OpenZeppelin | security

Stellar Contracts Library v0.3.0-rc.2 Audit

July 3, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-06-04 To 2025-06-18
Languages	Rust (Soroban)

Total Issues	37 (32 resolved, 2 partially resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	5 (4 resolved, 1 partially resolved)
Low Severity Issues	16 (16 resolved)
Notes & Additional Information	12 (8 resolved, 1 partially resolved)
Client Reported Issues	2 (2 resolved)

Scope

OpenZeppelin conducted a differential audit of the OpenZeppelin/stellar-contracts repository at commit cf05a5d against commit d3741c3.

In scope were the following files:

```
packages
├── access
│   ├── access-control
│   │   └── src
│   │       ├── access_control.rs
│   │       ├── lib.rs
│   │       └── storage.rs
│   ├── access-control-macros
│   │   └── src
│   │       └── lib.rs
│   ├── ownable
│   │   └── src
│   │       ├── lib.rs
│   │       ├── ownable.rs
│   │       └── storage.rs
│   ├── ownable-macro
│   │   └── src
│   │       └── lib.rs
│   └── role-transfer
│       └── src
│           ├── lib.rs
│           └── storage.rs
├── constants
│   └── src
│       └── lib.rs
├── contract-utils
│   ├── crypto
│   │   └── src
│   │       ├── hashable.rs
│   │       ├── hasher.rs
│   │       ├── keccak.rs
│   │       ├── lib.rs
│   │       ├── merkle.rs
│   │       └── sha256.rs
│   ├── default-impl-macro
│   │   └── src
│   │       ├── helper.rs
│   │       └── lib.rs
│   ├── macro-helpers
│   │   └── src
│   │       └── lib.rs
│   ├── merkle-distributor
```

```
│   │   └── src
│   │       ├── lib.rs
│   │       ├── merkle_distributor.rs
│   │       └── storage.rs
│   ├── pausable
│   │   └── src
│   │       └── pausable.rs
│   ├── pausable-macros
│   │   └── src
│   │       └── lib.rs
│   ├── upgradeable
│   │   └── src
│   │       ├── lib.rs
│   │       ├── storage.rs
│   │       └── upgradeable.rs
│   └── upgradeable-macros
│       └── src
│           └── derive.rs
└── tokens
    ├── fungible
    │   └── src
    │       ├── extensions
    │       │   ├── allowlist
    │       │   │   ├── mod.rs
    │       │   │   └── storage.rs
    │       │   ├── blocklist
    │       │   │   ├── mod.rs
    │       │   │   └── storage.rs
    │       │   ├── burnable
    │       │   │   ├── mod.rs
    │       │   │   └── storage.rs
    │       │   ├── capped
    │       │   │   └── storage.rs
    │       │   └── mod.rs
    │       ├── fungible.rs
    │       ├── impl_token_interface_macro.rs
    │       ├── lib.rs
    │       ├── overrides.rs
    │       ├── storage.rs
    │       └── utils
    │           ├── mod.rs
    │           ├── sac_admin_generic
    │           │   ├── mod.rs
    │           │   └── storage.rs
    │           └── sac_admin_wrapper
    │               ├── mod.rs
    │               └── storage.rs
    │
    └── non-fungible
        └── src
            ├── extensions
            │   ├── mod.rs
            │   └── royalties
            │       ├── mod.rs
            │       └── storage.rs
            ├── lib.rs
            ├── non_fungible.rs
```

```
        ├── overrides.rs
        └── storage.rs
```

System Overview

The Stellar Contracts Library is a collection of modular, reusable, and secure components for building smart contracts on the Stellar network. This audit assesses the third release candidate (RC3) of the library, which introduces significant architectural enhancements, powerful new features, and an expanded set of developer utilities. Compared to previous versions, this release focuses on improving the developer experience through a more cohesive and extensible architecture while adding highly requested, production-ready features.

Architectural Refinement of Token Standards

In RC3, one of the most significant changes is a deep architectural refactoring of the fungible and non-fungible token (NFT) modules. The previous model, which relied on composing multiple extension traits, has been replaced by a more unified and streamlined one.

This new architecture is centered around a `Base` contract type and a `ContractOverrides` trait. Core logic, such as minting and metadata handling, is now integrated directly into the `Base` implementation. This simplifies the development of standard tokens by reducing boilerplate code. For more advanced use cases, the `ContractOverrides` system allows extensions (like the new `allowlist` and `blocklist` modules) to cleanly and safely override default token behaviors, such as `transfer` and `approve`, without requiring developers to rewrite the entire token interface.

Stellar Asset Contract (SAC) Admin Modules

The SAC admin modules allow users to integrate with the SACs. The admin modules support two integration approaches: a generic approach and a wrapper approach. Taking the generic approach, the `sac-admin-generic` module leverages the `__check_auth` function to handle both authentication and authorization, enabling the injection of custom authorization logic. In contrast, the `sac-admin-wrapper` module makes the admin module function as middleware, defining specific entry points for each admin function and forwarding those calls to the corresponding SAC functions. This allows custom logic to be applied before delegation, resulting in a modular and straightforward design.

Access Control Framework

This release introduces a comprehensive and flexible framework for managing on-chain permissions, composed of two distinct modules:

- **`ownable`**: A module for implementing simple, single-owner access control. It provides a straightforward pattern for functions that require the owner's authorization via the `#[only_owner]` macro.
- **`access-control`**: A full-fledged Role-Based Access Control (RBAC) system for contracts requiring more granular permissions. It supports the creation of custom roles (e.g., `minter`, `manager`, etc.), the assignment of roles to multiple accounts, and the ability to establish administrative hierarchies by designating certain roles as admins for other roles.

