code4rena

Solana Foundation

Smart Contract Security Assessment

Audit dates: Aug 21 — Sep 23, 2025

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Solana Foundation smart contract system. The audit took place from August 21 to September 23, 2025.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded no High or Medium severity vulnerabilities.

Additionally, C4 analysis included 7 reports detailing issues with a risk rating of LOW severity or non-critical.

The issues presented here are linked back to their original finding, which may include relevant context from the judge and Solana Foundation team.

Scope

The code under review can be found within the C4 Solana Foundation repository.

The code in C4's Solana Foundation repository was pulled from:

- zk-sdk Repository: commit hash `aa22038e`
- ZK ElGamal Proof Program Repository: commit hash `afed4fc`
- Token-2022 Confidential Transfer ZK Repository: commit hash `2bca494`

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Low Risk and Non-Critical Issues

For this audit, 7 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Almanax** received the top score from the judge.

The following wardens also submitted reports: AgileGypsy, codegpt, johny7173, pa1va, ret2basic, and sarugami.

Table of Contents

- Token-2022 Program (extensions)
- zk-sdk (ZK ElGamal stack)
- ZK ElGamal Proof Program (native on-chain verifier)

Token-2022 Program (extensions)

Assertions on native accounts can panic instead of returning a controlled error

Location(s): `program/src/extension/confidential_transfer/processor.rs:#L451-L456`, also in `program/src/extension/confidential_transfer/processor.rs:#L570-L573`, and `program/src/extension/confidential_mint_burn/processor.rs:#L211-L214`

Impact: Using a native (wrapped SOL) token account in deposit/withdraw/mint-burn paths by a valid owner causes a panic, aborting the transaction without a clear error. No privilege escalation or funds loss; failure is confined to the current transaction.

Description (root cause): Uses `assert!(!token_account.base.is_native())` instead of returning a structured error.

```
assert!(!token_account.base.is_native());

assert!(!token_account.base.is_native());

assert!(!token_account.base.is_native());
```

Minimal Repro (conceptual):

1. Use a native wrapped SOL token account as input to deposit/withdraw/mint.
2. Path hits `assert!(!token_account.base.is_native())` and panics.

Remediation: `if token_account.base.is_native() { return Err(TokenError::NativeNotSupported.into()); }`.

Missing check_program_account before unpack/mutate (pattern across multiple call sites)

Location(s):

- `program/src/extension/confidential_transfer_fee/processor.rs:#L182-L221` (destination account)
- `program/src/extension/confidential_transfer_fee/processor.rs:#L326-L341` (mint in harvest-to-mint)

Impact: Fails late with less clear errors when non—Token-2022 accounts are supplied; adds compute and reduces observability. No privilege escalation or funds loss; failure is confined to the current transaction.

Description (root cause): Unpack/mutate without first calling `check_program_account(..)`. Remediation: Insert `check_program_account(owner)?` before unpack/mutation.

On-chain unwrap may panic in decryptable balance update

Location(s): `program/src/extension/confidential_transfer/account_info.rs:#L136-L140`

Impact: `.unwrap()` on `checked_add` can panic and abort the transaction if inputs overflow. No privilege escalation or funds loss; failure is confined to the current transaction.

Description (root cause): `new_decryptable_available_balance` uses `checked_add(...).unwrap()` instead of returning a structured error on overflow.

```
let new_decrypted_available_balance = current_available_balance
    .checked_add(pending_balance)
    .unwrap(); // total balance cannot exceed `u64`
```

Minimal Repro (conceptual):

1. Provide values such that `current_available_balance.checked_add(pending_balance)` returns `None`.
2. `.unwrap()` panics, aborting the transaction.

Remediation: Replace unwrap with error handling; propagate `TokenError::Overflow`.

Re-derive and check registry PDA before update (robustness, not security)

Location(s): `confidential-transfer/elgamal-registry/src/processor.rs:#L76-L100`

Impact: Fails late if a wrong (but program-owned) account is passed; explicit PDA check improves invariants and observability. No privilege escalation or funds loss; failure is confined to the current transaction.

Description (root cause): `process_update_registry_account` updates the registry without re-deriving the PDA for `wallet + program_id` and comparing to `elgamal_registry_account_info.key`.

```
// ... no PDA check here ...
let elgamal_registry_account_info =
next_account_info(account_info_iter)?;
let proof_context = verify_and_extract_context::<
    PubkeyValidityProofData,
    PubkeyValidityProofContext,
>(account_info_iter, proof_instruction_offset, None)?;
// ... mutate registry fields ...
```

Evidence / Reproduction: Passing a program-owned but non-canonical account yields fail-late behavior; PDA mismatch isn't surfaced early.

Remediation: Recompute PDA with `get_elgamal_registry_address_and_bump_seed(wallet, program_id)` and require equality before mutation.

Notes: QA/Low for clarity/consistency; not a bypass of PDA at creation.

