OpenZeppelin | security

Stellar Contracts Library v0.5.0 Re-Audit

October 23, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-10-07 To 2025-10-14
Languages	Rust (Soroban)

Total Issues	12 (11 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	2 (2 resolved)
Medium Severity Issues	2 (1 resolved)
Low Severity Issues	6 (6 resolved)
Notes & Additional Information	2 (2 resolved)

Scope

OpenZeppelin diff-audited the OpenZeppelin/stellar-contracts repository with HEAD commit 7b14668 against BASE commit 028c429.

In scope were the following files:

```
packages/
├── accounts/
│   └── src/
│       ├── policies/
│       │   ├── simple_threshold.rs
│       │   ├── spending_limit.rs
│       │   └── weighted_threshold.rs
│       ├── smart_account/
│       │   ├── mod.rs
│       │   └── storage.rs
│       └── verifiers/
│           ├── ed25519.rs
│           ├── mod.rs
│           ├── utils/
│           │   └── extract_from_bytes.rs
│           └── webauthn.rs
└── tokens/
    └── src/
        ├── fungible/
        │   ├── extensions/
        │   │   └── vault/
        │   │       ├── mod.rs
        │   │       └── storage.rs
        │   └── mod.rs
        └── rwa/
            ├── claim_issuer/
            │   ├── mod.rs
            │   └── storage.rs
            ├── claim_topics_and_issuers/
            │   └── storage.rs
            ├── compliance/
            │   ├── mod.rs
            │   └── storage.rs
            ├── extensions/
            │   └── doc_manager/
            │       ├── mod.rs
            │       └── storage.rs
            ├── identity_claims/
            │   └── storage.rs
            ├── identity_registry_storage/
            │   ├── mod.rs
            │   └── storage.rs
```

```
        ├── identity_verifier/
        │   ├── mod.rs
        │   └── storage.rs
        ├── utils/
        │   └── token_binder/
        │       ├── mod.rs
        │       └── storage.rs
        ├── mod.rs
        └── storage.rs
```

OpenZeppelin was engaged by the Stellar Contracts Team to perform an additional 1-week audit on account of the changes made to the codebase since the previous audit. This re-audit was necessitated by the significant re-factoring of the codebase, particularly with respect to the Smart Account and RWA modules. An overview of all the critical changes is given below.

Key Changes in the Smart Account Module

A significant refactoring was conducted that focused on improving the security, performance, and robustness of the smart account policies.

Security Hardening and Warnings

- **Signer Divergence Mitigation:** Extensive documentation and warnings have been added to the `simple_threshold` and `weighted_threshold` policies. This addresses a critical security concern where modifying an account's signers does not automatically update the policy's threshold. This could lead to a **denial of service** (if signers are removed) or **silent security degradation** (if signers are added). Administrators are now explicitly instructed to manually update policy thresholds when managing signers.
- **Context Rule Fingerprinting:** A new "fingerprint" mechanism has been implemented to prevent the creation of duplicate context rules that have the same set of signers and policies. This is enforced by a new `DuplicateContextRule` error.

Performance and Robustness

- **Optimized Spending Limit:** The `spending_limit` policy has been optimized by adding a `cached_total_spent` field. This avoids recalculating the total spent amount from the entire transaction history for each new transaction, thereby improving efficiency.
- **History Capacity Limit:** The `spending_limit` policy now enforces a `MAX_HISTORY_ENTRIES` limit to prevent the spending history from growing indefinitely, which protects against potential denial-of-service attacks. A new `HistoryCapacityExceeded` error is returned if the limit is reached.

- **Context Rule Limit:** The number of context rules per smart account is now capped at `MAX_CONTEXT_RULES` (15).

API and Code Refinements

- Several internal functions in the policies have been updated to use the `context_rule_id` instead of passing the entire `ContextRule` object, simplifying the API.
- The `spending_limit` policy now requires at least one authenticated signer to enforce a rule.
- The signature of the `authenticate` function has been updated for a more streamlined authentication flow.

Testing

- All tests have been updated to align with the API changes.
- New tests have been added to cover the new features and error conditions, including history capacity limits, rule limits, and fingerprinting logic.