Both modules feature a secure, **two-step transfer mechanism** for the primary `owner` or `admin` role. This design choice prevents the accidental loss of control by requiring the proposed new administrator to explicitly accept the role before the transfer is finalized.

Developer Experience and Macros

A primary theme of this release is the significant improvement to the developer experience, largely driven by the introduction and refinement of powerful procedural macros. These macros have been designed to reduce boilerplate code, enforce consistent security patterns, and automate common implementation tasks, making the library more accessible and robust.

- **Authorization Macros:** A new suite of attribute macros provides a clean and declarative way to implement access control:
  - `#[only_owner]`: Used with the `ownable` module, this macro restricts function execution to the single, designated contract owner.
  - `#[only_admin]`: Used with the `access-control` module, this macro restricts function execution to the top-level contract administrator.
  - `#[has_role(...)]`: A flexible macro for the `access-control` module that checks if a specified account possesses a given role (e.g., `#[has_role(caller, "minter")]`). This allows developers to easily create and enforce granular, role-based permissions for different contract functions.
- **Default Implementation Macro (`#[default_impl]`):** This existing macro has been architecturally enhanced to further simplify development. It now operates in conjunction with a `type ContractType` associated type on the core token traits. This allows

developers to simply specify the desired base behavior (e.g., `type ContractType = Base;`) and have the macro automatically generate the full, standard-compliant interface. This change drastically reduces the amount of code developers need to write for standard token implementations and has been expanded to support the new `AccessControl` and `Ownable` traits.

- **Upgradeable Contract Macros:** The existing `#[derive(Upgradeable)]` and `#[derive(UpgradeableMigratable)]` macros have been updated with a key new feature: they now automatically read the `CARGO_PKG_VERSION` environment variable during compilation and embed this version string as `binver` metadata into the contract's Wasm. This creates an immutable, on-chain record of the contract's version, providing crucial information for off-chain tooling and upgrade management.

On-Chain NFT Royalties

A new `royalties` extension has been added to the NFT module, providing a standardized mechanism for on-chain royalty payments compliant with the **ERC-2981** standard. This allows creators and developers to programmatically enforce royalty economics directly from their NFT contracts.

The extension supports:

- A **default royalty** for the entire collection, which applies to all tokens unless overridden.
- **Token-specific royalties**, which can be set at the time of minting to define unique royalty terms for individual NFTs.

Merkle Tree and Cryptographic Utilities

To support more advanced use cases like airdrops and off-chain voting, RC3 introduces a set of generic cryptographic utilities:

- A `crypto` package provides versatile hashing primitives (`Sha256`, `Keccak256`, etc.) and a generic verifier for Merkle proofs.
- A `merkle-distributor` module leverages these primitives to offer a reusable template for building applications where claims are verified on-chain via Merkle proofs.

These utilities are showcased in new example contracts, including `fungible-merkle-airdrop` and `merkle-voting`, which provide developers with practical templates for implementing these patterns.

Additional Token Extensions and Examples

Beyond the major features discussed above, this release also includes:

- New `allowlist` and `blocklist` extensions for fungible tokens, giving contract administrators fine-grained control over which accounts are permitted to interact with the token.
- New examples demonstrating patterns for creating custom administrators for the SAC, leveraging the new `access-control` framework to manage a SAC's privileged functions.

Security Model and Trust Assumptions

The security of any contract built using this library is contingent upon both the library's own correctness and a set of trust assumptions regarding its environment and usage.

- **Out-of-Scope Dependencies:** This audit assumes the security and correct functionality of the underlying Soroban environment, the target WASM compiler, external crates, and the Soroban SDK. These components form the trusted foundation upon which the contracts operate but are outside the direct scope of this review.

- **Developer Responsibility:** The Stellar Contracts Library provides a set of powerful but unopinionated primitives. It does not enforce a specific security model but rather gives developers the tools to build their own. The ultimate security of a smart contract is, therefore, critically dependent on its implementation by the developer. Key responsibilities include:

    - **Implementing Authorization:** Low-level functions, such as `Base::mint`, have been intentionally designed without built-in access control. As such, developers **must** wrap calls to these functions within functions that enforce appropriate authorization, for example, by using the provided `#[only_owner]` or `#[has_role(...)]` macros. Failure to do so would result in privileged functions being publicly callable.

    - **Secure Initialization:** The `owner` or `admin` of a contract must be set securely within the constructor. An improperly configured contract could be left without an administrator, rendering it unmanageable.
    - **Cross-Contract Call Authorization** Due to the Stellar network's authorization model, when a contract address is the invoker of a function, the `require_auth()` check passes automatically. Hence, it is assumed that if a contract address is assigned a privileged role, proper authorization checks are performed in the invoking contract instead.

- **Off-Chain Components:** Features that rely on off-chain processes, such as the `merkle-distributor`, depend on the secure and correct generation of data outside the blockchain. While this audit covers the on-chain proof-verification logic, it assumes that the Merkle root provided to the contracts has been generated correctly and is trusted.

Critical Severity

C-01 Royalty Logic Unusable for Typical NFT Market Values Due to `u32` Constraint

The `royalty_info` function is designed to implement EIP-2981 royalty calculations by returning the royalty amount owed based on a given `sale_price` and basis points. The function uses the `(sale_price * basis_points) / 10000` formula to compute the payout. However, the use of a `u32` type for the `sale_price` input imposes significant practical limitations. For example, on Stellar where USDC has 7 decimal places (1 USDC = 10,000,000), the maximum value a `u32` can hold (4,294,967,295) equates to roughly 429 USDC.