Single authority can toggle harvest-to-mint without time delay or quorum**

Centralized Power / Trust Assumption: The configured `ConfidentialTransferFee` authority can enable/disable harvest-to-mint immediately without time-bound safeguards or multisig, affecting fee flows.

Location(s): `program/src/extension/confidential_transfer_fee/processor.rs:#L364-L391`, `#L394-L421`

Risk Rationale: A compromised or malicious authority could toggle fee routing in ways that disrupt operations or monitoring.

Recommended Controls: Consider multisig for the fee authority, a timelock for toggles, and eventing/monitoring on changes.

zk-sdk (ZK ElGamal stack)

Plaintext not zeroized after AEAD decryption

Location(s): `zk-sdk/src/encryption/auth_encryption.rs:#L76-L88`

Impact: Decrypted confidential balances may linger in heap memory, increasing risk of recovery via crash dumps, swap/snapshots, or memory disclosure on compromised hosts. No privilege escalation or funds loss; failure is confined to the current transaction.

Description (root cause): `AuthenticatedEncryption::decrypt` converts the `Vec<u8>` returned by `Aes128GcmSiv::decrypt` into a `u64` and returns it without zeroizing the plaintext buffer.

```
fn decrypt(key: &AeKey, ciphertext: &AeCiphertext) -> Option<u64> {
    let plaintext = Aes128GcmSiv::new(&key.0.into())
        .decrypt(&ciphertext.nonce.into(),
ciphertext.ciphertext.as_ref());

    if let Ok(plaintext) = plaintext {
        let amount_bytes: [u8; 8] = plaintext.try_into().unwrap();
        Some(u64::from_le_bytes(amount_bytes))
    } else {
        None
    }
}
```

Evidence / Reproduction: Decrypt any `AeCiphertext`; function returns early with `u64` while the heap `Vec<u8>` is dropped without zeroization.

Remediation: After copying out the 8 bytes, call `plaintext.zeroize()`; on mismatched length, zeroize before returning `None`. Optionally zeroize the temporary `[u8; 8]`.

Notes: Classified Low due to host-level/operational preconditions; the repo uses Zeroize in many places and these are outliers.

Ephemeral scalars in RangeProof::new are not zeroized

Location(s): `zk-sdk/src/range_proof/mod.rs:#L148-L175, #L168-L174`

Impact: Ephemeral secret scalars used to construct range proofs may persist in process memory and be recoverable via operational artifacts, modestly increasing exposure. No privilege escalation or funds loss; failure is confined to the current transaction.

Description (root cause): `RangeProof::new` samples several owned `Scalar` values (`a_blinding`, `s_L`, `s_R`, `s_blinding`) and returns without zeroizing them.

```
let a_blinding = Scalar::random(&mut OsRng);
let mut A = a_blinding * &(*H);
// ...
let s_L: Vec<Scalar> = (0..nm).map(|_| Scalar::random(&mut
OsRng)).collect();
let s_R: Vec<Scalar> = (0..nm).map(|_| Scalar::random(&mut
OsRng)).collect();
// ...
```

```
    let s_blinding = Scalar::random(&mut OsRng);
```

Evidence / Reproduction: After proof construction, function returns `Ok(RangeProof { ... })` without calling `.zeroize()` on these owned secrets or iterating to wipe vectors.

Remediation: Before returning, invoke `.zeroize()` on `a_blinding`, `s_blinding`, and iteratively zeroize all elements of `s_L`/`s_R` (e.g., loop and call `.zeroize()` for each).

Notes: Low severity; the repo generally zeroizes secrets elsewhere; these are outliers.

Sensitive key material derives Debug (risk of logging AES key)

Location(s): `zk-sdk/src/encryption/auth_encryption.rs:#L91-L96`

Impact: If downstream code logs `AeKey` with `{:?}` (directly or inside a struct), the AES key bytes are emitted to logs/telemetry, expanding exfiltration surface.

Description (root cause): `AeKey` derives `Debug`; the default formatter prints the inner `[u8; AE_KEY_LEN]`. While this repo does not log `AeKey`, the API shape encourages accidental leakage in consumers.

```
#[cfg_attr(target_arch = "wasm32", wasm_bindgen)]
#[derive(Clone, Debug, Zeroize, Eq, PartialEq)]
pub struct AeKey([u8; AE_KEY_LEN]);
```

Evidence / Reproduction: In a consumer, `tracing::debug!(?key);` or `format!("{:?}", key)` prints raw key bytes.

Remediation: Remove `Debug` from the derive or implement a redacting `Debug`:

```
impl core::fmt::Debug for AeKey {
    fn fmt(&self, f: &mut core::fmt::Formatter<'_>) -> core::fmt::Result
{
        f.write_str("AeKey([REDACTED])")
    }
}
```

Notes: Classified Low; no in-repo logging sinks observed, but hardening prevents accidental disclosure by consumers.

ZK ElGamal Proof Program (native on-chain verifier)

No QA findings for the in-scope files.

Reviewed entrypoints and proof verification glue; found no panics/unwraps or ownership gaps analogous to Token22 patterns.

Disclosures