OpenZeppelin | security

Stellar Contracts Library v0.5.0 Audit

October 28, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-09-09 To 2025-09-22
Languages	Rust (Soroban)

Total Issues	30 (23 resolved, 2 partially resolved)
Critical Severity Issues	3 (3 resolved)
High Severity Issues	5 (5 resolved)
Medium Severity Issues	15 (12 resolved, 2 partially resolved)
Low Severity Issues	7 (3 resolved)
Notes & Additional Information	0 (0 resolved)

Scope

OpenZeppelin diff-audited the OpenZeppelin/stellar-contracts repository with HEAD commit 028c429 against BASE commit 857bd36.

In scope were the following files:

```
packages
├── accounts
│   └── src
│       ├── lib.rs
│       ├── policies
│       │   ├── mod.rs
│       │   ├── simple_threshold.rs
│       │   ├── spending_limit.rs
│       │   └── weighted_threshold.rs
│       ├── smart_account
│       │   ├── mod.rs
│       │   └── storage.rs
│       └── verifiers
│           ├── ed25519.rs
│           ├── mod.rs
│           ├── utils
│           │   ├── base64_url.rs
│           │   ├── extract_from_bytes.rs
│           │   └── mod.rs
│           └── webauthn.rs
├── contract-utils
│   └── src
│       ├── lib.rs
│       ├── math
│       │   ├── fixed_point.rs
│       │   ├── i128_fixed_point.rs
│       │   ├── i256_fixed_point.rs
│       │   ├── mod.rs
│       │   └── soroban_fixed_point.rs
│       └── merkle_distributor
│           └── mod.rs
├── macros
│   └── src
│       ├── default_impl_macro.rs
│       └── lib.rs
└── tokens
    └── src
        ├── lib.rs
        ├── fungible
        │   ├── extensions
        │   │   └── vault
        │   │       ├── mod.rs
```

```
│   │       └── storage.rs
│   └── mod.rs
└── rwa
    ├── mod.rs
    ├── storage.rs
    ├── claim_issuer
    │   ├── mod.rs
    │   └── storage.rs
    ├── claim_topics_and_issuers
    │   ├── mod.rs
    │   └── storage.rs
    ├── compliance
    │   ├── mod.rs
    │   └── storage.rs
    ├── extensions
    │   ├── mod.rs
    │   └── doc_manager
    │       ├── mod.rs
    │       └── storage.rs
    ├── identity_claims
    │   ├── mod.rs
    │   └── storage.rs
    ├── identity_registry_storage
    │   ├── mod.rs
    │   └── storage.rs
    ├── identity_verifier
    │   ├── mod.rs
    │   └── storage.rs
    └── utils
        ├── mod.rs
        └── token_binder
            ├── mod.rs
            └── storage.rs
```

System Overview

The Stellar Contracts Library has undergone significant enhancements in the release under review, introducing three major new modules that expand the library's capabilities for sophisticated blockchain applications. This audit assesses the smart accounts framework, the vault extension for fungible tokens, and the comprehensive Real World Assets (RWA) token implementation.

Smart Accounts Framework

The new smart accounts module (`packages/accounts/`) introduces a flexible and modular framework that transcends traditional account management. Unlike simple EOAs, this system enables advanced authentication and authorization patterns through composable components.

The architecture centers around context rules, which define authorization policies for different types of operations. Each context rule can contain multiple signers and policies, with built-in support for expiration times and dynamic management. The system supports a maximum of 15 signers and 5 policies per rule, providing a balance between flexibility and performance.

Key capabilities include:

- **Multi-signature verification** supporting Ed25519 and WebAuthn authentications
- **Policy-based authorization** with pre-built policies for spending limits, simple thresholds, and weighted threshold schemes
- **Context-aware rules** that adapt authorization requirements based on the type of operation being performed

The WebAuthn integration is particularly noteworthy, enabling hardware security key and biometric authentication directly on-chain, while intentionally omitting certain WebAuthn validations (origin, RP ID hash, signature counter).

Vault Extension for Fungible Tokens

The vault extension (`packages/tokens/src/fungible/extensions/vault/`) implements the ERC-4626 tokenized vault standard, enabling fungible tokens to represent

shares in an underlying asset pool. This module provides a sophisticated mechanism for creating yield-bearing tokens and liquidity pools.

The vault system maintains automatic conversion between shares and underlying assets based on the total supply and assets under management. It provides both high-level functions with built-in safety checks and low-level primitives for custom implementations. The mathematical operations handle edge cases like initial deposits (when total supply is zero) and implement proper rounding directions to prevent manipulation.

Critical design features:

- **Flexible deposit/withdrawal mechanisms** supporting both asset-based and share-based operations
- **Preview functions** for simulating operations before execution
- **Maximum limits enforcement** to prevent overflow conditions
- **Event emissions** for all major operations to support off-chain indexing

Real World Assets (RWA) Token Implementation

The RWA module (`packages/tokens/src/rwa/`) represents the most comprehensive addition, implementing a T-REX-inspired (Token for Regulated Exchanges) framework for security tokens. This implementation addresses fundamental limitations of traditional ERC-3643 designs while providing a more flexible and efficient architecture for tokenizing real-world assets.

Architectural Innovations

The module departs significantly from the rigid, bureaucratic structure of traditional RWA implementations. Instead of enforcing a tightly-coupled web of interdependent contracts following the ERC-734/735 pattern, this design embraces modularity through deliberate abstraction:

- **Identity Agnosticism**: The core RWA contract only requires two essential functions: `verify_identity()` and `can_transfer()`. This minimal interface allows for diverse identity verification approaches - from claim-based systems and Merkle tree proofs to zero-knowledge protocols or even off-chain OAuth bridges. The architecture explicitly avoids hardcoding any specific identity paradigm.

- **Simplified Contract Hierarchy**: The traditional three-layered structure (Token → IdentityRegistry → IdentityRegistryStorage) has been flattened. The new

`IdentityVerifier` trait can be implemented either as part of the token contract or as a standalone module, reducing cross-contract calls and improving gas efficiency.

- **Unified Claim Management**: The previously separate `ClaimTopicsRegistry` and `TrustedIssuersRegistry` have been merged into a single `ClaimTopicsAndIssuers` contract. This consolidation eliminates redundant mappings, reduces cross-contract calls from n+1 to 1 during transfers, and prevents configuration inconsistencies.

