CANTINA

Solaxy

Competition

August 5, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
High	*Must* fix as soon as possible (if already deployed) and can be triggered by any user without significant constraints, generating outsized returns to the exploiter. For example: loss of user funds (significant amount of funds being stolen or lost) or breaking core functionality (failure in fundamental protocol operations).
Medium	Global losses <10% or losses to only a subset of users, requiring significant constraints (capital, planning, other users...) to be exploited. For example: temporary disruption or denial of service (DoS), minor fund loss or exposure or breaking non-core functionality
Low	Losses will be annoying but easily recoverable, requiring unusual scenarios or admin actions to be exploited.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

Solaxy is the next evolution in blockchain technology, designed to unlock the full potential of the Solana ecosystem.

From Jun 11th to Jun 30th Cantina hosted a competition based on solaxy. The participants identified a total of **25** issues in the following risk categories:

- High Risk: 2
- Medium Risk: 5
- Low Risk: 15
- Gas Optimizations: 0
- Informational: 3

The present report only outlines the **high** and **medium** risk issues, which were fixed by Solaxy. The fixes were reviewed by Zigtur.

3 Findings

3.1 High Risk

3.1.1 Stale owner index lets attackers create "*ghost*" programs and exhaust resources

Submitted by 0xnija, also found by Rhaydden, Dystopia, Codertjay, 0xgh0st, hecok, 0x1982us, thisvishalsingh, Tokyo, 0xprincexyz and chainsentry

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: `owner_index` is never synchronised when accounts are updated or removed. As a result, program discovery relies on stale data, producing invisible (ghost) programs and an unbounded memory/CPU growth vector.

Finding Description: The project keeps two related data structures in every `SVMStorage` implementation:

- `accounts` – a `pubkey \rightarrow Account` map.
- `owner_index` – a `loader-id \rightarrow Vec<pubkey>` reverse index.

During initialisation the index is correct, but subsequent mutations break the invariant:

1. `set_account()` overwrites an existing entry in `accounts` without touching `owner_index`.

   When an account's `owner` field changes (e.g., program upgrade, closure or ordinary lamport transfer) the key stays under the old loader in the index.

2. `remove_account()` deletes the entry from `accounts` yet leaves the key behind in `owner_index`.

   Any normal purge (lamports == 0) therefore produces a dangling reference.

3. `get_program_accounts()` trusts the stale index, first collecting every pubkey listed under both loader IDs, then filtering each key with `get_account()`.

   - Keys that still exist but changed owners are silently skipped, so their programs are never cached.
   - Keys that no longer exist incur a failed lookup and remain in the vector, growing it unchecked.

An attacker can repeatedly create, close or re-assign accounts, causing thousands of dead entries. `Engine::fill_cache()` iterates the entire vector every time the cache is rebuilt, so iteration cost and memory footprint scale linearly with the number of orphaned keys - a denial-of-service vector. Meanwhile, legitimate programs that changed ownership disappear from the cache, breaking normal execution.

The root cause is that `owner_index` is updated only when the key did not exist before:

```
if self.get_account(pubkey)?.is_none() {
    self.set_owner_index(account.owner(), pubkey)?;
}
```

Because this check is "missing key" instead of "owner changed", any later modification leaves the index in an inconsistent state. No background task ever reconciles the map, so the divergence is permanent.

Impact: High - Incorrect program cache leads to failed transactions, while the unbounded growth of `owner_index` allows deliberate memory/CPU exhaustion.

Likelihood: High - Every program upgrade, closure, or ordinary lamport transfer that changes `owner` triggers the bug, and creating/removing accounts is trivial for an attacker.

Recommendation: Synchronise `owner_index` on every account mutation and purge.

Solaxy: Fixed in PR 287.

Zigtur: Fixed verified.

3.1.2 Incompatible token-account unpacking and strict size checks break SPL-Token-2022 balance queries

Submitted by Rhaydden

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: `get_token_account_balance` method incorrectly unpacks token account data. It uses `TokenAccount::unpack (from spl_token)` that fails for accounts from `spl_token_2022`.

If we use `TokenAccount::unpack` from `spl_token` to deserialize account data for accounts created with `spl_token_2022` that include extensions, this will not work as expected. Accounts with extensions have a different data layout that includes additional extension data after the base account struct, and the standard unpacking logic does not account for this extra data. More detailed information can be found in Solana's documentation about extensions and StackExchange question on how to deserialize extension data for mint accounts.

