OpenZeppelin | security

Stellar Contracts Library v0.2.0 Audit

July 3, 2025

Table of Contents

Summary

Type	Library
Timeline	From 2025-03-31 To 2025-04-09
Languages	Rust (Soroban)

Total Issues	21 (21 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	3 (3 resolved)
Low Severity Issues	9 (9 resolved)
Notes & Additional Information	8 (8 resolved)

Scope

OpenZeppelin conducted a diff audit of the OpenZeppelin/stellar-contracts repository at commit fc03869 against commit ca8da91.

In scope were the changes made to the following files:

```
packages/
├── constants/
│   └── src/
│       └── lib.rs
├── contract-utils/
│   ├── default-impl-macro/
│   │   └── src/
│   │       ├── helper.rs
│   │       └── lib.rs
│   ├── upgradeable/
│   │   └── src/
│   │       ├── lib.rs
│   │       ├── storage.rs
│   │       └── upgradeable.rs
│   └── upgradeable-macros/
│       └── src/
│           ├── derive.rs
│           └── lib.rs
└── tokens/
    ├── fungible/
    │   └── src/
    │       ├── extensions/
    │       │   ├── capped/
    │       │   │   ├── mod.rs
    │       │   │   └── storage.rs
    │       │   └── mintable/
    │       │       └── mod.rs
    │       └── impl_token_interface_macro.rs
    └── non-fungible/
        └── src/
            ├── extensions/
            │   ├── burnable/
            │   │   ├── mod.rs
            │   │   └── storage.rs
            │   ├── consecutive/
            │   │   ├── mod.rs
            │   │   └── storage.rs
            │   ├── enumerable/
            │   │   ├── mod.rs
            │   │   └── storage.rs
            │   └── mod.rs
            ├── utils/
```

```
│   ├── mod.rs
│   └── sequential/
│       ├── mod.rs
│       └── storage.rs
├── lib.rs
├── non_fungible.rs
├── overrides.rs
└── storage.rs
```

Update: *All audit changes were last merged in [pull request #217] at commit [d3741c3].*

System Overview

The audited codebase contains a set of smart contract components that could be used for developing smart contracts for the Stellar blockchain. For more details regarding the components which were previously audited, please refer to the previous audit report.

New changes introduced to the codebase include the addition of the Upgradeability Framework, the new `capped` extension for the fungible token implementation, a non-fungible token (NFT) implementation, and utilities - including macros that allow for easy derivation of function implementations for the contracts.

The following sections describe each component in more detail.

Upgradeability Framework

This framework defines a minimal system for managing contract upgrades, with optional support for handling migrations and rollbacks in a structured and safe manner. It consists of two main traits: `Upgradeable` and `Migratable`.

`Upgradeable`

The `Upgradeable` trait should be implemented by any contract that is designed to be upgradeable. It provides the `upgrade` function along with its default implementation which can be easily derived. This default implementation requires developers to additionally implement the `_upgrade_auth` function, which should contain the authorization logic. This design provides developers with flexibility regarding how the authorization looks like. In particular, they are able to implement any desired logic, which does not have to be restricted to just checking the signature of a single entity.

`Migratable`

The `Migratable` trait is optional and should be implemented by any contract that contains migration- and rollback-specific logic. It exposes two functions: `migrate` and `rollback`, where the former is expected to handle migration logic and the latter is to be used only when there is a need to roll back the changes introduced during the upgrade. The implementations for both of these functions can be automatically derived and require implementing two internal

functions: `_migrate` and `_rollback`, which are expected to contain both the authorization logic and the migration or rollback logic that is specific to a given contract.

`capped` Extension for Fungible Tokens

This simple extension provides a set of helper functions that allow for setting, modifying, and querying the current maximum total supply of tokens. It also contains a function that validates that the new token supply after minting new tokens does not exceed the cap that has been imposed. These functions are not included in any trait and the developers are responsible for ensuring that they are used correctly. This includes providing sufficient access control for setting the cap and verifying the new token supply against the cap during each token minting.

NFT Implementation

The base implementation of non-fungible tokens has been provided along with multiple extensions, including `Burnable`, `Consecutive`, and `Enumerable`.