Compliance Framework

The compliance module implements a hook-based system that maintains backwards compatibility while enabling advanced regulatory controls:

- **Event-Driven Hooks**: Five distinct hook types (`Transferred`, `Created`, `Destroyed`, `CanTransfer`, and `CanCreate`) allow modules to either validate operations proactively or react to completed events. This separation between read-only validation and state-modifying operations optimizes gas consumption.
- **Modular Compliance Rules**: Multiple compliance modules can be registered to the same hook, enabling layered regulatory checks. Modules execute in sequence with short-circuit evaluation - if any module rejects a transfer, the entire operation fails.
- **Reusable Components**: Unlike traditional designs where each token requires its own compliance infrastructure, modules can be shared across multiple tokens with identical regulatory requirements or token partitions (tranches).

Advanced Token Controls

The implementation provides comprehensive administrative capabilities required for regulated securities:

- **Granular Freezing Mechanisms**: Support for both full address freezing and partial balance freezing, enabling precise regulatory enforcement without affecting an investor's entire position.
- **Identity-Linked Recovery**: Lost wallet recovery tied to verified identities, allowing legitimate investors to recover assets while maintaining security.
- **Forced Transfers**: Court-ordered or regulator-mandated transfers can be executed while maintaining full audit trails.
- **Pausable Operations**: Emergency pause functionality for the entire token during critical events.

Role-Based Access Control (RBAC)

Instead of the traditional owner/agent model, the implementation uses fine-grained RBAC, which:

- allows for defining multiple specialized roles (minter, burner, freezer, and recoverer)
- eliminates redundant agent management across contracts
- enables principle of least privilege by assigning minimal necessary permissions
- supports role hierarchies and delegation patterns

Document Management Extension

An optional module following ERC-1643 design principles provides:

- on-chain document storage with versioning and history retention
- batch operations for efficient document updates
- role-based management aligned with transfer agent responsibilities
- integration with the token's identity verification system for document access control

Security Model and Trust Assumptions

The security guarantees of the contracts built using these new modules depend on proper implementation and configuration by developers, as well as several foundational trust assumptions.

Environmental Dependencies

- **Platform Security**: The modules assume that the underlying Soroban runtime, WASM compiler, and Stellar SDK operate correctly and securely. Any vulnerabilities in these foundational components could compromise contract security.
- **Cryptographic Primitives**: The smart accounts module relies on the security of Ed25519 and secp256r1 (for WebAuthn) signature schemes. The correctness of signature verification is critical for authentication.

- **External Contract Interfaces**: RWA tokens depend on external contracts for identity verification and compliance. The security of the overall system requires these external contracts to be correctly implemented and maintained.

Developer Responsibilities

The library provides powerful primitives that require careful implementation:

Smart Accounts Configuration:

- Developers must properly initialize context rules with an appropriate combination of signers and policies.
- The spending limit policy requires careful consideration of period length and limit amounts.
- WebAuthn implementations should include expiry timestamps in signed data for enhanced security.
- Cross-contract authentication requires understanding that contract addresses automatically pass `require_auth()` checks.

Vault Security:

- The vault's low-level functions intentionally bypass authorization for flexibility. Developers must implement proper access controls using Ownable or Access Control patterns.
- Initial vault configuration must set the correct underlying asset address.
- Share/asset conversion logic assumes honest reporting of total assets by the vault contract.

RWA Implementation:

- Identity verification and compliance contracts must be deployed and configured before token operations.
- The modular compliance system requires careful module registration to avoid conflicting rules.
- Frozen addresses and recovery operations need robust off-chain processes for identity verification.
- Document management extensions should validate document authenticity off-chain.

Operational Assumptions

- **Time-based Security**: Spending limits and context rule expiration rely on ledger sequence numbers for time measurement, assuming consistent block production.
- **Identity Claims**: The default RWA implementation trusts designated claim issuers for identity verification. The security model assumes that these issuers perform proper KYC/AML procedures.
- **Compliance Module Trust**: Once registered, compliance modules can block or allow transfers. As such, administrators must carefully vet modules before registration.
- **Recovery Mechanisms**: Both smart accounts and RWA tokens include recovery features that, if misconfigured, could allow unauthorized access to funds.

Cross-Contract Security Considerations

The modular architecture introduces specific security considerations:

- Smart account policies execute with the account's authority and must be trusted code.
- RWA tokens making external calls for verification must handle potential reentrancy attempts.
- Vault operations involving external token contracts must account for potential callback patterns.
- Compliance modules registered to multiple tokens could create systemic risks if compromised.

These modules significantly expand the Stellar Contracts Library's capabilities while maintaining the security standards expected for financial infrastructure. However, their power and flexibility demand careful implementation and ongoing security considerations from developers deploying production systems.

Candidate Detectors for Static Analysis Tools

During the audit, several categories of issues were identified that can be effectively flagged by static analyzers:

Access Control

- **missing-auth-check** — Functions modifying state without `require_auth` / `require_auth_for_args`.

- **inconsistent-auth-pattern** — Mixed or inconsistent authorization patterns across similar functions.
- **operator-confusion** — Operator parameter not aligned with the actual asset provider.

Storage Management

- **storage-ttl-inconsistency** — Inconsistent TTL values for related data.
- **missing-storage-initialization-check** — Accessing storage before initialization.
- **missing-ttl-management** — Lack of TTL enforcement for persistent data.

RWA Compliance

- **compliance-module-bypass** — Transfers bypassing compliance checks.
- **compliance-hook-mismatch** — Incorrect hook registration or execution.

Cross-Contract Interactions

- **unchecked-external-calls** — External calls without result validation.
- **malicious-contract-injection** — User-supplied contract addresses not validated.
- **circular-dependency-risk** — Risk of circular dependencies between contracts.

Mathematical Operations

- **unchecked-arithmetic** — Risk of unwanted panics from unchecked math.
- **division-by-zero** — Missing denominator validation.
- **precision-loss** — Integer division occurring before multiplication.

Event Emissions

- **missing-critical-events** — State changes without corresponding event emissions.
- **event-data-mismatch** — Events emitted with incorrect or inconsistent parameters.

Denial of Service

- **unbounded-loops** — Loops over user-controlled arrays.
- **state-bloat** — Unbounded storage growth (e.g., dynamic values in persistent or instance storage).

Critical Severity

C-01 Unbounded Inflation via Self-Recovery in `recovery_address`

The `recovery_address` function is intended to force-transfer an investor's tokens from a lost wallet to a new wallet after identity verification and RBAC checks on the operator. However, when `lost_wallet` equals `new_wallet`, the function reads the same account's balance twice and then writes once, effectively doubling the balance without updating the supply accounting.

