ZKSECURITY

Audit of Risc0 Solana Programs: Verifier/Ownable/Router

Date: March 20, 2025

Introduction

On February 9th, 2025, Risc0 engaged zkSecurity to perform an audit of the Risc0 Solana programs. The engagement lasted for one week and focused on the risc0-solana repository.

The main components audited are the following:

- A **Groth16 proof verifier**, which implements the main logic for verifying Risc0 proofs.
- An **ownable** library, which provides a mechanism to implement ownership of a Solana account, and to transfer ownership with a two-step process.
- A **verifier router**, which is a Solana program that acts as a router and for different verifier implementation programs, and also provides an emergency stop mechanism.

Overall, we found the code to be well-structured and well documented. The codebase presents also a good level of test coverage for the different functionalities of the programs, using the anchor testing framework.

Overview of the Programs

We now give an overview of the different programs in the codebase.

Groth16 Proof Verifier

The verifier for the Groth16 proof system is very simple, consisting of:

1. Computing a commitment to the public input.
2. Followed by a single pairing product equation.

The verification of:

- A proof π
- Under the verification key vk
- With public input $\vec{a} \in \mathbb{Z}_q^k$.

Is as follows:

- $\pi = (A, B, C) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1$
- $vk = (\gamma, \delta, \alpha, \beta, \vec{IC}) \in \mathbb{G}_2 \times \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1^k$
- $PI = \langle \vec{a}, \vec{IC} \rangle \in \mathbb{G}_1$
- $e(A, B) = e(PI, \gamma) \cdot e(C, \delta) \cdot e(\alpha, \beta) \in \mathbb{G}_T$

In the Risc0 implementation, $A \in \mathbb{G}_1$ is negated and the equivalent check becomes:

$$1 = e(A, B) \cdot e(PI, \gamma) \cdot e(C, \delta) \cdot e(\alpha, \beta) \in \mathbb{G}_T$$

Because of its simplicity, the scope for errors is relatively limited compared to other proof systems with more complex verifiers.

The primary pitfalls for Groth16 proof verifiers are:

- Failing to check that the proof points ($A, C \in \mathbb{G}_1$, $B \in \mathbb{G}_2$) points are on the respective curves.
- Accepting public inputs with ambiguous encodings.

In the first case, the Solana `alt_bn128_pairing` syscall verifies membership of the points in the respective curves before performing the pairing operation. In the latter case, the Solana implement of Groth16 does not check that every public input is canonical: it's an integer less than the group order. We explore this in slightly more detail in the findings section, however, within the context in which the Groth16 verifier is used it does not lead to a security issue.

Ownable

The ownership mechanism allows a user to implement ownership for any Anchor account. The overall architecture is similar to the `Ownable2Step` contract in Solidity.

The ownership information is stored in the `Ownership` structure, which contains the current owner and the pending owner.

```
pub struct Ownership {
    /// The current owner's public key
    owner: Option<Pubkey>,
    /// The public key of the pending owner during a transfer, if any
    pending_owner: Option<Pubkey>,
}
```

The `Ownership` structure provides an implementation for all ownership operations. When this struct is created through `Ownership::new`, the `owner` field is set to the public key of the initial owner, and the `pending_owner` field is set to `None`. To transfer ownership, the owner must call `Ownership::transfer_ownership`, which sets the `pending_owner` field to the new owner's public key. The fact that the owner has called this function is checked using Anchor's mechanism of specifying the owner as a `Signer` of the transaction. The original owner can also cancel the ownership transfer by calling `Ownership::cancel_transfer`, which resets the `pending_owner` field to `None`.

The new owner can accept the ownership transfer by calling `Ownership::accept_ownership`, which sets the `owner` field to the `pending_owner` field, and the `pending_owner` field to `None`. The new owner can also renounce the ownership by calling `Ownership::renounce_ownership`, which is an irreversible operation that sets the `owner` field to `None`.