To properly handle accounts with extensions, we should use types like `StateWithExtensions` or `BaseStateWithExtensions` from `spl_token_2022`, which are made to parse both the base account data and any enabled extensions.

`get_token_account_balance` calls `TokenAccount::unpack(token_slice)`. `TokenAccount::unpack` is defined in the legacy `spl_token` crate and assume the oriiginal fixed length (165 B) layout. But when the queried account is an SPL-Token-2022 account that includes TLV extensions, its length exceeds `165 B` and the unpack routine returns an error (`InvalidAccountData`).

So the RPC therefore responds with `Invalid param: could not decode account data or Failed to unpack spl token account from data`, making the balance queries for all extended Token-2022 accounts unusable.

Also in `is_mint_account` function, for Token-2022 mints, the code requires `account.data.len() == spl_token_2022::state::Mint::LEN`. Albeit, mints that carry extensions are larger, so the equality check fail which will cause false negatives in ATA discovery (`get_ata_for_single_mint`, `get_atas_for_spl_program`) and other mint validations.

There are a couple of places where the extensions show up:

1. In `/solaxy/svm-rollup-cantina/crates/svm/tests/spl_token_2022.rs`:
   - `let extensions = vec![ExtensionType::TransferFeeConfig].`
   - `try_calculate_account_len::<Mint>(&extensions)` is used to size the mint.

   Any token account opened for that mint automatically carries the `TransferFeeAmount` extension, so its on-chain size is `165 + X` bytes.
2. `spl_associated_token_account::instruction::create_associated_token_account(..., &spl_token_2022::id())` will allocate the larger data size required by the mint's extensions. So after those instructions execute, both the mint and every token account for that mint are longer than 165 bytes.

But the `solaxy/svm-rollup-cantina/crates/tester/src/spl_token_2022.rs` tests still pass in the test harness because this was added:

```
fn unpack_token_account_data(data: &[u8]) -> Result<TokenAccount> {
    let trimmed = &data[..TokenAccount::LEN];      // <<<<<< forcibly shrink
    TokenAccount::unpack(trimmed)
}
```

Trimming the slice hides the length mismatch, so the test can still read the base 165 byte struct. The production RPC path (`svm/src/rpc/mod.rs`) does not do these trim. It calls `TokenAccount::unpack(data)` directly, so it fails as soon as it meets one of the extended accounts created above. In general, the impact is that balance queries (`getTokenAccountBalance`) fail for any SPL-Token-2022 account with extensions and ATA look-ups and token-account scans skip valid mints/accounts with extensions.

Recommendation: Use the Token-2022 `StateWithExtensions` unpacker for both token and mint accounts, and allow Token-2022 mints to be at least base size (not exact). or at least accept `data.len() >= 165` in the RPC implementation.

Solaxy: Fixed in PR 323.

Zigtur: Fixed verified.

3.2 Medium Risk

3.2.1 When an unsigned transaction sneaks in, everything blows up

Submitted by Rhaydden, also found by Kingnull, Codertjay and 0xerenyeager

Severity: Medium Risk

Context: sequencer.rs#L281

Description: In `send_transaction`, the signture check is:

```
// Verify transaction signatures without hashing the message
if !solana_tx.verify_with_results().iter().all(|res| *res) { // <<<
    return Err(SVMSequencerError::InvalidParams(
        "Failed to verify transaction signatures".to_string(),
    )
    .into());
}
```

For a transaction with zero signatures, `verify_with_results()` returns an empty `Vec<bool>`, and `.all(...)` over an empty iterator returns `true` (vacuous truth). An unsigned transaction therefore slips past this check. Later, in `send_tx`, the code blindly does:

```
async fn send_tx(&self, tx: Transaction) -> SVMSequencerResult<String> {
    let signature = tx.signatures[0].to_string(); // <<<
    let raw_message = borsh::to_vec(&tx)?;
    let fully_baked = RT::encode_with_solana_auth(raw_message);
    let _result = self.sequencer.accept_tx(fully_baked).await.map_err(|e| {
        SVMSequencerError::SovereignSequencerError(format!(
            "Failed to accept transaction: {e:?}"
        ))
    })?;

    Ok(signature)
}
```


Indexing `tx.signatures[0]` when signatures is empty triggers a runtime panic.