With `A` as both wallets and `b = Base::balance(A)`:

1. Read `lost_balance = b`.
2. Compute `new_balance = Base::balance(A) + lost_balance = b + b = 2b`.
3. Write `Balance(A) = 0` then `Balance(A) = 2b` (same key) → final balance `2b`.

No supply variable is updated here, so:

- Sum of balances inflates by `+b`.
- Any tracked `total_supply` (e.g., inside `Base`) stays unchanged, breaking invariants.

Since the `recovery_address` function has no authorization and only checks `verify_identity(new_wallet)`, any caller controlling an eligible address with a non-zero balance can call repeatedly with `(A, A)` to mint exponentially: `b → 2b → 4b → 8b …` (until `i128` wrap/limits). It also emits `transfer(A, A, b)`, further confusing indexers while the root issue is silent inflation.

Consider requiring `lost_wallet != new_wallet` and route balance changes through the canonical update path (e.g., `Base::update`) to preserve invariants, hooks, and `total_supply`.

Update: *Resolved in pull request #432.*

C-02 Missing Same-identity Check in `recovery_address` Enables Malicious Recovery

The `recovery_address` function only verifies the new wallet's identity and then transfers all funds from `lost_wallet` to `new_wallet`. It never ensures that both wallets belong to the same investor identity and that the `investor_onchain_id` parameter is not used for any binding. In practice, the parameter appears to be used only for event emission.

Without enforcing same-identity linkage, a caller can nominate any `lost_wallet` and a `new_wallet` that merely satisfies generic identity checks, and drain the `lost_wallet`'s entire balance to the `new_wallet`. This violates the intended "identity-preserving" recovery flow described in the design, and breaks the trust assumption that recovery only migrates balances within the same investor's identity. Additionally, the function moves funds out of `lost_wallet` but leaves that address in a clean, unfrozen state and still associated with the original `investor_onchain_id` in the identity system. This creates a "zombie account": an address known to be compromised yet still treated as valid. An attacker retaining the compromised keys could continue to receive funds to that address and could even trigger a fraudulent recovery to move funds back from the `new_wallet` to `lost_wallet`.

Consider adding an explicit identity-continuity check before moving funds by fetching both wallets' identities from `IdentityRegistryStorage` and requiring equality. In addition, as part of remediation, consider implementing the following recommendations:

- Either remove `investor_onchain_id` entirely or enforce that it matches the identities derived for both `lost_wallet` and `new_wallet`.
- After successful recovery, update `IdentityRegistryStorage` to sever or neutralize the `lost_wallet` association, and remove the mapping (`remove_identity(lost_wallet, operator)`).

Update: *Resolved in pull request #380.*

C-03 Signer Aliasing via Non-Canonical Key Bytes

Smart accounts combine signers, context rules, and pluggable policies. Delegated signers offload cryptographic checks to stateless verifier contracts. In this design, rules can compose multiple signers and policies, and the matching algorithm succeeds only when a rule's requirements are met. Delegated verification is a first-class path: a signer is modeled as

`(verifier_contract, key_bytes)`, and the framework relies on those verifiers being strict and trustworthy.

In the current verifiers, both Ed25519 and WebAuthn accept `key_data` that is at least the required length and then silently truncate to the first 32 bytes (Ed25519) or 65 bytes (WebAuthn). As the smart-account layer treats the full `(verifier, key_bytes)` as the signer identity, a malicious user can register multiple "different" delegated signers that share the same real public key but differ only in trailing junk bytes. At verification time, all of these aliases would resolve to the same underlying key, so a single signature can be replayed across multiple supposed signers, satisfying simple thresholds or inflating weight in weighted policies. The practical impact is severe: malicious users can create backdoors or "fake signers" that appear to diversify authorization when in reality there is only one, undermining multisig guarantees and allowing threshold or weight requirements to be met with fewer independent keys than intended.

Consider enforcing canonical key material end-to-end. Verifiers must strictly reject `key_data` whose length is not exactly the scheme's size (32 bytes for Ed25519 and 65 bytes for uncompressed P-256/WebAuthn) rather than accepting longer inputs and truncating.

Update: *Resolved in pull request #384.*

High Severity

H-01 TTL Mismatch Enables Duplicate Modules And Can Brick Removal

The compliance contract acts as a dispatcher that pulls the list of registered module addresses for a given hook and then calls those modules during validation (`canTransfer`, `canCreate`) and after-effects (`transferred`, `created`, and `destroyed`).

The issue arises from a storage-lifetime mismatch between two pieces of persistent state that together represent "module X is registered for hook H." One entry stores the vector of module addresses per hook (the list), and another entry stores a per-module "existence flag" for that hook. The list's TTL is extended on every read path (because hooks fetch it frequently), but the per-module existence flag's TTL is only extended when a specific helper is called. In normal operation, hooks read the list often, keeping it fresh, while the existence flags for those same modules quietly expire. In Soroban, persistent entries that expire are archived and can be

recovered, but they are not automatically restored: they temporarily disappear from the contract's live state until explicitly repaired.

This means that the list continues to include a module address while the corresponding existence flag gets aged out. The next attempt to add that same module passes the "existence" check and appends a duplicate to the list. Later, when attempting removal, the flow first checks that the (now missing) existence flag and fails, even though the module still appears in the list. Otherwise, if the removal proceeds against one copy, it deletes the existence flag while leaving other duplicates behind, blocking further removals. Over time, this desynchronization leads to the duplicate execution of compliance logic, difficulty or impossibility in removing a module, and premature exhaustion of the per-hook module bound.

The issue affects both the operation and security of the system. Duplicate entries cause repeated module invocation, amplifying side effects and gas costs, and making behavior non-deterministic to operators. Inability to remove a module from the active list, despite it still being present, means that a misconfigured or malicious module can remain active and block token operations or enforce unintended rules.

Consider eliminating the possibility of drift and enforcing a single source of truth or, at minimum, synchronizing the lifetimes. Concretely, extend TTL for the per-module existence entries whenever the list is read for a hook and reject additions if the module already appears in the list, not only if the existence flag is present. Upon removal, operate on the list as the decisive source and clear the existence flag only when no copies remain, periodically deduplicate the list on read or at admin-controlled maintenance points, and consider replacing the dual-structure entirely by deriving the list from the per-module map (or vice-versa) so that registration status is represented in one place.