The base implementation is based on the Non-Fungible Tokens SEP, which is similar to the ERC-721 standard for Ethereum, but contains some notable differences. These include a lack of "safe transfers", which invoke calls on the recipient in order to ensure that they are capable of receiving the tokens. The implementation provides ready-to-use functions that handle querying the information and modifying the state. The first type of function includes querying the token owner address, the balance of a given user, approval for a given token, and metadata information. The second group of functions specifies the logic for minting, approving, and transferring the tokens and setting the metadata.

`Burnable`

The `Burnable` extension provides two additional functions: `burn` and `burn_from`, which could be used to remove given tokens from circulation. Both implementations already provide an access control and verify token existence.

`Consecutive`

The `Consecutive` extension enables users to efficiently mint multiple tokens with consecutive IDs to a single recipient in a single transaction. Instead of assigning ownership to each token individually, only the first token in the minted batch is explicitly assigned an owner. The `owner_of` function—implemented specifically for this extension—handles ownership verification for the remaining tokens in the batch.

In order to accommodate this new logic, it was also necessary to override some other functions from the default implementation (e.g., `approve` or `transfer`). It is worth noting that it is not trivial to implement the `owner_of` function for this extension in an efficient way due to the restrictive read limits imposed on each smart contract transaction on Stellar. The current implementation involving an iteration over multiple tokens will not work for bigger batches and, hence, a more reliable solution is needed.

`Enumerable`

The `Enumerable` extension allows anyone to access tokens either from a given owner or from the global list, by using just their indexes. As a result, it is possible to efficiently enumerate all the tokens belonging to a given owner. The extension works by inserting new tokens into both lists whenever they are minted and removing them from both lists when they are burned. Furthermore, the lists of owners are modified during token transfers.

It is worth noting that different extensions may contain different low-level functions that handle minting tokens. For example, a contract may use the `mint` or `non_sequential_mint` function allowing to mint an NFT with any given ID, but it can also use a `sequential_mint` or `batch_mint` function, which will mint tokens starting from the first available ID. It is the responsibility of developers to provide relevant access control for these functions and ensure that the NFTs minted by them do not conflict with each other and that the usage of both methods does not introduce any vulnerabilities to the implementation. It is highly recommended to only use one type of mint function in the NFT implementation.

Utilities

The utilities implemented in the codebase allow the developers to easily generate default implementations for `Upgradeable` and `Migratable` traits using the `#[derive(...)]` macro, for both fungible and non-fungible tokens, by using the `#[default_impl]` macro. Furthermore, the `impl_token_interface!` macro has been added in order to generate the implementation of `TokenInterface` for the fungible tokens.

Security Model and Trust Assumptions

The Stellar Contracts Library fundamentally relies on the security of the Soroban SDK and the supporting crates that facilitate secure macro development. We assume that these out-of-scope dependencies are both safe and actively maintained. Furthermore, since the codebase has been designed for high flexibility, it is the responsibility of the developers to use the implemented functions correctly and to provide necessary access control for the low-level functions. It is also essential for the developers to extensively study the documentation of each implemented component before using it.

High Severity

H-01 Integer Underflow in `approve_for_all`

The `approve_for_all` function implemented for the `Base` type can be used to set or remove approval for operators managing tokens on behalf of the owner. In order to revoke an approval, the owner can specify the `live_until_ledger` parameter as 0. In such a case, the specified operator is removed from the list of approved operators of the owner and the TTL of the entire list is extended.

However, the value of the TTL extension is calculated as `live_until_ledger - e.ledger().sequence()`, and since `live_until_ledger` is 0, when approval is revoked, the integer underflow will happen during this subtraction. As a result, the code will attempt to set the TTL to a very high value which will cause panic in the Host.

Consider increasing the TTL by a constant value or not increasing it at all when approvals are revoked.

Update: *Resolved in pull request #170 at commit 9f6a748.*

Medium Severity

M-01 Insufficient Tests

While the codebase contains an extensive testing suite, not all of its parts are sufficiently tested:

- There are no tests for the `[#default_impl]` attribute which can be used to provide default implementations of various traits. While example contract implementations using this macro do exist, they only use the trait implementations generated for non-fungible tokens, leaving implementations generated for other traits, such as `FungibleToken`, untested and not used anywhere in the codebase.
- There are no tests covering all the execution paths of the `token_id_to_string` function. In particular, there is no test that executes this function with `value == 0`.