The `ownable` library also provides a macro to allow the automatic derivation of the `Ownable` trait for any account struct that contains an `Ownership` field. The macro automatically generates the `transfer_ownership`, `cancel_transfer`, `accept_ownership`, and `renounce_ownership` functions for the account struct. As a result, the user of the ownable library can easily define an ownable account by adding an `Ownership` field to the account struct and deriving the `Ownable` trait by using `#[derive(Ownable)]`.

Verifier Router

The verifier router is a program that allows the management of different verifier programs. The state of the router is kept in a `VerifierRouter` account, which is Ownable, and contains the ownership information and the number of verifiers currently registered in the router.

```
#[account]
#[derive(Ownable)]
pub struct VerifierRouter {
    pub ownership: Ownership,
    pub verifier_count: u32,
}
```

This information is kept in a PDA account, always derived by the router with seed `"router"` using the canonical bump.

There are two operations supported by the router: registering a new verifier and verifying a proof using a registered verifier. Every registered verifier is identified by a `selector`, which is just a u32 incremental value.

To **add a new verifier**, the authority must be the router PDA owner and must provide a verifier program that has the `router` PDA as the upgrade authority. To check this, the transaction also asks for a PDA account owned by the LoaderV3 program, which stores the information about the deployed program. The `upgrade_authority_address` field of the `ProgramData` account must be equal to the router PDA address. This allows the router to have the authority to close the verifier in the emergency stop, by virtually signing a CPI invocation to the LoaderV3 program on behalf of the router PDA.

```
#[derive(Accounts)]
#[instruction(selector: u32)]
pub struct AddVerifier<'info> {
    // ...

    /// Program data account (Data of account authority from LoaderV3) of the verifier
being added
    #[account(
        seeds = [
                verifier_program.key().as_ref()
            ],
            bump,
            seeds::program = bpf_loader_upgradeable::ID,
            constraint = verifier_program_data.upgrade_authority_address ==
Some(router.key()) @ RouterError::VerifierInvalidAuthority
    )]
    pub verifier_program_data: Account<'info, ProgramData>,

    /// The program executable code account of the verifier program to be added
    /// Must be an unchecked account because any program ID can be here
    /// CHECK: checks are done by constraint in program data account
    #[account(executable)]
    pub verifier_program: UncheckedAccount<'info>,

    // ...
}
```

When a verifier is added, a new PDA account is created, derived from seeds `"verifier"` and the `selector` value. This PDA contains information about the `selector` value and the verifier public

key.

To **verify a proof** with any of the registered verifiers, the user provides the `selector` of the verifier and the proof to be verified. The router then derives the corresponding verifier PDA account, retrieves the stored verifier public key, and invokes a CPI instruction to the verifier program.

Emergency Stop Mechanism

The emergency stop mechanism aims to permanently close a verifier program if a critical vulnerability is discovered. Once closed, the verifier program cannot be used anymore to verify proofs.

There are two ways to trigger the emergency stop mechanism: by the router owner or by providing an invalid proof. The router owner can unconditionally close any verifier registered in the router by invoking the `emergency_stop_by_owner` function. On the other hand, any authority who provides a proof for the zero `image_id` and `journal_digest` can invoke the `emergency_stop_with_proof` function. Creating a valid proof with zero `image_id` and `journal_digest` proves that the verifier is critically compromised, and it is possible to prove a false statement. The proof is verified by invoking a CPI call on the selected verifier: if the proof verification succeeds, the verifier program is closed.

In either case, the selected verifier program is permanently closed by invoking a CPI instruction to the LoaderV3 program. The router has the rights to close the verifier because the verifier program has the router PDA as the upgrade authority, and the router can virtually sign the CPI call on behalf of the router PDA.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	risc0-solana/solana-verifier/programs/groth16_verifier/src/lib.rs	Groth16 Verifier Reduces Public Inputs	**Low**
#01	risc0-solana/solana-verifier/programs/groth16_verifier/src/client.rs	Client Methods are Not Correct on Malicious Inputs	**Informational**
#02	risc0-solana/solana-verifier/programs/groth16_verifier/src/lib.rs	Point Negation is Only Sound for Reduced Points on the Curve	**Informational**
#03	risc0-solana/solana-verifier/programs/verifier_router/src/estop/mod.rs	The Emergency Stop Mechanism Proof of Exploitation Could Be More General	**Informational**