Update: *Resolved in pull request #411.*

H-02 Bypassing Signature Check in Spending Limit Policy

The spending limit policy is called by the smart account contract during `__check_auth` to ensure that transfers do not exceed configured limits. However, the `_authenticated_signers` parameter containing verified signers is ignored in both `can_enforce` and `enforce`: these functions only validate the transfer amount, not the presence of valid signatures.

As a result, when a rule specifies non-zero signers, their count is not checked. `get_validated_context` will still call `can_enforce` with zero valid signers, which

approves transfers as long as limits are not exceeded. This allows an attacker to repeatedly drain funds from the account up to the configured spending limit without any authorization.

This reflects a broader design issue in `get_validated_context`:

- Without policies, all rule signers must sign (like an n-of-n multisig on the rule level).
- With policies, signer checks are delegated to policies. Meaning, a multisig with a `spending_limit` must also include an authentication policy (e.g., `simple_threshold`).

This complexity risks operational errors and compromises. Since policies are shared external contracts, consider keeping the entire signature-validation logic within the smart account itself against its own stored valid signers rather instead of outsourcing it.

Update: *Resolved in pull request #408.*

H-03 Silent Allow-All Rule After Removing the Last Signer or Policy

The smart-account design evaluates rules newest-first, falling back to defaults, and treats policies as all-or-nothing checks. If a rule has no policies, all its signers must authenticate. The spec also states that each rule must contain at least one signer or one policy, which is the intended invariant for safe matching.

When a rule is created, the implementation enforces this invariant. However, when signers or policies are later removed, there is no validation to prevent a rule from ending up with neither signers nor policies. In this state, the "no-policy" matching branch considers the empty set of signers as fully authenticated by vacuous truth, so the rule matches automatically and authorizes the context without any signatures. If such a rule is the most recent for its scope, or applies broadly as a default, it silently becomes a permit-all and can bypass the intended authorization for subsequent operations. This contradicts the documented requirement that a rule must always include at least one signer or policy and relies on the matching behavior where no-policy rules succeed only when all signers authenticate.

Consider re-enforcing the invariant after every mutation that changes a rule's composition. Specifically, on any signer or policy removal, immediately validate the updated rule and reject the change if it would leave the rule with neither signers nor policies. This preserves the documented safety property, prevents vacuous matches, and keeps runtime behavior aligned with the design's matching guarantees.

Update: *Resolved in pull request #409.*

H-04 Identity Verification Fails Prematurely When a Trusted Issuer Lacks a Claim

The RWA token enforces identity and compliance controls as a prerequisite for asset movements and issuance. Identity verification is the gate that ensures a wallet's on-chain identity holds the required set of claims (for example, KYC or AML) and that at least one trusted issuer is authorized to attest each required topic. This check underpins regulatory posture, investor onboarding, and day-to-day operations by preventing transfers and mints when identities are not properly attested.

The `verify_identity` function iterates over required claim topics and, for each topic, checks multiple trusted issuers. It attempts to fetch the claim for the first issuer and treats a missing claim as a hard failure, which aborts verification for that topic before evaluating the remaining issuers. As a result, the "any of the trusted issuers can satisfy the topic" intent is not honored. Bookkeeping to detect the "last issuer" is present but rendered ineffective by this early failure behavior:

- **False Rejections:** Compliant users with valid claims from later issuers are incorrectly rejected.
- **Operational Risk:** Legitimate transfers and mints can fail unpredictably depending on issuer ordering, leading to user support load and degraded UX.
- **Denial-of-Service Vector:** An unavailable or misconfigured single issuer can block all verifications for a topic even when alternatives exist.
- **Inconsistent Business Semantics:** The system behaves as "first-issuer must have the claim" rather than "any trusted issuer may satisfy the topic," undermining policy expectations.

Consider handling missing or unavailable claims from an issuer as non-matches and continue evaluating all trusted issuers for the topic. Only declare the topic failed after every issuer has been checked and none provided a valid claim.

Update: *Resolved in pull request #386.*

H-05 Risk of Signature Reuse From Ambiguous `claim_data`

The claim validation signs a digest of `identity` (Address), `claim_topic` (u32), and `claim_data` (Bytes). While `identity` and `claim_topic` provide partial domain separation, `claim_data` is an unstructured byte array with no defined encoding. This lack of specification creates ambiguity: a signature may be reused across different claim issuers. For instance, the reference ClaimIssuer includes no domain separation, enabling replay across issuers.

Consider adding a domain separator (e.g., the verifying contract address). Furthermore, consider defining a serialization schema for `claim_data` per `claim_topic` to ensure consistent interpretation.

Update: *Resolved in pull request #415. The development team stated:*

> *The message to sign includes the `network_id`, `claim_issuer`, and `nonce` values.*

Medium Severity

M-01 Duplicate-Entry Handling Allows for Mapping Divergence and Can Cause Administrative Issues

The RWA design intentionally merges the Claim Topics and Trusted Issuers registries to reduce cross-contract hops and to keep the topic↔issuer mapping consistent. This combined registry is queried during identity verification, which iterates claim topics and their corresponding issuers to validate claims before transfers, mints, recoveries, and burns. However, the current implementation permits duplicate entries on both sides of the mapping, undermining those guarantees and setting up the failure modes described below.

- **Duplicate Topics Per Issuer:** The registry accepts and stores repeated instances of the same claim topic for a single issuer (`add_trusted_issuer` and `update_issuer_claim_topics` store `claim_topics` exactly as provided), and `remove_claim_topic` only removes the first occurrence when a topic is deleted. This leaves stale topic entries attached to issuers after the topic has been globally removed.

- **Duplicate Issuers Per Topic:** The reverse mapping allows an issuer to be inserted multiple times under the same topic. Counts, lookups, and subsequent maintenance operations then operate on inflated, inconsistent lists.

The two behaviors reinforce each other and can push the registry into an inconsistent state that is hard to recover from. After removing a claim topic globally, any issuer that still holds duplicated references to that topic will cause both `remove_trusted_issuer` and `update_issuer_claim_topics` to touch a non-existent reverse mapping entry (the functions call `get_claim_topic_issuers` during cleanup, which panics with `ClaimTopicDoesNotExist`), reverting the transaction and effectively blocking administrative actions such as issuer updates or removals until the topic is recreated. Beyond revert risk and wasted compute, the duplicates distort off-chain assumptions: identity checks that iterate issuers per topic may process the same issuer more than once and event streams may misrepresent the effective policy footprint.