This constraint makes the function unusable for high-value sales or tokens with high decimal precision, which are common in NFT marketplaces. As a result, any sale price above this threshold cannot be used. This limitation directly undermines the intended functionality of the royalty system, particularly affecting artists and creators who rely on royalties from secondary sales. With current market dynamics, NFT sales often exceed thousands of USDC, making `u32` grossly inadequate for real-world applications.

Consider changing the `sale_price` and royalty return type to `i128`, which offers a sufficiently large range to accommodate realistic sale amounts and ensures accurate royalty distribution without sacrificing precision.

Update: *Resolved in pull request #290 at commit ad09aac and pull request #329 at commit 68397ff.*

High Severity

H-01 `#[has_role]` Macro Offers Minimal Benefit and Introduces Authorization Risks

The `#[has_role(account, "role")]` procedural macro introduces security risks by decoupling authentication (`require_auth()`) from authorization (`ensure_role`). It verifies

that an arbitrary `Address` holds a role but does not authenticate the signature of this address. This separation can lead to critical vulnerabilities that allow the code to compile successfully but violate core access control principles.

The macro's design creates a false sense of security. Developers, especially those from ecosystems like Solidity, are used to role-based modifiers that authenticate the caller. The current macro encourages incorrect assumptions, leading to insecure implementations. For example:

```
#[has_role(minter_account, "minter")]
pub fn mint_for_anyone(e: &Env, minter_account: Address, recipient: Address) {
    // Missing minter_account.require_auth() allows an attacker to
    // impersonate a valid minter by simply passing their address.
    internal_mint(e, &recipient);
}
```

Even when used correctly, the pattern is confusing and forces developers and auditors to meticulously verify that the variable in the macro matches the one in the signature and that the `require_auth()` call exists for that exact variable. The security of the function hinges on a single, easy-to-miss line of code. This increases the likelihood of errors during development and review. Furthermore, the macro provides minimal ergonomic value. It saves only a single line of code compared to directly calling `ensure_role`, while introducing significant potential for misuse. This weak tradeoff undermines its utility.

Consider introducing a more secure and developer-friendly macro like `#[only_role(caller, "role")]` (similar to the popular modifier in the OpenZeppelin library for Solidity), which would automatically inject both the authentication and authorization checks. This approach aligns with expectations, eliminates ambiguity, and enforces security by default, truly leveraging the power of macros to reduce boilerplate without compromising safety.

Update: *Resolved in pull request #318 at commit b303e42.*

Medium Severity

M-01 Missing Checks on `Spender` in Fungible BlockList Extension

The `transfer_from` function of the `blocklist` extension does not validate whether the `spender` is blocklisted or not. As a result, a blocklisted address can still execute `transfer_from` operations on behalf of non-blocklisted users, provided it has been previously approved through `approve`. This is contradictory to the `BlockList` trait which specifies that blocked accounts cannot transfer tokens.

This can cause confusion where a `from` account might assume that a blocklisted spender is not allowed to interact with the smart contract despite the spender being able to still transfer token from pre-approved allowance, thereby undermining the integrity of the blocklist mechanism. A practical exploit scenario could involve a compromised contract that has amassed user approvals. Even after being blocklisted, the contract would retain the ability to drain user funds via `transfer_from`, bypassing the intended protections of the blocklist system.

In the `transfer_from` function, consider adding a check which ensures that the `from` and `spender` addresses are not blocklisted.

Update: *Partially resolved. The contract still allows blacklisted `spender` to interact with pre-approved tokens. However, documentation has been added to warn about this behavior in pull request #307 at commit 14813be. The OpenZeppelin Stellar development team stated:*

> *This is a deliberate choice following the convention from other ecosystems like Ethereum.*

M-02 Missing Functionality to Renounce Admin

The `access-control` trait does not provide a way for the admin to renounce their role. This is problematic because the `Admin` key is stored in instance storage and will continue to be accounted for during TTL extensions, even if it is no longer in use, thereby increasing the cost of the operation. In addition, without a renounce mechanism, there is no way to intentionally make a contract permanently admin-less, which is often desirable for decentralization or upgrading purposes.

Consider adding a `renounce_admin` function that allows the current admin to remove themselves from the role.

Update: *Resolved in pull request #316 at commit 0b8dbbd.*

M-03 `ensure_if_admin_or_admin_role` Panics When `Admin` Is Not Set and Ignores Role Admins

The `ensure_if_admin_or_admin_role` function is intended to authorize callers who are either the contract admin or the admin of a specific role. However, the current implementation checks for the contract admin first and calls `get_admin`, which panics with an `AdminNotSet` error if the `Admin` key is not present in storage. This prevents the function from proceeding to check if the caller is the role admin, effectively disabling access to sensitive role-based functions like `grant_role` and `revoke_role` in contracts without a contract-wide admin.

The `AccessControl` trait states that a contract must set the admin to be able to use the trait's functionalities. However, given that this is a library, it should provide enough flexibility to allow for setting up roles in an admin-less contract. Imagine a scenario where an admin assigns the required roles and role admins and then renounces its power. In such an event, the respective role admins should be able to control the access to their respective roles.

Consider handling the `AdminNotSet` error returned by `get_admin` within the `ensure_if_admin_or_admin_role` function and ensuring that the TTL of the `Admin` key is timely extended.