#00 - Groth16 Verifier Reduces Public Inputs

Severity: Low **Location:** risc0-solana/solana-verifier/programs/groth16_verifier/src/lib.rs

During verification of a Groth16 proof, the verifier reduces the public inputs to ensure they are within the order of the elliptic curve used in the proof:

```
fn verify_groth16<const N_PUBLIC: usize>(
    proof: &Proof,
    public: &PublicInputs<N_PUBLIC>,
) → Result<()> {
    ...

    let mut prepared = vk.vk_ic[0];
    for (i, input) in public.inputs.iter().enumerate() {
        let reduced = reduce_scalar_mod_q(*input);
        let mul_res = alt_bn128_multiplication(&[&vk.vk_ic[i + 1][..],
&reduced[..]].concat())
            .map_err(|_| error!(VerifierError::ArithmeticError))?;
        prepared = alt_bn128_addition(&[&mul_res[..], &prepared[..]].concat())
            .map_err(|_| error!(VerifierError::ArithmeticError))?
            .try_into()
            .map_err(|_| error!(VerifierError::ArithmeticError))?;
    }
    ...
}
```

The code of interest is `reduce_scalar_mod_q` which reduces an integer $\text{input} \in [0, 2^{256})$ modulo the order q of the elliptic curve. A potential problem with this approach is that public inputs now have multiple possible representations modulo q: letting the input be $\text{input} + q$ would still result in the verifier accepting the proof. This behavior is *different* from the Solidity implementation of the Groth16 verifier by Risc0, which explicitly checks that the public input is reduced.

Impact: Within the context of Risc0 this does not give rise to any security issues. In particular, the caller of `verify_groth16` will only ever call `verify_groth16` with public inputs that are already reduced.

Recommendation: Replace the reduction with an assertion that the input is already reduced modulo q. This avoids potential future issues where other parts of the codebase might assume the uniqueness of the public inputs (as `[u8; 32]` arrays). A classic example where this might be an issue is systems where the public inputs include a *spending tag* which is used to prevent relays/double spends, in which systems this issue could be critical if the attacker can use the multiple representations to execute double spends.

#01 - Client Methods are Not Correct on Malicious Inputs

Severity: Informational **Location:** risc0-solana/solana-verifier/programs/groth16_verifier/src/client.rs

The methods in `groth16_verifier/src/client.rs` are a set of helper utilities that help the client prepare inputs for on-chain verification. However, these methods do not behave correctly on malicious inputs. For instance, the `convert_g1` method:

```
pub(crate) fn convert_g1(values: &[String]) → Result<[u8; G1_LEN]> {
    if values.len() ≠ 3 {
        return Err(anyhow!(
            "Invalid G1 point: expected 3 values, got {}",
            values.len()
        ));
    }

    let x = BigUint::parse_bytes(values[0].as_bytes(), 10)
        .ok_or_else(|| anyhow!("Failed to parse G1 x coordinate"))?;
    let y = BigUint::parse_bytes(values[1].as_bytes(), 10)
        .ok_or_else(|| anyhow!("Failed to parse G1 y coordinate"))?;
    let z = BigUint::parse_bytes(values[2].as_bytes(), 10)
        .ok_or_else(|| anyhow!("Failed to parse G1 z coordinate"))?;

    // check that z == 1
    if z ≠ BigUint::from(1u8) {
        return Err(anyhow!(
            "Invalid G1 point: Z coordinate is not 1 (found {})",
            z
        ));
    }

    let mut result = [0u8; G1_LEN];
    let x_bytes = x.to_bytes_be();
    let y_bytes = y.to_bytes_be();

    result[32 - x_bytes.len()..32].copy_from_slice(&x_bytes);
    result[G1_LEN - y_bytes.len()..].copy_from_slice(&y_bytes);

    Ok(result)
}
```

Does not check that:

- The (x, y) elements are reduced modulo the characteristic of the base field.
- The (x, y) lies on the affine curve.