Recommendation: Before `verify_with_results()` is callled, do a quick sanity check: if `solana_tx.signatures` is empty, bail out with a clear "misssing signatures" error. That way we don't quietly let an unsigned tx pass by vacuous truth. Then in `send_tx`, swap `tx.signatures[0]` for something like `tx.signatures.first().ok_or(...)` so we never panic on empty lists.

Solaxy: Fixed in PR 317.

Zigtur: Fixed verified.

3.2.2 Crash & Poison via `simulateTransaction`

Submitted by potato, also found by Kingnull, Codertjay, AlmanaxAI and avgust1

Severity: Medium Risk

Context: mod.rs#L630-L635

Description: `VersionedTransaction::into_legacy_transaction()` returns `None` for all modern v0/v1 message formats, so the unconditional `unwrap()` will panic.

Impact:

- The panic happens while the thread still owns several `RwLock` write-guards (`working_set`, `accounts`, etc.).
- When a panic unwinds through an `RwLock`, that lock is "poisoned". Any future RPC call that touches the same lock will hit `.unwrap_infallible()` elsewhere and panic immediately.
- In practice a single request brings down every subsequent RPC call until the process restarts.

Recommendation: Replace `unwrap()` with graceful error handling:

```
let legacy = transaction
    .into_legacy_transaction()
    .ok_or_else(|| create_client_error("Versioned tx not supported"))?;
```

Solaxy: Fixed in PR 302.

Zigtur: Fixed verified.

3.2.3 Delayed epoch update causes RPC to show outdated block height and slot

Submitted by Rhaydden, also found by Codertjay

Severity: Medium Risk

Context: hooks.rs#L78-L92

Description: In `end_rollup_block_hook`, we log confirmed transactions and blocks thru `handle_accessory_state(&epoch_info)` before bumping and persisting `epoch_info.slot_index` and `epoch_info.block_height`. Because of this, every block and txn is stamped with the previous slot/height. RPC endpoints therefore report values one behind the true state, meaning that every confirmed transaction and block is written with the previous slot/block height so RPC querries will always return off by one values.

```
#[cfg(feature = "native")]
self.handle_accessory_state(  //  it uses stale epoch_info here
    state,
    processed_transactions,
    &epoch_info,  // Pre-increment values
    latest_blockhash,
    blockhash,
    count,
);

// Metadata is updated AFTER accessory state processing
epoch_info.slot_index += 1;
epoch_info.absolute_slot += 1;
epoch_info.block_height += 1;
self.epoch_info.set(&epoch_info, state).unwrap_infallible();
```


Recommendation: Just swap the calls. Increment then invoke `handle_accessory_state(&epoch_info, ...)`. I.e `epoch_info.slot_index += 1; epoch_info.block_height += 1; ... self.epoch_info.set(&epoch_info, state);` then `handle_accessory_state(&epoch_info, ...)`.

Solaxy: Fixed in PR 318.

Zigtur: Fixed verified.

3.2.4 Post transaction account state is lost during `TransactionExecutionResultProxy` round trip

Submitted by Rhaydden, also found by Khaye, Dystopia, hecok and Wankunda John

Severity: Medium Risk

Context: wrappers.rs#L206

Description: When serializing a `TransactionExecutionResult::Executed`, the proxy only records the list of touched program `Pubkey`s:

```
pub enum TransactionExecutionResultProxy {
    Executed {
        details: TransactionExecutionDetails,
        programs_modified_by_tx: Vec<Pubkey>,   // only the keys
    },
    NotExecuted(TransactionError),
}
```

On deserialization, it recreates a `TransactionExecutionResult::Executed` with each key mapped to `Arc::new(Default::default())`:

```
TransactionExecutionResultProxy::Executed {
        details,
        programs_modified_by_tx,
    } => TransactionExecutionResult::Executed {
        details,
        programs_modified_by_tx: programs_modified_by_tx
            .into_iter()
    .map(|key| (key, Arc::new(Default::default()))) // <<< we throw away all the real account data
            .collect(),
    },
    TransactionExecutionResultProxy::NotExecuted(err) => {
        TransactionExecutionResult::NotExecuted(err)
    }
}
```

All real `AccountSharedData` is throw away and replaced by empty, zero lamport dummies so we lose the true posttransaction state of those accounts.