- The ledger sequence number is set to 0 by default during testing and the codebase contains the TTL extension logic which is dependent on the sequence number. As a result, some bugs may not be detected during testing as they would only manifest when the ledger sequence number is set to a more realistic value. For instance, the `revoke_approve_for_all_works` function tests the scenario where `live_until_ledger` is set to 0 in the `Base::approve_for_all` function, but the integer underflow is not detected by this test.

Consider expanding the testing suite by addressing the points mentioned above.

Update: *Resolved in pull request #191 at commit 1ed1035.*

M-02 Panic for Token ID 0 in `token_uri`

The `token_uri` functions implemented for the `Consecutive` and `Base` types can be used in order to return the URI of a token with a given `token_id`. Internally, both functions rely on the `compose_uri_for_token` function implemented for the `Base` type. The `compose_uri_for_token` function composes the URI based on the `base_uri` and the specified `token_id`. In order to convert `token_id` into a string, the `token_id_to_string` function is used.

However, in case `token_id` equals 0, the `token_id_to_string` function returns 0 as the number of digits, although the returned string "0" has a length of 1. As a result, the code will panic when attempting to copy the "0" string into the resulting slice due to the length mismatch. This will cause all `token_uri` calls specifying the `token_id` equal to 0 to panic.

Consider returning 1 as the number of digits in the `token_id_to_string` function in case the "0" string is returned.

Update: *Resolved in pull request #191 at commit 7c9a0f5.*

M-03 `owner_of` Function of `Consecutive` Extension may Exceed Resources

The `Consecutive` extension allows privileged entities to mint multiple tokens to a single owner in one transaction. In such a case, the target account is marked as the owner of the first minted token and it implicitly owns other tokens which have been minted in the batch. As a result, the owner of a given token may not be directly stored in storage and, hence, the `owner_of` function has to perform extra memory reads in order to determine the owner.

Currently, this is done inside the loop which iterates over the previous token IDs and returns the owner stored in the first existing entry.

However, this method of determining the owner is very inefficient and can easily exceed the resources allowed per transaction. In particular, the allowed number of ledger entry reads, which is currently set to 100, may be quickly exceeded. As a result, it may not be possible to determine the owner of some tokens. Thus, operations such as querying token URI, token approvals, transfers, and burns will not be available for some tokens minted in batch.

Consider implementing a more efficient method for determining the owner of a token and setting reasonable limits on the number of tokens that can be minted in a batch.

Update: *Resolved in pull request #213 at commit e690792.*

Low Severity

L-01 Ambiguity of the `Initial` Upgrade State

The default implementations of the `Upgradeable` and `Migratable` traits use the storage entry located under the `UPGRADE_KEY` key. This entry stores information regarding the upgrade status and can hold the values specified in the `UpgradeState` enum. However, the `get_upgrade_state` function, which returns the current upgrade state, returns the `Initial` value in two different cases: when the migration has already started and when the data located at `UPGRADE_KEY` equals `None`.

As a result, if a contract needs to determine whether the upgrade is in progress, the `get_upgrade_state` function cannot be used for this purpose as it can return the `Initial` state even if the upgrade has not started. Furthermore, it is possible to perform a migration even when no upgrade has taken place, which is not the desired behavior.

Consider introducing an additional state to the `UpgradeState` enum in order to represent the state that has not been initialized and returning it in the `get_upgrade_state` function if the upgrade state equals `None`.

Update: *Resolved in pull request #211 at commit ddaab82.*

L-02 Access Control Inside `Migratable` Is Not Explicit

The `Upgradeable` and `Migratable` traits may be used together in order to implement the upgradability logic in contracts. Both traits require an implementation of their internal counterparts: `UpgradeableInternal` and `MigratableInternal`, and these implementations should contain contract-specific logic, including a proper authorization mechanism.

However, while the `UpgradeableInternal` trait explicitly requires the authorization mechanism to be implemented through the `_upgrade_auth` function, the `MigratableInternal` trait does not provide a similar function. While the authorization logic can still be implemented inside the `_migrate` and `rollback` functions, having dedicated functions for this purpose would make the access control more explicit and would separate authorization logic from the migration-specific logic.

Consider adding dedicated functions for ensuring access control to the `MigratableInternal` trait, along with an additional argument for the `migrate` and `rollback` functions which will be used for authorization, similar to the `operator` argument of the `_upgrade_auth` function.