Key Changes in the RWA Module

Signature Invalidation with Nonces

A new mechanism has been introduced to invalidate all existing claim signatures for a specific identity and topic by incrementing a `nonce`. The claim message format has been updated to include the `network_id`, `claim_issuer` address and the `nonce`, effectively preventing replay attacks across different networks and issuers, and after a nonce has been incremented.

Redesigned Key Management

The storage structure for key management has been completely redesigned into a more robust bidirectional mapping system:

- A `SigningKey` is now a combination of a `public_key` and a signature `scheme`.
- The system now tracks which `claim_topics_and_issuers` registries a signing key is associated with and which signing keys are authorized for a given claim topic.

Integration with the `claim_topics_and_issuers` Registry

The `allow_key` function now performs cross-contract calls to a `claim_topics_and_issuers` registry. It verifies that the claim issuer is registered and is authorized to issue claims for the specified topic before allowing a new signing key.

Improved Revocation Mechanism

The per-claim revocation mechanism (`set_claim_revoked`) is now nonce-independent. It uses a stable claim identifier, ensuring that a revocation persists even if the nonce for that identity/topic pair is incremented.

Flexible Signature Schemes

The concept of a signature `scheme` (a `u32`) has been formalized, making the `ClaimIssuer` trait more flexible and capable of supporting multiple, custom verification methods within a single contract.

API and Trait Refinements

- The `SignatureVerifier` trait has been simplified: `build_claim_digest` has been renamed to `build_message` and `verify_claim_digest` has been renamed to `verify`.
- Event payloads have been updated to emit more contextual information, such as the `registry` and `scheme`.
- New errors and constants have been added to support the new logic (e.g., `MaxKeysPerTopicExceeded` and `IssuerNotRegistered`).

Comprehensive Test Coverage

The entire test suite has been rewritten to cover the new architecture, including the bidirectional mappings, nonce invalidation, and integration with a new mock registry.

Some Outstanding Concerns

While most of the issues from the previous report have been fixed, there are still several recommendations that have been addressed as explicit warnings but not fully resolved:

- **RWA Privacy Concern on Personally Identifiable Data**: Despite the explicit and extensive warning in the `identity_registry_storage` contract that the current implementation stores sensitive user information in plaintext on the public blockchain,

allowing unrestricted access to anyone, the risk of data harvesting for malicious purposes like fraud and identity theft is still present. It is hoped that the future evolution of the RWA module will provide privacy by default using one of the private alternatives mentioned in the documentations.

- **Smart Account Balancing Flexibility and Security Robustness** : Despite the explicit and extensive warning on the risk of potential out-of-sync states on the smart account with respect to external policies, the security of the smart contract is up to its administrators manually making sure the system is configured securely. Outsourcing some of the core authentication and signer validation logic to external add-on policies provides flexibility at the expense of an error-proof robustness. It is hoped that the future evolution of the smart account will enforce critical security directly on all signer-related logics within the core component.

Trust Assumptions

The trust assumptions from the previous audit report still stand true:

- **Soroban SDK:** The underlying platform's cryptographic functions are trusted to be secure and behave predictably.
- **Developer Integration:** The developers are trusted to correctly configure and integrate the library's components, implementing necessary access controls and validation.
- **Administrator Conduct:** The administrators are trusted to operate the system's privileged functions correctly and to not act maliciously.
- **Claim Issuers:** The off-chain claim issuers are trusted to diligently manage the entire lifecycle of claims.

High Severity

H-01 ID Counter Exhaustion Leads to Permanent Denial of Service

The smart account assigns a unique, auto-incrementing ID to each new context rule from a finite pool that is limited by the `MAX_CONTEXT_RULES` constant to a maximum of 15. The `add_context_rule` function fetches the current counter value for the new ID and increments the counter for subsequent calls. This ever-increasing counter is the sole mechanism for generating new rule IDs. This ID is also used for matching rule context precedence.

The `remove_context_rule` function deletes all data associated with a rule but does not reclaim its ID for future use. Since the `NextId` counter is never decremented, it will eventually reach the `MAX_CONTEXT_RULES` limit at 15, causing all subsequent calls to `add_context_rule` to fail permanently, even though there may not be 15 rules in existence since some can be removed.