Update: *Resolved in pull request #292 at commit 132ecab.*

M-04 Inconsistent Instance Storage TTL Extension

Throughout the codebase, the TTL for instance storage is not extended anywhere except in the `get_owner` function of the `ownable` trait. This creates an inconsistent pattern in how TTL extension is handled. Unlike other storage types, extending the TTL of instance storage affects the entire contract storage and can be costly. At the same time, not extending the TTL when needed can result in the loss of key functionality and may render the entire smart contract inaccessible once the TTL expires. As a result, it is important to clarify whether TTL extension should be handled within the library or explicitly by integrators.

Consider either extending the instance TTL throughout the codebase or removing the extension in the `get_owner` function, and clearly documenting that TTL extension is the responsibility of the integrators.

Update: *Resolved in pull request #293 at commit 4775033 and pull request #329 at commit 68397ff.*

M-05 Lack of `FungibleBurnable` Implementation on `AllowList` and `BlockList` Extensions

The `FungibleBurnable` trait provides two key functions: `burn` and `burn_from`. These are part of the SEP-0041 token interface and should be available to any fungible token extension. Currently, these functions are implemented on the `Base` contract type. However, when attempting to use the `FungibleBurnable` interface with other extensions such as `AllowList` or `BlockList`, the expected methods are not found due to a missing implementation of the trait on those specific types. This prevents users from generating default implementations using `default_impl_macro` with the `contractType` associated type specified as other than the `Base` type.

Consider adding `burn` and `burn_from` functions to the `AllowList` and `BlockList` implementations.

Update: *Resolved in pull request #294 at commit be401b0.*

Low Severity

L-01 Token-Specific Royalties Cannot Be Removed

The NFT royalty extension provides functions to define both default collection-wide royalties and token-specific royalties that override the default. The `set_token_royalty` function allows setting royalties for individual tokens, while `set_default_royalty` establishes the collection-level default. However, the implementation lacks a mechanism to remove token-specific royalties once they have been set.

This creates rigidity in the system: tokens with customized royalties cannot revert to using the collection default. Mimicking the default would require explicitly storing the same values in the token-specific entry, which is both redundant and inefficient. The `royalty_info` function always prioritizes token-specific settings over the default, meaning that once a token has been individually configured, it becomes permanently isolated from future changes to the collection default.

In practice, NFT collections often need to adjust royalty strategies over time, for example, reducing royalties to increase trading volume or changing recipients due to organizational updates. Without a way to remove token-specific overrides, these tokens become disconnected from evolving collection policies. This results in a fragmented royalty structure and limits flexibility.

Consider introducing a `remove_token_royalty` function that explicitly deletes token-specific royalty key from persistent storage, allowing tokens to fall back to the collection-wide default.

Update: *Resolved in pull request #296 at commit 4a27a5e.*

L-02 Macros Accept and Silently Ignore Arguments

Several procedural macros in the codebase accept and silently ignore arguments passed through the `TokenStream`, which can lead to confusion and misuse. For example, a macro like `#[only_admin(caller)]` accepts the `caller` argument syntactically, but the implementation disregards it entirely.

Specifically, the following macros ignore their argument inputs:

- `#[only_admin]` — ignores `_attrs: TokenStream` in `access-control-macros`
- `#[only_owner]` — ignores `_attrs: TokenStream` in `ownable-macro`
- `#[default_impl]` — ignores `_attr: TokenStream` in `default-impl-macro`

By accepting and then silently discarding their arguments, these macros give a false impression of customization: users may believe that they are configuring behavior (whether access checks or default implementations), whereas in reality, nothing happens. That mismatch between appearance and action not only muddies the code's intent and complicates reviews, but also provides an easy vector for bad actors to insert deceptive annotations that mask unauthorized logic or unexpected defaults.

To improve clarity and developer experience, consider explicitly rejecting unused arguments in the aforementioned macros. This can be done by checking if `attr.is_empty()` and triggering a compile-time panic with a message such as:

```
assert!(attr.is_empty(), "This macro does not accept any arguments");
```

This would help prevent misleading usage and improve the reliability and transparency of macro-based abstractions.

Update: *Resolved in pull request #295 at commit 966775e and pull request #327 at commit 110f6ce.*

L-03 Unrestricted Proof Size and Leaf Index in Merkle Distributor

The `set_root` function of the `merkle-distributor` module allows the root to be set only once, indicating that the tree has a static size (let's assume n). Consequently, there should be at most n leaves and, therefore, a maximum of n claims permitted. For a full binary Merkle tree, the height should be $log_2(n)$. However, when the verify_and_set_claimed function is called, there is no validation of the `proof` length (to ensure that it matches the tree height) or the `index` value (to confirm that it is less than n). This oversight could allow an attacker to forge a proof with an incorrect length that is either shorter or longer than the tree's actual height.

Since the tree is static with a predetermined size, consider including the tree size (or maximum number of leaves) as a variable alongside the root. This would allow for placing restrictions on `proof` length and leaf `index`, improving security against potential proof forgery.

Update: *Resolved in pull request #322 at commit 50cd251.*

L-04 Missing Function to Retrieve All Role Members

The `access-control` module implements access control through `get_role_member(role, index)` and `get_role_member_count(role)`, allowing enumeration of role members. However, it lacks a single-call getter that returns all members of a given role, as provided by the `getRoleMembers(bytes32 role)` function of OpenZeppelin's `AccessControlEnumerable` contract.