Consider normalizing inputs and maintaining set semantics in both directions. Specifically, deduplicate topic lists before persisting, enforce uniqueness when appending issuers under a topic, and remove all residual occurrences during cleanup. Add pre-flight validation which ensures that every referenced topic exists before any state is written, and make reverse-mapping updates tolerant of missing entries to avoid cascading panics during administrative workflows.

Update: *Resolved in pull request #410.*

M-02 No TTL Renewal for Document State

In the RWA core implementation, persistent keys essential for policy enforcement are regularly renewed within the getter methods to prevent expiration, ensuring that users can rely on critical state not silently disappearing. This is normal behavior elsewhere in the codebase and is aligned with the product's reliability posture. However, the document module does not renew TTL on reads or writes, nor does it maintain TTL on the document list, so documents are at risk of expiring silently. This divergence is surprising because the rest of the library manages TTL proactively, so integrators will reasonably assume the same safety net exists for documents, when in reality, it does not.

If document entries or the index list expire, contracts and off-chain consumers can observe missing documents without corresponding removal events, undermining availability and audit expectations. In a compliance context, silent loss of a governing document or its pointer can break investor disclosures and create reconciliation toil after ledger TTL windows elapse.

Consider mirroring the established TTL-management pattern: on every read, extend the TTL of both the per-document key and the document-list key using conservative thresholds.

Update: *Resolved in pull request #422.*

M-03 On-Chain Storage of Plaintext Personally Sensitive Information

The `identity_registry_storage` contract stores sensitive compliance data, including personally sensitive and maybe identifiable information, for user accounts. A public function, `get_country_data_entries`, provides unrestricted access to this information. The public visibility of this function exposes sensitive user data to all actors on the blockchain. Storing personal information in plaintext on an immutable ledger creates severe privacy and security risks, including data harvesting for malicious attacks such as fraud and identity theft and non-compliance with potential data protection regulations.

It is recommended to adopt a privacy-preserving model where only cryptographic commitments are stored on-chain. Raw sensitive data should be held off-chain and verified on-chain using proofs when required. As such, consider implementing the following recommendations:

- Store a hash or Merkle root of the compliance data on the contract, not the plaintext data itself.
- To verify an attribute, require users to submit a cryptographic proof that validates against the on-chain commitment.

This approach enables selective disclosure and mitigates the fundamental risks of storing sensitive information on a public ledger.

Update: *Partially resolved in pull request #431.*

M-04 Potentially Inconsistent `claim_topic` State

The `claim_issuer` module, which manages signing keys, does not communicate with the `claim_topics_and_issuers` module, which serves as the central registry for what topics are valid as well as the registered issuers. Specifically, the allow_key function lets an administrator authorize a signing key for any `u32` value as a `claim_topic`. It never checks if this topic is actually a valid, registered topic in the `ClaimTopicsAndIssuers` contract,

creating the potential for a split-view. Allowing authorization of invalid claim topics can create state inconsistencies and unpredictable behavior across components in the future.

Consider enforcing consistency across the system by validating that a `claim_topic` exists in the central registry and that this issuer is one of the registered claim topic issuers before authorizing a key for it. Conversely, when a topic is no longer supported in the registry, there should be a way to know. One way to achieve this is via the `is_key_allowed` view function where the registry can be queried regarding the validity of a `claim_topic`. This function should be used with the actual verification logic in is_claim_valid as well.

Update: *Resolved in pull request #428. The development team stated:*

> *We have refactored the claim issuer key management mechanism to use a `SigningKey { public_key, scheme }` with bidirectional storage (`Topics(claim_topic)` -> `Vec<SigningKey>` and `Registries(SigningKey)` -> `Vec<Address>`).*

M-05 Divergence Between Policy and Smart Account State

The `simple_threshold` policy enforces a multi-signer requirement by validating that the number of authenticated signers meets a specified threshold. This threshold is set relative to the total number of signers in a `ContextRule` at the moment of the policy's `installation` or last modification. The design separates the policy logic from the core smart account logic.

A vulnerability arises because the policy is not notified when the list of signers in the parent smart account's `ContextRule` is modified. This creates a divergence between the policy's understanding of the signer set and the actual state of the smart account. This divergence leads to two distinct side effects:

- **Denial of Service**: If signers are removed from a rule, the total number of signers may fall below the policy's stored threshold. This makes it impossible to meet the signature requirement, permanently blocking any actions governed by the policy.
- **Unintentional Security Degradation**: If M signers are added to a rule that was configured to be a strict N-of-N multisig, it silently becomes an N-of-(N+M) multisig. This weakens the security guarantee without any explicit warning, creating a false sense of security for the account administrators.

Consider adopting remediation strategies based on the desired trade-off between security and architectural flexibility. At a minimum, consider adding explicit documentation that warns

administrators to update the policy thresholds before modifying a `ContextRule`. For a more robust solution that retains modularity, consider implementing a notification mechanism going from the smart account to its policies upon signer changes. To eliminate this class of issue entirely, consider handling all signature validation logic (including thresholding) directly within the smart account, which would prevent state divergence by design at the cost of architectural flexibility.

Update: *Partially Resolved in pull request #412.*

M-06 Signature Scheme Is Not Propagated to Issuer Contract

The `identity_claims` contract is responsible for managing claims associated with an identity. When adding a claim via the `add_claim` function, it accepts a `scheme` parameter, which is an identifier for the cryptographic signature scheme used. The contract then performs a cross-contract call to an external issuer contract to validate the claim by invoking the is_claim_valid function.

The `scheme` parameter received by the `add_claim` function is not passed along in the `is_claim_valid` cross-contract call. The `ClaimIssuer` trait, which defines the interface for the issuer contract, does not include the scheme in its `is_claim_valid` function signature. This omission makes it impossible for the issuer contract to know which signature verification algorithm to apply, breaking the system's ability to support multiple signature schemes and rendering the claim validation process unreliable.

Consider either modifying the `ClaimIssuer` trait to include the scheme parameter in the `is_claim_valid` function or documenting clearly the assumption that each claim issuer can select their own signature scheme. In the latter case, one should also remove the `scheme` field from the `Claim` struct since it may result in conflicting states (i.e., the scheme does not match the signature) which may have downstream effect in the future.