Consider decoupling rule precedence from the rule ID value and introducing a `creation_ledger` field to each rule's metadata, which would be used to sort rules and determine evaluation order. With precedence managed independently, an ID reclamation mechanism, such as a free-list, can be safely implemented. This would allow deleted IDs to be recycled, preventing the ID counter from being exhausted and ensuring the long-term functionality of the contract.

Update: *Resolved in pull request #452. The development team stated:*

> *Instead of keeping track of free-list and recycling IDs, we have implemented a counter that tracks the number of rules. This is more effective code-wise and keeps IDs unique to avoid potential confusion.*

H-02 Permanent Dangling Keys

The `claim_issuer::storage` module provides a `remove_key` function to disassociate a signing key from a registry and claim topic. The design relies on a core invariant: a key associated with a topic must also be associated with at least one registry. The allow_key

function allows registering the same key for different topics from different registries. However in this case, removing a key from one registry does not remove its corresponding topic, resulting in a situation of dangling keys, permanently.

This can be illustrated with the following scenario:

1. A key `K` is associated with two unique topic-registry pairs: `(T1, R1)` and `(T2, R2)`. The contract's state is:
   - `topics(T1)` contains `K`
   - `topics(T2)` contains `K`
   - `registries(K)` contains `{R1, R2}`
2. A call to `remove_key(K, R1, T1)` is made. The function removes `R1` from the key's registry set but does not remove `K` from the topic `T1`. The state becomes:
   - `topics(T1)` contains `K`
   - `topics(T2)` contains `K`
   - `registries(K)` contains `{R2}`
3. A second call, `remove_key(K, R2, T1)`, removes the last registry `R2`. The function then proceeds to delete the `registries(K)` mapping entirely and removes `K` from the provided topic, `T1`. However, the mapping for `T2` is never touched.
4. The final state ends up being inconsistent: `topics(T2)` still contains `K`, but `K` has no associated registries. This creates a permanent dangling key, since it cannot be removed by calling `remove_key(K, R ,T2)` for any registry `R` as the function will fail with a `KeyNotFound` error.

The aforementioned inconsistency means that the returned value of `is_key_allowed_for_topic` can be incorrect if it is used as suggested in the documentation for the unimplemented `is_claim_valid` function. This could result in a claim being validated from an invalid key.

Consider refactoring the key-removal logic by considering all its supported topics from its respective registries.

Update: *Resolved in pull request #461. The development team stated:*

> *Besides solving the issue of dangling keys after removals, we also reconsidered the one-key-for-one-topic-per-registry invariant. Now, one key can be allowed for multiple topics in the same registry.*

Medium Severity

M-01 Secp256k1 Signature Malleability

`Secp256k1Verifier` is responsible for validating signatures for claims using the `secp256k1` scheme. The `verify` function implements this check by using the `secp256k1_recover` host function to recover a public key from the signature and then comparing it to the provided public key. However, this "recover-then-compare" pattern is not a secure substitute for a dedicated verification function. It introduces a critical signature malleability vulnerability, as ECDSA allows for multiple valid signatures for the same message and key.

More specifically, a malleable signature, as a different, valid signature for the same message and key, can be constructed from a valid signature (e.g., flipping the `s` value to `s'`) and a corresponding `recovery_id`:

- **Original Signature:** `(r, s)` with `recovery_id_1`
- **Malleable Signature:** `(r, s')` with `recovery_id_2`

The current implementation will accept these malleable signatures, which can lead to exploits if any part of the system uses the signature hash as an identifier.

Consider replacing the "recover-then-compare" logic with a call to a dedicated `secp256k1_verify` function that cryptographically validates the signature and enforces a canonical format to mitigate malleability. If the underlying Soroban SDK does not provide such a function, consider removing the `Secp256k1Verifier` entirely until a secure primitive is available.

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

> *This "signature malleability" can only create an additional valid signature, and in order to create a fake signature that will pass the verify signature, the following conditions are required:*
>
> - *The message should be present.*
> - *A valid signature for that message should be present.*
>
> *So, by creating another valid signature for the same message, what does the attacker gain? If this were a transaction, maybe a replay attack? However, in our case, we think that it is completely safe. Since this issue does not expose any exploitation or attack*

> *scenario as of yet, we have decided to not change the code. Nonetheless, we acknowledge the importance of using the `verify` variant instead of `recover` variant when it becomes available. As such, we will create an issue as a reminder and replace it with the `verify` host function variant when it becomes available.*

M-02 Claims Lack a Built-in Expiration Mechanism

The `claim_issuer` module facilitates the validation of cryptographic claims. The signed message payload, constructed in the `build_claim_message` function, includes parameters such as a nonce and the claim data, but omits a timestamp or expiration date. Consequently, a signed claim is perpetually valid unless it is actively invalidated by incrementing its corresponding nonce or adding it to the revocation list.