While it is understandable that the contract omits this function to avoid unbounded calls that could exhaust resource limits, the absence of a batch accessor may complicate off-chain integrations. Clients are required to manually iterate over indices from 0 to `get_role_member_count(role) - 1` to reconstruct the full list of role members, which introduces additional complexity and potential for integration bugs.

Consider adding a function that returns the complete set of members for a given role, with accompanying documentation that warns of potential resource limit issues for large sets. If this functionality is intentionally omitted, explicitly documenting this design choice would help guide developers.

Update: *Resolved in pull request #311 at commit 2bd717a and pull request #330 at commit 41060f2.*

L-05 `transfer_role` Panics if `live_until_ledger` Exceeds Maximum TTL for Temporary Entries

The `transfer_role` function allows for setting a `live_until_ledger` value to specify the expiration of a temporary entry. However, if the provided `live_until_ledger` implies a TTL that exceeds the maximum limit for temporary entries, the function will panic. This behavior diverges from the treatment of persistent entries, where TTLs are clamped to the maximum instead of causing a panic. This can result in unexpected failures, especially since the caller may reasonably expect the function to handle such cases gracefully. A panic in this context is not user-friendly.

Consider validating the `live_until_ledger` input before using it, or clamping the TTL to the maximum allowed value for temporary entries, thereby matching the behavior seen with persistent entries. Moreover, this edge case should be clearly documented to prevent misuse and ensure predictable contract behavior.

Update: *Resolved in pull request #298 at commit ccad351.*

L-06 Lack of Validation

The `set_admin` and `set_owner` functions are designed to be called only once within the lifecycle of a smart contract. Any further setting of these roles should be done through the

`transfer_admin_role` and `transfer_ownership` functions, respectively. However, there is no validation preventing the setter functions from being called multiple times.

Consider adding a check to verify whether the respective key is already set using `e.storage().instance().has(&key)`, and reverting with a panic with an appropriate error (e.g., `AdminAlreadySet` or `OwnerAlreadySet`) if it exists.

Update: *Resolved in pull request #299 at commit 16bdda0.*

L-07 Misleading Naming of Core Contracts May Confuse Developers Familiar with `openzeppelin-contracts`

Several contracts in the library use names that are widely recognized within the EVM ecosystem, such as `Ownable` and `AccessControl`, but implement variants of the commonly expected logic. For example, the contract named `Ownable` implements a two-step ownership transfer pattern, which aligns more closely with `Ownable2Step` in the OpenZeppelin EVM Contracts library. Similarly, the `AccessControl` implementation deviates from the standard EVM implementation.

Given that the library is being developed under the OpenZeppelin brand for the Stellar ecosystem, users transitioning from EVM environments may assume identical semantics and usage patterns based on familiar contract names. This could lead to incorrect assumptions about functionality, incorrect integrations, or even security vulnerabilities if developers rely on implicit behavior that is not present in these contracts.

Consider renaming these contracts to more accurately reflect their behavior. Alternatively, consider providing clear documentation disclaimers and naming clarifications to help developers properly understand the distinctions.

Update: *Resolved in pull request #328 at commit e47d949 and pull request #330 at commit 41060f2.*

L-08 Missing Documentation for Unsafe Functions

Both the `set_root` and `set_claimed` functions can perform sensitive state updates without any authorization. From the example usage, `set_root` should be used only in the constructor and `set_claimed` can be a private function.

For unsafe functions, consider adding documentation demonstrating their safe usage along with warnings regarding their lack of authorization.

Update: *Resolved in pull request #322 at commit 50cd251.*

L-09 Merkle Tree Verification Can Be More General

The `crypto` module only supports custom Merkle trees that have been generated using a commutative hash function. Looking at the documentation, it appears that this may be motivated by the OpenZeppelin/merkle-tree JavaScript library tree generation scheme, hence the warning when only using `keccak256` hash function for hashing leaves. Considering that the OpenZeppelin/merkle-tree library generates Merkle trees for membership inclusion proofs with double `keccak256` hash function for leaves on `abi.encode`ed underlying values and sorted hashing for internal nodes, it is rather adapted for the Ethereum context.

On the Stellar network, the encoding scheme is XDR and the `sha256` hash function seems more widely used. As such, it is less likely for developers to use the trees adapted to the Ethereum context. The limited support for Merkle tree verification to only commutative hashing hinders wider developer adoption, as most direct applications of any hash functions, be it `sha256` or `keccak256`, are not commutative. Furthermore, trees cumulatively built on-chain are naturally order sensitive, and it also requires more computation to sort and then hash in tree generation.

Consider supporting the simpler and also less restrictive plain hashing scheme without sorting first and providing relevant documentation that is more suitable for the Stellar network.

Update: *Resolved in pull request #321 at commit 79af2d9.*

L-10 Support Multiple Roles in `has_role` Macro

The current `has_role` macro only verifies a single role and reverts if the caller lacks it, forcing developers to duplicate logic in case several roles grant the same permission. This increases boilerplate code and obscures the intended access policy. Allowing `has_role` to accept multiple roles would enable specifying an array of roles and granting access as soon as the caller holds any one of them. This change enhances clarity and reduces repetitive code.

Consider extending the `has_role` macro's signature so that it takes a list of roles, iterating through each role, and allowing execution if any check passes, while maintaining backward compatibility for single-role checks.

Update: *Resolved in pull request #325 at commit e7251be.*

L-11 Potential Circular Admin

In the `access-control` module, one can potentially assign the RoleAdmin in a circular manner. For instance, it is possible to assign `MINT_ADMIN` to be the admin of `MINT_ROLE` and, at the same time, have `MINT_ROLE` be the admin of `MINT_ADMIN`. However, there could be unintended consequences when such a situation occurs. For instance, it could create a race condition for one role to revoke the other when they are each other's admin.