Update: *Resolved in pull request #414.*

M-07 Unbounded `DocumentList` Iteration

The `remove_document` function iterates over the entire `DocumentList` to construct a new list excluding the removed element. The computation cost of this operation scales linearly (O(n)) with the total number of documents. If an administrator adds a large number of

documents, the resource cost to execute this loop in one transaction can exceed the Soroban per-transaction resource limit.

In particular, the individual ledger entry size is 128 KiB which caps the number of documents allowed in the `DocumentList` ledger entry to approximately 4096. Consequently, once the list reaches this maximum, `set_document` and `remove_document` can fail, leading to a permanent DoS for these functions. The same unbounded iteration is also observed in the `get_all_documents` function.

Consider refactoring the `remove_document` algorithm by using a constant-time logic, such as a "swap and pop" pattern, to avoid iterating over unbounded lists and setting a limit for the maximum number of documents allowed.

Update: *Resolved in pull request #421.*

M-08 Unlimited and Duplicate Rules in Smart Accounts

The smart account authorization model uses `__check_auth`, which processes a list of `ContextRule` objects. For each context, it retrieves both context-specific and default rules, then evaluates them in last-in, first-out order. The first satisfied rule grants authorization. The `add_context_rule` function places **no limit** on the number of rules per `ContextRuleType`. This allows arbitrarily large rule lists, including for the Default context.

Each authorization check must process this entire list, meaning a sufficiently large set can exhaust computation resources, causing transactions to fail and creating a permanent DoS situation that freezes the account. In addition, each new rule created via `add_context_rule` is assigned a unique auto-incrementing ID, but the system does **not check for duplicates**. This enables multiple rules with identical contents but different IDs, increasing complexity and potential mis-configurations.

Consider enforcing a hard cap on rules per `ContextRuleType` and adding checks to prevent duplicates, ensuring predictable and bounded authorization logic.

Update: *Resolved in pull request #427.*

M-09 Unbounded Spending History Iteration

The `SpendingLimitData` struct stores a `spending_history: Vec<SpendingEntry>` vector to track all transactions within a given time window. However, this vector is not capped in size, leading to two DoS vulnerabilities:

- **Performance DoS**: The `calculate_total_spent_in_period` function is called on every transaction governed by the policy. This function iterates over the entire `spending_history` vector to sum the amounts, both at `can_enforce` and `enforce`, in one transaction, resulting in an O(N) complexity where N is the number of spending entries in the 1-day window. As an account's transaction history grows, the computational resources required for this iteration can exceed Soroban's per-transaction instruction limit. When this occurs, the transaction will fail, effectively blocking the user from making new transfers.

- **Storage DoS**: The entire `SpendingLimitData` struct, including the `spending_history` vector, is stored as a single entry in persistent storage. This single ledger entry is subject to a 128 KiB size limit. Each `SpendingEntry` item has a raw size of 20 bytes (`i128` for amount is 16 bytes, `u32` for `ledger_sequence` is 4 bytes). Thus, the maximum number of spending entries is approximately: `131,072 bytes (128 KiB) / 20 bytes/entry ≈ 6,553 entries`. Practically, it is possible for this maximum to be reached by an automated trading account in a day. Once the number of entries approaches this limit, the `e.storage().persistent().set()` call within the `enforce` function will fail due to the entry size being too large, causing a permanent DoS.

Consider caching the total spent amount in a new storage field and only updating the cached total as entries are added or expired. Consider also putting a hard cap on the history vector's length to prevent the storage DoS vulnerability.

Update: *Resolved in pull request #408.*

M-10 `FungibleVault` Deviates From ERC-4626 Specification Requirements

The `FungibleVault` aims to comply with the ERC-4626 tokenized vault standard but currently violates multiple "MUST" requirements outlined in the specification.

- The `asset()` function (currently implemented as `query_asset()`) reverts if the underlying asset in the instance storage is not set. This violates the ERC-4626

requirement that `asset()` must not revert. Since `total_assets()` depends on `query_asset()`, it inherits this revert behavior, leading to further non-compliance. While this behavior is due to Soroban's storage model, it should be documented clearly. In compliant implementations, the vault must ensure a default or error-resilient return for `asset()`.

- According to the specification, if there is no effective limit on deposits or mints, these functions must return the maximum representable value (`2**256 - 1`). In Soroban's numerical domain, the appropriate equivalent is `i128::MAX`, but the current implementation uses `i64::MAX`, which is significantly lower and restrictive. This limits composability and breaks compatibility expectations.

- ERC-4626 requires support for the standard ERC-20 `approve` / `transferFrom` allowance-based flow. However, the current `deposit_internal` implementation uses `token.transfer` with direct authentication (`from=operator`), bypassing the allowance mechanism. This makes the vault incompatible with existing tooling and patterns relying on delegated transfers. To comply, an `approve` / `transfer_from`-based path should be added alongside the auth-based flow.

Ensuring compliance with the aforementioned MUST requirements of the specification essential for interoperability, tooling support, and behavioral predictability across all ERC-4626-compatible systems. Where deviations are necessary due to Soroban-specific constraints, they should be clearly documented to avoid implementation confusion or integration errors.

Update: *Resolved in pull request #423.*

M-11 Non-Standard Message Pre-Hashing for Ed25519

The `Ed25519Verifier` implementation pre-hashes the claim message using `keccak256`. However, the standard `Ed25519` signature algorithm (as defined in RFC 8032 and used in most cryptographic libraries) uses the EdDSA signature scheme with fixed internal hashing. The `ed25519_verify` function expects the full, raw message as input, and it performs its own internal hashing (using SHA-512).

By pre-hashing with keccak256, the signature scheme deviates from the standard primitive. As a result, the non-standard signing process becomes incompatible with standard off-chain tools or wallets: a standard `Ed25519` library signs the raw message (`sign(message)`), while the contract expects a signature over the hashed message (`sign(keccak256(message))`), causing the signatures to mismatch, thus fails the on-chain verification.

For improved security, interoperability and better developer experience, consider adhering to the Ed25519 standard by passing the full, unhashed message to the verification function.

Update: *Resolved in pull request #415.*

M-12 State Desynchronization Risk in Identity Registry

The `identity_registry_storage` contract is responsible for linking user accounts to their corresponding on-chain identity contracts. In addition to managing the primary `account -> identity` mapping, it also stores detailed `IdentityProfile` that includes `CountryData` information, which is keyed directly by the user's account address. The contract includes a `modify_identity` function intended to update an account's associated identity contract.