The current signature invalidation design necessitates active, on-chain lifecycle management for all claims, which can introduce operational overhead and increase costs for the issuer. It also creates a risk that old, unrevoked claims could be accepted long after their intended period of relevance has passed, potentially leading to unintended access or behavior when consuming contracts that do not implement their own separate deadline enforcement.

Consider enhancing the claim validation process to include a mechanism for passive invalidation. This could be achieved by adding an `expiration_timestamp` field to the data signed within the `build_claim_message` function. The corresponding validation logic, such as the `is_claim_valid` function or the verifier implementations, would then also check that the current ledger timestamp is not past the claim's expiration time. This would ensure that claims expire automatically and align with the principle of secure-by-default design.

Update: *Resolved in pull request #456. The development team stated:*

> *The creation and expiration timestamps can be optionally encoded in `claim_data`*

Low Severity

L-01 Misleading Return Value in Signature Verification Functions

The `claim_issuer` module provides verifier implementations for different signature schemes. The `Ed25519Verifier` and `Secp256r1Verifier` implementations contain a `verify` function responsible for validating cryptographic signatures.

The `verify` functions in both `Ed25519Verifier` and `Secp256r1Verifier` unconditionally return `true`. However, the underlying Soroban SDK cryptographic functions (`ed25519_verify` and `secp256r1_verify`) do not return a boolean value. Instead, they panic if signature validation fails. This creates a misleading function signature, as the `verify` method will never return `false`, and any invalid signature will result in a transaction-level panic rather than a controlled failure.

Consider refactoring the `verify` function in the `SignatureVerifier` trait and its implementations to not return a boolean value. This change would more accurately reflect the behavior of the underlying cryptographic primitives, where a failed verification results in a panic. The function's documentation should also be updated to clarify this behavior.

Update: *Resolved in pull request #466. The development team stated:*

> *Signature verification functions now panic.*

L-02 Unused Generic Parameter in `SignatureVerifier` Trait

The `claim_issuer` module defines a `SignatureVerifier<const N: usize>` trait to abstract different cryptographic signature schemes. This trait includes a constant generic parameter `N`, which is intended to specify the byte-length of the message digest used by the verification algorithm. All implementations hardcode a value of `32` and subsequently use hashing functions like sha256 or keccak256 that produce 32-byte digests.

Consider refactoring the `SignatureVerifier` trait and its implementations to remove the unused `N: usize` constant parameter.

Update: *Resolved in pull request #450.*

L-03 Lost Caller Attribution in `recover_balance`

In the `recover_balance` function, if the old account were frozen, the code reapplies the freeze to the new account by calling `Self::set_address_frozen(e, &e.current_contract_address(), new_account, true)`. This causes the `AddressFrozen` event to list the RWA contract itself as the caller, not the caller who may be an admin/operator who initiated the recovery. This loss of attribution weakens the audit trail, making it difficult to determine who was responsible for the action.

Consider using the invoker's address as the second argument for the `Self::set_address_frozen()` function.

Update: *Resolved in pull request #445. The development team stated:*

> *We believe that this finding is valid. However, as explained in our reply to the previous audit, we believe that events themselves are enough for the audit trail. In addition, we will not put the operator/caller name in events. If we decide to do so, we would put them in every event for consistency, which we think would be overkill. For the fix, and for consistency, we have removed the `operator` argument from that event as well.*

L-04 Unnecessary Allocations Lead to Increased Resource Consumption

Throughout the token binder's storage module, multiple functions handle cases where a storage key may not exist by providing a default empty `Vec<Address>`. This pattern is used when retrieving token buckets that have not yet been created.

The default value is provided using `unwrap_or(Vec::new(e))`. However, this implementation is eager, causing a new empty vector to be allocated via a host function call every time these lines are executed, even in the common case where the storage entry exists, and the default value is immediately discarded. This results in unnecessary object allocations and wasted CPU instructions, particularly in functions that loop over many buckets like `get_token_index` and `is_token_bound`, leading to higher transaction fees for users.