Since set_role_admin_no_auth is a restricted function, consider warning about the aforementioned behavior if it cannot be prevented in the code.

Update: *Resolved in pull request #312 at commit e0a8118.*

L-12 Suboptimal Storage and TTL Strategies

Throughout the codebase, multiple opportunities for code optimization were identified:

- Functions like `block_user`, `unblock_user`, `allow_user`, and `disallow_user` always perform a storage write and emit an event, even if the existing value has not changed.
- `allowed` and `blocked` functions extend the TTL even if the stored boolean is `false`.
- `disallow_user` and `unblock_user` explicitly store `false` instead of removing the entry, leading to unnecessary storage fees and TTL tracking overhead.

To avoid incurring unnecessary costs, consider adding `if` checks to perform storage writes (and emit events) only when the new state differs from the current one. In addition, when revoking access or unblocking users, consider removing entries using the following code instead of writing `false`.

```
e.storage().persistent().remove(&key);
```

Update: *Resolved in pull request #303 at commit bf51ae5.*

L-13 Duplication of code

Duplication of code can result in redundant operations and waste computational cost.

In the `grant_role_no_auth` function, the `AccessControlStorageKey::HasRole(account, role)` key is first set in the `add_to_role_enumeration` function and then set again in the `grant_role_no_auth` function.

Consider removing any instances duplicate code.

Update: *Resolved in pull request #304 at commit 7158c50.*

L-14 Unused `AccessControlStorageKeys` Can Be Removed

`AccessControlStorageKeys.RoleAdmin(RoleSymbol)` can be set and reset using the same role key in `set_role_admin` function. However, for a role key that is no longer in use, it cannot be removed. Each time the `get_role_admin` function is invoked, its TTL can be extended, thus increasing the cost for the invoker.

Similarly, when `AccessControlStorageKeys.RoleAccountsCount(RoleSymbol)` is zero (e.g., when the role symbol is no longer in use) the ledger key cannot be removed. Hence, when the `get_role_member_count` function is invoked, the TTL can be extended, thereby increasing the cost for the invoker.

Consider allowing the admin to remove unused keys for roles that are no longer in use.

Update: *Resolved in pull request #306 at commit d9d3ab5. The pull request introduces the `remove_role_admin_no_auth` and `remove_role_accounts_count_no_auth` functions along with a warning that these functions do not implement any authorization.*

L-15 `transfer_role` TTL Extension-Only Policy May Exceed Intended Expiration

The current documentation for `transfer_role` implies that setting `live_until_ledger` will cause the pending role entry to expire exactly at that ledger. In reality, Soroban's `extend_ttl` can **only** increase an entry's TTL if its remaining TTL is below the given threshold. It cannot shorten or reset a larger, default TTL. As a result, when `live_for =`

`live_until_ledger − current_ledger` is smaller than the temporary entry's default TTL, the call to

```
e.storage().temporary().extend_ttl(pending_key, live_for, live_for);
```

has no effect. The entry will then remain active until its original TTL elapses, not at `live_until_ledger`, causing potential user confusion.

To correct this misleading documentation, consider updating the function's comments to explicitly warn that:

- `live_until_ledger` is an **upper bound** rather than a guaranteed expiration time
- Soroban's TTL policy **only extends** a key's TTL and **cannot reduce** an existing TTL
- if the computed `live_for` is shorter than the default minimum TTL, the entry will outlive `live_until_ledger`

Including these clarifications in the documentation will help ensure developers understand that actual expiration may exceed the specified ledger.

Update: *Resolved in pull request #323 at commit 72053cb.*

L-16 Misleading and Inaccurate Documentation

Throughout the codebase, multiple instances of misleading and/or inaccurate documentation were identified:

1. The `FungibleAllowList` trait's documentation claims that a non-allowlisted account cannot execute transfers or approvals. However, in practice, a non-allowlisted spender can still transfer tokens on behalf of an approved holder. Likewise, the `FungibleBlockList` documentation states that blocked accounts cannot transfer or approve tokens, yet blocklisted spenders are able to transfer tokens they have already been approved to move. Consider updating both trait descriptions to clarify that these lists only restrict direct operations by non-listed accounts and allow transfers performed via existing approvals. Alternatively, consider changing the implementation so it reflects the documentation.

2. Across the codebase, `caller` is being used inconsistently. For example, `enforce_admin_auth` does not accept a `caller` parameter, so references to `caller` actually denote the transaction invoker. Choose one term (e.g., "invoker") and apply it uniformly throughout the documentation to eliminate ambiguity.

3. The Royalties trait is described as "following the ERC-2981 standard". However, since ERC-2981 is EVM-specific, rephrase the comment to state that this implementation is inspired by ERC-2981 and adapts its logic for Stellar's environment.

4. The documentation for the `#[only_owner]` macro shows an example expansion that does not match the code injected by the macro. The real expansion is:

   ```rust
   stellar_ownable::enforce_owner_auth(e);
   ```

   Amend the example to reflect this exact injected call.

Applying these changes will ensure that the documentation remains accurate, coherent, and aligned with the actual behavior of the codebase.

Update: *Resolved in pull request #307 at commit 14813be and pull request #329 at commit 68397ff.*

Notes & Additional Information

N-01 Inconsistent Folder Structure

Unlike other libraries in the project that separate implementation (`src/*.rs`) from tests (`src/test.rs`), `crypto` traits such as `hashable.rs` place both public functions and tests in the same file.