When the `modify_identity` function is executed, it only updates the pointer from the account to the new identity contract address. It fails to update or remove the `IdentityProfile` and `CountryData` that are also stored under the same account key. This results in a state desynchronization, where the registry holds stale profile data belonging to the previous identity while pointing to a new one. Consequently, consumers of the registry can receive contradictory data, leading to incorrect behavior and potential security risks.

Consider refactoring the architecture to enforce a single source of truth and eliminate data duplication. The `identity_registry_storage` contract should be simplified to serve exclusively as a registry, managing only the `account -> identity` mapping. All data intrinsic to an identity, including the `IdentityProfile` and its associated `CountryData`, should be moved into the respective identity contract itself. This change would ensure that an identity's data is always consistent and encapsulated, removing the risk of state desynchronization.

Update: *Acknowledged, not resolved. The development team stated:*

> *The finding incorrectly assumes that profile data is tied to the identity contract, whereas it is related to the account:*
>
> - *account -> identity*
> - *account -> profile data*
>
> *So, there is no risk of state desynchronization.*

M-13 Non-Standard Hashing Algorithm for ECDSA `Secp256r1` Signature Scheme

The `Secp256r1Verifier` implementation uses `keccak256` to hash the claim message before signature verification. Standard practice pairs `Secp256r1` (aka `P-256`) with `SHA-256` in ES256 signing algorithm as registered in the IANA COSE Algorithms registry. In particular, the webAuthn specification also uses ES256. Thus, the current implementation deviates from established cryptographic standards. As a result, standard off-chain tools cannot produce valid signatures, breaking interoperability and forcing signers to rely on building custom signing process (`sign(keccak256(message))`) instead of the standard (`sign(sha256(message))`).

Consider aligning with the established cryptographic standard and updating the hashing algorithm from `keccak256` to `sha256` for both `Secp256r1Verifier` instances.

Update: *Resolved in pull request #415.*

M-14 Vault Artificially Caps `deposit/mint` for High-Decimal or Low-Unit-Value Assets

`max_deposit()` and `max_mint()` return `i64::MAX as i128`. Since `deposit()`/`mint()` restrict calls using these values, and `assets`/`shares` are expressed in base units, the effective cap becomes **`i64::MAX` base units**.

For an asset with `d` decimals, the maximum whole-token deposit is:

$\frac{2^{63}-1}{10^d}$

- With **18 decimals** (like the tokens in the `examples` folder), that is $\approx 9.223372036854776$ tokens, any deposit/mint > 9.223372036854775807 reverts with `VaultExceededMaxDeposit` (or the mint equivalent).
- Even with fewer decimals, **low-unit-value assets** (e.g., memecoins) can still hit the cap in USD terms because they require huge token counts. **Example (9 decimals):** max whole tokens $\approx 9,223,372,036.854775807$. If price = $1e-8/token$, the max deposit is only 92.

This is an artificial DoS for common 18-decimal assets and many low-price tokens. It also contradicts the trait docs ("currently `i128::MAX`" in both `max_deposit` and `max_mint`) and risks breaking integrators that infer "unbounded" behavior.

Consider returning `i128::MAX` from both `max_deposit()` and `max_mint()`.

Update: *Resolved in pull request #381.*

M-15 Recovery Clears Freeze Accounts and Unfreezes Tokens

`recovery_address` exists to restore custody when an investor loses a wallet: after identity verification (and RBAC checks on the operator), it force-transfers the position from the lost wallet to a new wallet controlled by the same person. Currently, the function zeroes out the source wallet's frozen bucket, clears the source address's frozen state, transfers the full balance to the destination, and omits the compliance transfer hook. This turns frozen or partially frozen positions into fully liquid balances on the destination wallet and erases a source-level freeze that may be tied to identity-level sanctions.

This behavior enables a compliance-policy bypass: a frozen or sanctioned position can be laundered via recovery instead of an explicit unfreeze, and downstream systems miss the movement because the `transferred` compliance hook is not called. Even with identity verification and privileged execution, the function modifies the compliance posture instead of merely transferring control, which undermines auditability and regulatory guarantees.

Consider modifying the recovery procedure so that it preserves the compliance state while moving control. Carry over the frozen amount to the destination wallet's frozen bucket instead of emitting an "unfrozen" event for the source. If the source address is frozen, mark the destination as frozen, and call the compliance `transferred` hook while emitting transfer and recovery events. Any unfreeze operation should remain a separate, explicit, and logged step.

Update: *Resolved in pull request #380.*

Low Severity

L-01 `burn()` Can Temporarily Set Negative Frozen Balance Before Failing

`burn()` conditionally "unfreezes" tokens before burning. If `amount > free_tokens`, it computes `tokens_to_unfreeze = amount - free_tokens` and subtracts it from `current_frozen`. Since the type of `amount` and the stored values is `i128`, this

subtraction does not underflow. Instead, it simply yields a negative `new_frozen`. The function then writes this negative value to storage and emits an "unfrozen" event before calling the underlying balance update. If the subsequent balance update rejects the burn (e.g., insufficient total balance), the runtime will revert atomically and roll back the temporary write and event so that no durable state corruption occurs.

However, even with atomic rollback, the pattern is still undesirable. It momentarily violates the invariant that "frozen ≤ balance" and that the frozen amount is non-negative, and it relies on a later failure to unwind that violation. This increases cognitive load for auditors, complicates reasoning about invariants, and risks confusing telemetry/log consumers because emitted events that accompany the temporary write are also rolled back.

Consider validating the total balance before any state mutation and failing with a clear domain error when the burn cannot be covered, mirroring the upfront balance check used in `forced_transfer`.

Update: *Resolved in pull request #405.*

L-02 Events Omit Operator Identity

Document events are a primary source for downstream indexing, monitoring, and forensic review in RWA systems. Therefore, they should capture not only what changed but also who enacted the change, so that role-based workflows and accountability requirements are satisfied. Currently, the document update and removal events include the operator's name (and for updates, the URI, hash, and timestamp) but omit the operator's address. The impact is a weaker audit trail: it is harder to attribute actions to specific operators, correlate changes with RBAC logs, or demonstrate proper separation of duties to auditors.

Consider including the operator address as a topic in all document-related events to ensure that the emitting path passes through the authenticated caller end-to-end. Doing so improves traceability, enables straightforward role/actor filtering by indexers, and aligns document operations with the accountability standards expected of regulated assets.