Recommendation: Change the proxy to serialize both the key and its full account state via an `AccountProxy`, then rebuild each real `AccountSharedData` on deserialization instead of using `Default::default()`. Something like this:

```
  pub enum TransactionExecutionResultProxy {
      Executed {
          details: TransactionExecutionDetails,
-         programs_modified_by_tx: Vec<Pubkey>,
+         programs_modified_by_tx: Vec<(Pubkey, AccountProxy)>,
      },
      ...
  }

  impl From<TransactionExecutionResultProxy> for TransactionExecutionResult {
      fn from(proxy: TransactionExecutionResultProxy) -> Self {
          match proxy {
              TransactionExecutionResultProxy::Executed {
                  details,
-                 programs_modified_by_tx,
+                 programs_modified_by_tx,
              } => TransactionExecutionResult::Executed {
                  details,
-                 programs_modified_by_tx: programs_modified_by_tx
-                     .into_iter()
-                     .map(|key| (key, Arc::new(Default::default())))
-                     .collect(),
+                 programs_modified_by_tx: programs_modified_by_tx
+                     .into_iter()
+                     .map(|(key, acct_proxy)| (key, Arc::new(acct_proxy.into())))
+                     .collect(),
              },
              ...
          }
      }
  }
```

Solaxy: This is not relevant for us since it's a cache entry that is being stored here - not account state, so we chose to leave a default value.

Zigtur: Fixed verified.

3.2.5 Transaction Double-Counting in New Epochs

Submitted by Sparrow, also found by Rhaydden

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The transaction counting logic in the `end_rollup_block_hook` function has an issue where transaction counts are double-counted when initializing a new `EpochInfo`. This leads to inflated transaction metrics that misrepresent the actual activity on the chain.

Finding Description: In the `end_rollup_block_hook` method within `hooks.rs`, transaction counts are erroneously counted twice when a new epoch is initialized:

```
let mut epoch_info = self
    .epoch_info
    .get(state)
    .unwrap_infallible()
    .unwrap_or(EpochInfo {
        epoch: 0,
        slot_index: 0,
        block_height: 0,
        absolute_slot: 0,
        slots_in_epoch: 0,
        transaction_count: Some(processed_transactions.len() as u64), // First count here
    });

// Later in the same function
epoch_info.transaction_count =
    Some(epoch_info.transaction_count.unwrap_or_default() + count as u64); // Second count here
```

This creates an issue where the same transactions are counted both during epoch initialization and during the normal transaction counting logic, resulting in duplicate counts that artificially inflate transaction metrics.

Impact:

1. `getEpochInfo` RPC Endpoint: In `src/rpc/mod.rs`, this endpoint returns the entire `EpochInfo` structure which includes the double-counted `transaction_count` field. This endpoint is explicitly used by:
    - Block explorers to display epoch statistics.
    - Network monitoring tools that track chain activity.
    - Analytics platforms that aggregate blockchain metrics.
2. `getTransactionCount` RPC Endpoint: This endpoint in `src/rpc/mod.rs` directly exposes the accumulated transaction count. It's called by:
    - Test clients (`TestClient.svm_get_transaction_count` in test code).
    - Client applications tracking blockchain activity.

- Epoch Boundaries: Transaction counts will suddenly jump by the number of transactions in the first block of each new epoch.
- Historical Data: Each epoch's total transaction count will be artificially inflated by exactly the number of transactions in its first block.

Likelihood: This is a deterministic issue with 100% likelihood whenever a new epoch is initialized. The double-counting will occur in the following situations:

1. When the system first starts and initializes its first epoch.
2. At every epoch transition thereafter.

Proof of Concept: The issue can be verified by examining the `end_rollup_block_hook` function in `hooks.rs`:

1. When `epoch_info` is first initialized (or a new epoch begins), the `transaction_count` field is initialized with `processed_transactions.len() as u64`.
2. Later in the same function, regardless of whether this is a new epoch or not, the transaction count is incremented by the same amount: `epoch_info.transaction_count.unwrap_or_default() + count as u64`.
3. This means the same set of transactions is counted twice at every epoch initialization.

Recommendation: To fix this issue, the `transaction_count` field should be initialized to zero when creating a new `EpochInfo` instance, allowing the subsequent increment to correctly count transactions only once.

Solaxy: Fixed in PR 295.

Zigtur: Fixed verified.