To improve the consistency and clarity of the codebase, consider aligning the `crypto` directory with the overall structure by moving the tests into a separate file.

Update: *Resolved in pull request #321 at commit 79af2d9.*

N-02 Inconsistency in Panic Handling

There is an inconsistency in how the codebase handles panics related to missing keys and authorization failures.

For example, the `get_admin` function retrieves the `Admin` key from instance storage and panics directly if the key is not found. This function is used within `enforce_admin_auth`, which performs an implicit panic on failed authorization without surfacing a meaningful or structured error. In contrast, the `get_owner` function returns `None` if the `Owner` key is missing. When used in `enforce_owner_auth`, it checks authorization and panics with a `NotAuthorized` error. However, this error might be misleading if the actual problem is that the owner key does not exist.

Consider standardizing the handling of missing keys and failed authorizations by using `panic_with_error!` with clear and distinct error messages.

Update: *Resolved in pull request #326 at commit 9a396fe.*

N-03 Potentially Increasing Constants

Currently, all TTL threshold and extension amount values are defined in a single file, `constants/src/lib.rs`. While this works for now, as the codebase and number of libraries grow, this centralized approach may become harder to maintain. Managing all constants in one location can lead to readability issues.

Consider modularizing the constants in `constants/src/lib.rs` by introducing a dedicated `constants.rs` file at each relevant directory or module level. This would promote better organization, encapsulation, and easier maintainability as the project evolves.

Update: *Acknowledged, will resolve. The OpenZeppelin Stellar development team stated:*

> *This is a good suggestion. But we are planning to do that when the project grows. Right now, this will add unnecessary complexity to the project. It is more compact as it is.*

N-04 Admin Role Transfer Lacks Enforced Delay

The `transfer_admin_role` function of the `access-control` module includes a `live_until_ledger` parameter to limit the window during which the transfer can be accepted, but it does not enforce a mandatory delay between the initiation and acceptance of the transfer. As a result, the new admin can accept the role in the very next ledger, effectively enabling instantaneous role transfer.

This differs from the OpenZeppelin `AccessControlDefaultAdminRules` contract, which enforces a minimum delay between initiation (`beginDefaultAdminTransfer`) and acceptance (`acceptDefaultAdminTransfer`). That delay serves as a crucial safeguard

against malicious or compromised admin transfers by giving stakeholders time to detect suspicious activity and coordinate a response.

Without a forced delay, the system becomes vulnerable to rapid takeover, leaving no time for community review, off-chain coordination, or emergency intervention. A malicious proposal or mistaken transfer could be finalized before anyone notices. A time-lock mechanism would provide a low-friction, high-value protection layer. It buys time for token-holders, integrators, and governance participants to react, pause contracts, or raise red flags. It also gives the current admin a chance to cancel or roll back the transfer if new concerns emerge after initiation.

To align with best practices in secure governance tooling, consider adding a minimum enforced delay between the initiation and acceptance of admin transfers.

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

> *The delay functionalities provide indeed another security layer, but similar features are expected to be developed in a dedicated timelock contract that can be used to achieve comparable results. Having delays separately developed for this particular module seems to be unnecessary duplication work.*

N-05 Vulnerable Dependencies

Running `cargo audit` revealed two known vulnerabilities, `curve25519-dalek 3.2.0` and `ed25519-dalek 1.0.1`, in the transitive dependencies used by `sac-admin-generic-example 0.3.0`.

While the aforementioned dependencies have only been used in examples, consider upgrading them to their latest, non-vulnerable versions. Alternatively, consider refactoring or removing the affected example code to eliminate reliance on vulnerable crates.

Update: *Resolved in pull request #319 at commit a619477.*

N-06 Missing Warning When Changing SAC Admin Address

The Stellar documentation explicitly warns that when updating a SAC Admin address, the new admin address is not validated during the change. As a result, it is possible to unintentionally

assign an invalid or incorrect admin address, which can irreversibly lock administrative control of the SAC.

While this is true for any administrative smart contract, consider adding this warning to SAC admin-related library documentation to reflect the risks and align with the Stellar documentation.

Update: *Resolved in pull request #320 at commit 873a416.*

N-07 Typographical Error

Throughout the codebase, multiple instances of typographical errors were identified:

- In line 64 of `access-control/src/access_control.rs`, "has" should be "have".
- In line 44 of `access-control/src/lib.rs`, "to with" should be "to go with".
- In line 22 of `default-impl-macro/src/lib.rs`, "macro's" should be "macros".

Consider correcting all instances of typographical errors in order to improve the clarity and readability of the codebase.

Update: *Resolved in pull request #308 at commit dab8fc5.*

N-08 Overloaded Error Obscures Distinct Failure Cases

The current implementations of certain errors conflate multiple, semantically distinct failure scenarios into a single generic error, significantly impairing clarity and troubleshooting for developers. Despite occurring in different contexts, the error message remains the same, offering no information about the underlying cause of failure:

- The `AccessControlError::AccountNotFound` error is used to signal various unrelated issues. In the `get_role_member` function, an error is triggered when the provided index is out of bounds for the role's member list, where a more descriptive error like `IndexOutOfBounds` would be appropriate. In both `revoke_role` and `renounce_role` functions, an error occurs when the target account does not have the specified role, which would be better represented as `RoleNotHeld`. In the `remove_from_role_enumeration` function, an error may arise from the role having no members or the specified account not being part of the role, which would be more accurately captured by distinct errors such as `RoleIsEmpty` and `RoleNotHeld`.