Consider replacing all instances of `unwrap_or(Vec::new(e))` with the lazy equivalent, `unwrap_or_else(|| Vec::new(e))`. This change ensures that the default empty vector is only allocated when it is strictly required, conserving CPU resources and reducing transaction costs.

Update: *Resolved in pull request #454.*

L-05 Inefficient Check for Already Bound Tokens

The `bind_tokens` function facilitates binding multiple tokens in a single transaction. As part of its validation, the function verifies that none of the tokens in the provided batch have already been bound to the contract.

This verification is implemented by first loading the entire list of already-bound tokens from storage into memory, and then iterating over the input batch to perform a linear scan for each token. However, this approach leads to excessive memory allocation with the worst case of loading 10,000 addresses in memory. A cleaner approach can be achieved by invoking the `is_token_bound` function, which reads from storage one bucket at a time (with 100 addresses), instead of the entire list. In addition, the `require_auth_from_bound_token` function can also use `is_token_bound()` instead of `linked_tokens()` to be more efficient.

Consider refactoring the both of the aforementioned instances to reduce unnecessary memory consumption.

Update: *Resolved in pull request #462.*

L-06 Inefficient Duplicate Check

The bind_tokens function is used to bind up to 200 token addresses to the contract in a single batch operation. To prevent the same token from being bound twice in one transaction, the function checks for duplicates within the input vector.

This check is implemented with an algorithm of `O(n^2)` complexity with `slice` and `contains` method. The worst case scenario is when 200 unique token addresses are added in a batch, leading to heavy resource consumption in this step alone. This can be improved by using a sort-and-check-for-adjacent-duplicates approach. The `slice.sort()` algorithm implements this with a complexity of `O(nlogn)`, which reduces the entire duplicate check complexity to `O(nlogn)`. In addition, checking duplicate input topics can also benefit from such a refactoring.

Consider refactoring the duplicate check algorithm for improved efficiency.

Update: *Resolved in pull request #458.*

Notes & Additional Information

N-01 Redundant Check in `is_key_authorized`

The `is_key_authorized` function checks that the registry client has the current contract address as a trusted issuer and that it also has the provided claim topic. However, the first check is redundant since `has_claim_topic` checks the topic mapping keyed by the trusted issuer, which does not exist if the issuer is not trusted (e.g., when it is removed).

Consider only using the `has_claim_topic` check to reduce cross-contract calls in `is_key_authorized`.

Update: *Resolved in pull request #447.*

N-02 Misleading Comments

Throughout the codebase, multiple instances of misleading comments were identified:

- The comment stating that `can_transfer()` is the only function expected from the compliance client is misleading, since other functions such as `transferred()`, `can_create()`, `created()`, and `destroyed` are also needed.
- The `onchain_investor_id` parameter has been removed as an argument to the function.
- This instance of "assets" should be "shares" and this instance should also be "shares".
- Here, `context_rule` should be `context_rule_id`.

Consider updating the aforementioned comments to correctly reflect the changes made to the codebase.

Update: *Resolved in pull request #443.*

Conclusion

This differential audit of the OpenZeppelin Stellar Contracts library reveals a mature and sophisticated codebase, reflecting the development team's deep expertise in smart contract engineering. The project successfully tackles complex challenges inherent in creating robust, reusable primitives for real-world assets (RWA) and advanced account structures. However, certain architectural design considerations related to smart accounts and privacy concerns in the RWA system, which were identified in an earlier audit, have only been partially addressed through comments instead of implemented fixes.

Overall, the codebase demonstrates a strong security posture with additional tests. The issues identified are generally nuanced and relate to complex, stateful interactions, while some touch on compute optimizations, code clarity, and minor efficiency gains. This pattern suggests that the core logic is sound, and the value of the audit has been in refining and hardening an already high-quality foundation.

To further enhance the library's practical applicability and security, it is recommended to develop comprehensive, ready-to-use, real-world examples that demonstrate secure integration patterns. Looking ahead, future iterations could significantly benefit from implementing a private RWA solution to address confidentiality concerns and a built-in threshold signature mechanism for smart accounts to improve key management and operational security.