to_public_pre: Transfer`,

```
let mut to_public_pre = ToPublic::build(authorization, [sender_0, sender_1], [
    receiver_1], asset_3);
```

we are still able to create a (valid) post after mutating the address partition:

```
let old_address_partition = to_public_pre.receivers[0].note.address_partition;
let new_address_partition: u8 = old_address_partition.wrapping_add(1);
to_public_pre.receivers[0].note.address_partition = new_address_partition;

let to_public = to_public_pre
    .into_post(
        FullParametersRef::new(&PARAMETERS, utxo_accumulator.model()),
        &PROVING_CONTEXT.to_public,
        Some(&spending_key),
        Vec::from([ALICE.into()]),
        &mut rng,
    )
    .expect("Unable to build TO_PUBLIC proof.")
    .expect("");

// Ledger validation...
```

If set incorrectly, wallets will not display the asset as owned by the user. The `sync_with` method invokes `NoteOpen::open`: In this case, the user must then go to the chain in order to retrieve the

```
#[inline]
fn open(
    &self,
    decryption_key: &Self::DecryptionKey,
    utxo: &Self::Utxo,
    note: Self::Note,
) -> Option<(Self::Identifier, Self::Asset)> {
    let address_partition = // compute wallet address partition....
    if address_partition == note.address_partition {
        // Decrypt ....
    } else {
        None
    }
}
```

Snippet 4.1: Snippet from `impl NoteOpen for Parameters`:

asset.

Similarly, the field `light_incoming_note` is unconstrained. However, this is not easily fixed since the note is encrypted using AES.

Impact Bugs in frontends may seemingly lead to disappearing funds.

Malicious users may intentionally set `address_partition` to an incorrect value in order to trick other users into thinking that transactions failed. This may enable third-parties to perform social engineering attacks.

Similar issues arise if `light_incoming_note` is set incorrectly.

Recommendation Require that the `address_partition` is computed correctly via the zk constraints.

Further, ensure wallet users are able to easily "self-audit" (checking the `incoming_note` instead of the `light_incoming_note`) to search for transactions.

Developer Response This is the intended behavior of the system.

4.1.6 V-MANZ-VUL-006: Use of RngCore trait without CryptoRng

Severity	Info	**Commit**	ba1a4b7
Type	Maintainability	**Status**	Open
File(s)	See description		
Location(s)	See description		

Several encryption schemes rely on internal randomness in order to ensure that multiple encryptions of the same plaintext appear unrelated. This randomness is usually sampled via the `Sample` trait. For instance, the `Randomness` trait in hybrid.rs implements `Sample` as follows: In

```
impl<K, E, DESK, DR> Sample<(DESK, DR)> for Randomness<K, E>
where
    K: EphemeralSecretKeyType,
    E: RandomnessType,
    K::EphemeralSecretKey: Sample<DESK>,
    E::Randomness: Sample<DR>,
{
    #[inline]
    fn sample<R>(distribution: (DESK, DR), rng: &mut R) -> Self
    where
        R: RngCore + ?Sized,
    {
        Self::new(rng.sample(distribution.0), rng.sample(distribution.1))
    }
}
```

Snippet 4.2: Snippet from hybrid.rs

practice, `rng` is instantiated with a value of type `OsRng`, which satisfies `CryptoRng` (indicating that the RNG is cryptographically safe). However, the `CryptoRng` trait bound is not enforced by this implementation.

Impact Any random number generator used for encryption should be cryptographically secure. Without the `CryptoRng` trait bound, one must trace through the application to find which RNG is being used in order to ensure that the generator is (believed to be) cryptographically secure.

Recommendation Add the `CryptoRng` trait bound to any implementation of the `Sample` trait which may be invoked during the transfer protocol (i.e. for signatures, ephemeral keys, symmetric encryption randomness, and UTXO randomness).

This has the added benefit of making clear which `Sample` implementations are critical to cryptographic security, and which ones are primarily used for testing.

Developer Response The developers acknowledged this issue and they are planning on fixing it in a future version of their circuit.

4.1.7 V-MANZ-VUL-007: field_try_into! assumes field is larger than target data-type

Severity	Info	**Commit**	ba1a4b7
Type	Maintainability	**Status**	Open
File(s)	manta-crypto/src/arkworks/ff.rs		
Location(s)	field_try_into!		

The macro `field_try_into!` (which converts an element in an arkworks `PrimeField` to some unsigned integer type) assumes that the field is larger than the target datatype. In the above

```
if x < F::from(2u8).pow([$type::BITS as u64]) {
    let mut bytes = x.into_repr().to_bytes_le();
    bytes.truncate(byte_count($type::BITS) as usize);
    Some($type::from_le_bytes(into_array_unchecked(bytes)))
} else {
    None
}
```

Snippet 4.3: Snippet from `field_try_into!`

code snippet, if `F` is smaller than `2.pow([$type::BITS as u64])`, the exponentiation will wrap around and prevent some valid values from being converted from `F`.

Impact Any future instance of this macro with a target type larger than the field will produce an incorrect implementation. For instance, if `try_into_u256` were implemented using this macro, the implementation would prevent some valid conversions from 254-bit prime fields.

Although fallible conversion is unlikely to be implemented from a field into a larger type (and no such instance of the macro is currently used), instances of this macro may be created for consistency with other parts of the code base or as a convenience function.

Note also that the function created by this macro applies to ***all*** Arkworks `PrimeFields`. For instance, the full `ed_on_cp6_782` curve has a scalar field with a 373-bit prime subgroup. If future implementations rely on large fields such as this one, they may use this macro to create a `try_into_u256` method, which would be incorrect for smaller prime fields.

Recommendation Add an `or` to the `if` statement that allows successful conversion when the target type is larger than the field.

Developer Response The developers acknowledged this issue and they are planning on fixing it in a future version of their circuit.