- The `AccessControlError::Unauthorized` error is used in `ensure_if_admin_or_admin_role` function to indicate that the caller is neither contract admin nor role admin, however, this error is also thrown when there is no role admin is set, which could be more accurately captured by an error such as `RoleAdminNotHeld`.
- When a non-existing role is used with the `has_role` macro, the `AccessControlError::Unauthorized` error message is returned, which is the same error that is returned when an account does not have an existing role.

Consider refactoring the `AccessControlError` enum to include more granular variants to reflect correct error messages. This change provides immediate, context-relevant feedback to developers and improves the maintainability of the contract.

Update: *Resolved in pull request #309 at commit 86d038d.*

N-09 Edge Case in `transfer_role` Results in `NoPendingTransfer`

The `live_for` value is calculated as the difference between `live_until_ledger` and `current_ledger`. If the `live_until_ledger` and `current_ledger` values are equal, `live_for` becomes zero. Consequently, the `pending_key` is set but the TTL is not extended, resulting in a no-op and wasting computational resources. The acceptance of the transfer will be limited to the current ledger which could be a very short time frame, and will incur additional cost if the transfer were to be initiated again.

Consider checking whether `live_for` meets a minimum threshold before attempting TTL extension and ensuring that the intended TTL duration is sufficient for practical use.

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

> *In order to cover the edge case, the suggestion is to put extra checks, which will make the general use case more costly. We simply do not think that the pros of this approach outweigh the cons of worsening the general use case.*

N-10 Naming Suggestions

Throughout the codebase, multiple opportunities for improved naming were identified:

- The `validate_param_type` function can be renamed to `validate_address_type` since it is only validating the `Address` type.
- Use `unsafe` in function names to be more explicit about the danger. This is a common pattern in Rust for unsafe operations:
  - `grant_role_no_auth` -> `unsafe_grant_role`
  - `revoke_role_no_auth` -> `unsafe_revoke_role`
  - `set_role_admin_no_auth` -> `unsafe_set_role_admin`
- In the Merkle tree context, `node` often refers to the internal node, and `leaf` is used to refer to a member of the tree. As such, `IndexableNode` can be more accurately renamed to `IndexableLeaf`. Similarly, the node function argument can be named `leaf` and the local variable leaf can be named `leafHash` to differentiate it from the argument.

Consider implementing the above-listed suggestions to improve the clarity and maintainability of the codebase.

Update: *Partially resolved in pull request #313 at commit f0a4dae and pull request #322 at commit 50cd251. The OpenZeppelin Stellar development team stated:*

> *We have accepted 2 of the 3 suggestions. However, we will not replace `no_auth` with `unsafe`. This is because in Rust, `unsafe` has a different meaning. It means that we are opting out of the Rust compiler's safety guarantees for more flexibility and optimization. This is not the case for our `no_auth` functions.*

N-11 Unused Return Value

The `enforce_admin_auth` and `enforce_owner_auth` functions return the `admin` and `owner` values, respectively. However, these values are not used in the context of `onlyAdmin` or `onlyOwner` macro making the usage of the macro slightly more expensive. The `enforce_owner_auth` is called in the `transfer_ownership` function, where the `owner` return value is being used. On the contrary, the `transfer_admin_role` calls the `get_admin` and `admin.require_auth()` instead of calling the `enforce_admin_auth` function. Having a consistent approach to using these return values will improve code readability.

To improve the consistency and maintainability of the codebase, consider either using the return values in the `transfer_admin_role` function or removing them altogether.

Update: *Resolved in pull request #315 at commit a954d5d.*

N-12 Unnecessary TTL Extension in Renounce Ownership

In the `renounce_ownership` function, the `PendingOwner` key in temporary storage is checked to determine whether an ownership transfer is in progress. If the key is present, the TTL is extended and immediately after this, the function reverts, indicating that if the ownership transfer is pending, renouncing of ownership should not be allowed. Given that the function panics immediately after the TTL extension, the extension is rolled back and has no effect, which leads to wasted computation cost.

Consider removing the TTL extension from the `renounce_ownership` function.

Update: *Resolved in pull request #314 at commit a15bf50.*

Client Reported

CR-01 Lack of Events in Royalties

The `royalties` extension for the `non-fungible` trait does not emit any events when royalties are set.

Consider emitting an event upon royalty-related state changes.

Update: *Resolved in pull request #251 at commit 4267970.*

CR-02 Missing Default Implementations

Some supported traits do not have default implementations via the `default_impl` macro. Such traits include FungibleAllowList, FungibleBlockList, and NonFungibleRoyalties.

Consider including the default implementation for all supported traits.

Update: *Resolved in pull request #252 at commit e7d8120.*

Conclusion

The audited codebase introduces Release Candidate 3 (RC3) of the Stellar Contracts Library, which brings significant architectural improvements, enhanced developer tooling, and expanded token and access control functionality. The redesign of the `tokens` module into a unified `Base` with `ContractOverrides` model, along with the introduction of a robust role-based access control mechanism, reflects a thoughtful and forward-looking evolution of the system.

Overall, the codebase is well-structured and benefits from strong modularity and developer ergonomics, particularly through the use of expressive procedural macros. However, some areas, such as the enforcement of security and critical patterns like authorization and initialization, remain the responsibility of the developer and could benefit from additional guidance or built-in protections.

The OpenZeppelin Stellar development team was responsive and provided detailed context throughout the engagement. The audit team appreciates the opportunity to review this release and contribute to the continued security and reliability of the Stellar ecosystem.