Update: *Resolved in pull request #211 at commit ddaab82.*

L-03 No Overflow Checks in `check_cap`

The `check_cap` function verifies that after minting the `amount` of tokens, the new total supply does not exceed the predefined cap amount.

However, the actual check does not verify whether the specified `amount` added to the total supply causes an overflow. As a result, if an overflow happens, the check may still succeed.

Consider performing an overflow check when comparing the new total supply with the cap amount in the `check_cap` function.

Update: *Resolved in pull request #186 at commit 9021508.*

L-04 Token Supply Cap May Be Lower Than Current Supply

The `set_cap` function allows for setting a maximum possible token supply. The function may be either called in the constructor or at any given time later on. However, while the function ensures that the newly specified cap is nonnegative, it does not perform a similar check in order to verify that the cap is not lower than the current total supply.

Consider verifying that the cap provided to the `set_cap` function is greater than or equal to the current token supply.

Update: *Resolved in pull request #200 at commit a936e08. A comment explaining the current design choice of not enforcing the cap to not be lower than the current total supply has been added.*

L-05 Typographical Errors

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

- In line 7, line 23, line 26, and line 27 of the `contract-utils/deafult-impl-macro/src/lib.rs` file, all instances of the word "Trait" could be written in lower case and whenever any of them directly reference the `Trait` trait from the given example, they should be put inside "``".
- In line 8 of the `contract-utils/deafult-impl-macro/src/lib.rs` file, "stellar" should be "Stellar".
- In line 26 of the `tokens/non-fungible/src/extensions/burnable/mod.rs` file, "NonFungibleBunrable" should be "NonFungibleBurnable".
- In line 63 of the `tokens/non-fungible/src/extensions/consecutive/mod.rs` file, "token" should be "tokens".
- In line 80 of the `tokens/non-fungible/src/non_fungible.rs` file, "Contract Type" should be "contract type" or "`ContractType`".
- In line 5 of the `tokens/non-fungible/src/overrides.rs` file, "Extension" should be written in lowercase.
- In line 18 of the `tokens/non-fungible/src/overrides.rs` file, "`NonFungibleTrait`" should be "`NonFungibleToken` trait".
- In line 26 of the `tokens/non-fungible/src/overrides.rs` file, "ContractType" should be "`ContractType`".
- In line 107 of the `tokens/non-fungible/src/storage.rs` file, "ApprovalData" should be "`ApprovalData`" and "Entry" should be "entry".

- In line 140 of the `tokens/non-fungible/src/storage.rs` file, "metadata", "base_uri", "name", and "symbol" should be put in backticks.
- In line 213 of the `tokens/non-fungible/src/storage.rs` file, "the" should be removed.
- In line 530 of the `tokens/non-fungible/src/storage.rs` file, "don't enough approval" should be "doesn't have enough approval".
- In line 92 of the `tokens/non-fungible/src/extensions/enumerable/mod.rs` file, "owner's local list" should be "global list".
- In line 207 and line 281 of the `tokens/non-fungible/src/extensions/enumerable/storage.rs` file, "increment" should be "decrement".

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

Update: *Resolved in pull request #187 at commit c66db7c.*

L-06 Insufficient `amount` Validation in `batch_mint`

The `batch_mint` function implemented for the `Consecutive` type allows for minting non-fungible tokens in batches. The desired number of tokens to mint should be specified in the `amount` parameter of this function. However, there is no validation ensuring that `amount > 0`. As a result, it is possible to assign ownership of a token that has not been minted as the code assumes that at least one token will be minted. Furthermore, in case the specified `amount` equals 0, an event containing the ID of a previously minted token will be emitted, and the old token ID will be returned.

Consider validating that the specified `amount` of tokens to mint is greater than 0.

Update: *Resolved in pull request #195 at commit 7bc4ca8.*

L-07 Lack of Necessary Derives for `NonFungibleTokenError`

The `NonFungibleTokenError` error enum defined in the `stellar-non-fungible` crate does not follow the requirements from the documentation. Particularly, the error definition does not use the `#[derive(Copy)]` attribute.

To follow best practices, consider implementing all the derive macros suggested by the docs when defining contract errors with the `#[contracterror]` attribute.