For instance, if $x \geq 2^{256}$ then `32 - x_bytes.len()` underflows causing a panic. Similarly, if $y \geq 2^{512}$ then `G1_LEN - y_bytes.len()` underflows causing a panic, if $2^{512} > y \geq 2^{256}$ then it overwrites part of the x-coordinate.

Impact: Because of the way these methods are used, no trust boundary is being crossed: within the wider system these methods are being invoked by the same party which computed the proof.

Recommendation: Make the validation logic in `groth16_verifier/src/client.rs` more robust, be as strict with validation as possible, even when the code does not appear security critical: the code might be used across a security boundary in the future, e.g. via an API endpoint accepting a proof and submitting it to the chain.

#02 - Point Negation is Only Sound for Reduced Points on the Curve

Severity: Informational **Location:** risc0-solana/solana-verifier/programs/groth16_verifier/src/lib.rs

The function used to negate a BN254 G1 curve point:

```
/// Negate a BN254 G1 curve point
pub fn negate_g1(point: &[u8; 64]) -> [u8; 64] {
    let mut negated_point = [0u8; 64];
    negated_point[..32].copy_from_slice(&point[..32]);

    let mut y = [0u8; 32];
    y.copy_from_slice(&point[32..]);

    let mut modulus = BASE_FIELD_MODULUS_Q;
    subtract_be_bytes(&mut modulus, &y);
    negated_point[32..].copy_from_slice(&modulus);

    negated_point
}
```

Takes (x, y) where $x, y \in [0, 2^{256})$ and returns $(x, q - y)$. Assuming that (x, y) is:

1. A point on the Weierstrass curve.
2. The integer y has $y < q$.

The result of this method is the negated point $(x, -y) \in \mathbb{E}[\mathbb{F}_q]$. However, the method never checks that the point lies on the curve G1 and that the inputs has been reduced, i.e. $y < q$. If either of these conditions are not met, the negation will not be correct, e.g. if $y > q$ then `subtract_be_bytes` underflows and produces an incorrect result.

Impact: this method, despite being in `lib.rs` (as opposed to `client.rs`) is not used by the on-chain verifier. It is only used by the client to prepare the inputs for submission on-chain. Hence, this finding has no immediate security implications.

Recommendation: Add checks to ensure that the point lies on the curve G1 and that the inputs have been reduced. Alternatively, move it to `client.rs` to show that it is assumed that the inputs to this function are honest.

#03 - The Emergency Stop Mechanism Proof of Exploitation Could Be More General

Severity: Informational **Location:** risc0-solana/solana-verifier/programs/verifier_router/src/estop/mod.rs

The emergency stop mechanism can be triggered either by the owner of the router account, or by any party that provides an invalid proof. The rationale is that if anyone can provide such a proof, then the verifier is considered compromised, and it is irreversibly closed.

To achieve this, any authority can provide a valid Groth16 proof for `image_id` and `journal_digest` set to an array of zeros.

```
let zero_array = [0u8; 32];
// [...]
let _ = groth_16_verifier::cpi::verify(verify_ctx, proof, zero_array, zero_array);
```

This catches the case where the proof system in the outer layer, i.e., Groth16, is completely compromised. However, there could be weaker bugs in the proof system that do not necessarily imply that an attacker can provide a valid proof for a zero `image_id` and `journal_digest`.

For example, if there is a bug in the inner layers, e.g., the soundness of one RiscV instruction in the zkVM is violated, then an attacker could provide an invalid proof (because the execution trace may be not correct), but not necessarily a proof for a zero `image_id` and `journal_digest`.

Impact: A user that finds a bug in the zkVM, which violates soundness but does not allow for the generation of a proof for a zero `image_id` and `journal_digest`, cannot trigger the emergency stop mechanism.

Recommendation: Consider requiring a more general proof of exploitation. One naive way to implement it would be to set up a RISC-V program test suite that aims to test the soundness of the zkVM, with substantial coverage of the instruction set. Then, the emergency stop mechanism could be triggered simply by exhibiting a valid proof for *any* of the programs in the test suite, yielding a different result than the one expected.