Update: *Acknowledged, not resolved. The development team stated:*

> *Document updates are expected to be performed by admins only, so those changes can be easily tracked. Furthermore, if we decide to follow the recommendation, we should apply it to all state-changing admin functions to keep consistency across all the rwa modules. As such, we are fine to leave this as it is.*

L-03 Unbounded Decimals Offset Can Overflow and Brick Conversions/Previews

`convert_to_assets_with_rounding` and `convert_to_shares_with_rounding` compute `pow = 10_i128.checked_pow(offset)`. Since `i128::MAX ≈ 1.7014e38`, `10^38` fits but `10^39` overflows. As such, if `offset > 38`, `checked_pow` fails and both conversions revert for all inputs. This also cascades to any preview paths that call these helpers, conflicting with EIP-4626 guidance which requires that previews should only revert on unreasonable (huge) input, not due to configuration. `decimals()` also exposes an unbounded virtual precision (`underlying + offset`) which can be inconsistent with the chosen numeric domain if not documented.

To prevent a configuration-induced DoS, consider bounding the offset at definition time and enforcing it in the setter.

Update: *Resolved in pull request #420.*

L-04 Opaque Numeric Claim Topics and Signature Scheme Identifiers (`u32`)

`claim_topic` is encoded as a `u32` value in an identity claim. The signature `scheme` is likewise represented as a `u32` value. Representing these meaningful topics/schemes as opaque integers creates "magic numbers" whose meanings only live off-chain without explicit consensus. This increases the risk of stale/mismatched mappings, misconfiguration, incorrect claim interpretation, failed signature validation, and higher audit friction.

For better clarity, maintainability, and to reduce the risk of human error, consider making accepted `claim_topic` and signature `scheme` more explicit (e.g., by using symbol type or Enum) or clearly documenting the reference for what each value represents.

Update: *Acknowledged, not resolved. The development team stated:*

> *We prefer to keep it consistent with the reference Solidity implementation and also to give enough flexibility so that those numbers can be interpreted independently for certain specific use cases.*

L-05 Incorrect Slice-Length Validation

The `extract_from_bytes` utility function of the `stellar-accounts/verifiers` module is responsible for extracting a fixed-size slice from a larger data buffer. The validation logic does not correctly enforce that the resulting slice's length is exactly `N`. The check only prevents the slice from being longer than `N` but permits it to be shorter. If a caller specifies a range where the length is less than `N`, the validation passes, but the subsequent attempt to convert the slice into a fixed-size buffer will panic when `copy_from_slice`. This is because in `soroban-sdk`, the `as_slice()` function returns a slice of length (`self.len`) despite the capacity of the buffer being `N`.

Consider strengthening the validation logic to prevent range underflow and ensure that the slice length is strictly equal to `N`. As such, the check should be updated to `end > data.len() || end - start != N as u32` to prevent the panic.

Update: *Resolved in pull request #407.*

L-06 Deviation from Native Stellar Signature Validation

Native Stellar transactions enforce a strict validation model whereby a transaction is rejected if it includes more signatures than the minimum necessary to meet the required threshold. The smart account contract, however, implements a different authentication pattern where it approves a transaction as soon as the number of valid signatures equals the specific rule context signer count or the attached policy's threshold, without validating the total number of signatures provided. This behavior is a deviation from the security principles of the underlying Stellar protocol. It permits an actor to attach superfluous, valid signatures to a transaction before submission. This affects the authentication loop that iterates all supplied signatures.

If the current, more permissive behavior is an intentional design choice, consider explicitly documenting this deviation. Due to the current design of off-loading part of the authentication check to attached policies, it could add too much complexity to reject superfluous signatures. However, implementing a stricter check is most straightforward if the smart account manages signer and threshold information directly, rather than outsourcing important details to external contracts.

Update: *Acknowledged, not resolved. The development team stated:*

> *We agree with the stated principal. However, we do not envision a situation where someone has an interest in doing this. Superfluous signatures will mean a higher tx cost*

> *which does not matter for the network as long as the used resources are paid. Also, the one who submits the tx will not have an interest in submitting a "heavier" tx because they will pay a higher cost.*

L-07 Missing Document Version History

The RWA design proposal frames document management as an auditable extension that should preserve evidence over time, including the ability to retain a history of document hashes. This expectation is called out explicitly in the “Document management” section of the proposal, which highlights auditability and batch workflows as first-order requirements.

In the current implementation, updating a document overwrites the single stored record for that name, discarding previous URIs, hashes, and timestamps. The impact is an auditability gap: you cannot prove which exact version of a term sheet, prospectus, or addendum was in force at a given time, and you cannot reconstruct a change timeline from on-chain state alone. For regulated assets, this weakens evidentiary guarantees and complicates incident response and compliance reviews.

Consider maintaining an append-only version log alongside the canonical “latest” record. Each update should append a new version entry containing the prior URI, hash, timestamp, and the actor who performed the change, while the “latest” pointer is replaced. A read path can return either the latest or the full history, and events should mirror this by emitting a versioned record for downstream indexing. Since there are some resource limits on the vector size, consider using a ring buffer structure.

Update: *Acknowledged, not resolved. The development team stated:*

> *Thanks to the emitted events, one can track down and reconstruct document's version history. So, it seemed redundant to devise such an on-chain system for this extension.*

Conclusion

The audit of the Stellar Contracts Library highlights an expanded functionality and complexity, especially with the introduction of the Real World Asset (RWA) and smart account frameworks. While these modules provide powerful capabilities, systemic issues raise concerns about their readiness for production. The most urgent areas of focus are the complexity of the RWA module, the security of the smart account design, and the overall developer experience.

The RWA framework is not yet mature, with state desynchronization across identity, claims, and compliance components, as well as critical flaws in wallet recovery that could result in fund drainage, token inflation, or policy bypass. Its intricacy currently outpaces its maturity. Comprehensive integrated testing covering full asset and identity life cycles, failure cases, and recovery scenarios is essential to establish reliability.

The smart account framework prioritizes flexibility over security by delegating part of signature validation and core protections to pluggable external policies. This design makes it possible to bypass multisig safeguards or create insecure rules. The architecture could be reoriented to enforce immutable security guarantees within the main account contract, ensuring that signer validation and thresholds cannot be circumvented.

Finally, given the multitude of different components provided by the library, it would be of great benefit to provide real-world, ready-to-use, and secure examples for common scenarios to help developers adopt the library safely and reduce vulnerabilities in the ecosystem built on it.