Update: *Resolved in pull request #185 at commit 63e9ffc.*

L-08 Performance Improvement Suggestions

Throughout the codebase, multiple opportunities for performance improvement were identified:

1. The `approve_for_all` function implemented for the `Base` type may be used to approve an operator to manage all tokens owned by the owner or to revoke such an approval. In order to store the operators approved for each owner, the `Map` type is used. In case a new operator is added, their address is inserted into the map, and when the approval is revoked for them, their entry is removed from the map. However, this mechanism is not optimal in terms of usage as it requires multiple storage reads when retrieving the elements, and inserting new elements may also incur additional cost. Furthermore, after each operation, the TTL is extended for the entire map as it is not possible to extend the TTL of each entry individually. As a result, it is possible that the map contains operators for whom the approval has already expired, which unnecessarily increases its size.

2. Furthermore, the `compose_uri_for_token` function checks the last character of the `base_uri` string and appends the `/` character at the end if it is not already present. This adds extra complexity to the code and this logic can be removed as it can be expected that base URIs will already contain the `/` character at the end as is done in OpenZeppelin's ERC-721 implementation in Solidity. This assumption should be included in the documentation.

3. Finally, the `set_owner_for` function continues execution even when a token with the given `token_id` has an owner, which will result in an additional, unnecessary storage read.

Consider storing approval information in a more efficient form, for example, by using the `(owner, operator)` tuple as a key and the `live_until_ledger` as the value, so that reading and modifying this information incurs lower costs and the entries expire by default for approvals which have already ended. Additionally, consider removing the logic related to appending `/` to base URI from the `compose_uri_for_token` function. Finally, consider returning early inside the `set_owner_for` function if a given token already has an owner.

Update: *Resolved in pull request #212 at commit 97cb96b.*

L-09 Global List Not Updated for Sequential Mints and Burns

The `Enumerable` extension allows users to access tokens owned by a specific account through their indexes in the owner's tokens list. A similar ability is provided for accessing tokens from the global list. However, in the case of sequential mints and burns, the global list is not updated with the reasoning that it is not needed as the tokens can be accessed through their IDs. A problem could arise when some tokens are burned. In such a case, a token with an ID equal to `n` would not be the `n`th entry in the global list anymore and could have any index from 0 to `n-2`, but this would not be recorded anywhere.

Consider updating the global tokens list inside sequential mint and burn functions so that the tokens can be listed even if the tokens with lower IDs have been burned.

Update: *Resolved in pull request #212 at commit 97cb96b.*

Notes & Additional Information

N-01 Misleading Documentation

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

- This comment states that the `operator` argument of the `_upgrade_auth` function can be a C-account or another contract. However, according to the Stellar documentation, "C...account" already relates to a contract account, hence, "G...account" should be included in the comment instead.
- This comment suggests that the `check_cap` function panics if the new `amount` of tokens exceeds the maximum token supply. However, the panic will happen if the new `amount` of tokens added to the current token supply exceeds the maximum supply.
- This comment implies that the `burn` function of the `NonFungibleBurnable` trait destroys the token with `token_id` from `account`. However, the `burn` function does not have the `account` parameter and the `from` parameter specifies the account from which the token with `token_id` should be destroyed. Furthermore, this comment could be rephrased so that it is clear that it refers to burning a token with a given `token_id`, and not destroying the `token_id`. The same suggestions also apply to the

documentation of other functions related to burning non-fungible tokens in the codebase.
- This comment specifies an incorrect value of `u128::MAX`.
- This comment refers to "`OwnedTokens`". However, the codebase does not use or reference this term anywhere else.

Consider addressing the above-mentioned instances of misleading documentation to improve the clarity and maintainability of the codebase.

Update: *Resolved in pull request #183 at commit 7ece61d.*

N-02 Code Duplication

The `check_cap` function verifies that after adding the `amount` of tokens to the current supply, the maximum allowed amount of tokens will not be exceeded. In order to get that maximum amount, it fetches this value from memory under the `CAP_KEY`. However, the exact same code is included in the `query_cap` function.

Consider using the `query_cap` function inside the `check_cap` function to determine the maximum possible token supply in order to reduce code duplication.

Update: *Resolved in pull request #182 at commit ad7f028.*

N-03 Argument Names Mismatch

The `impl_token_interface!` macro may be used in order to generate the implementation of the `TokenInterface` trait for the contracts that already implement the `FungibleToken` and `FungibleBurnable` traits.

However, the generated `allowance` function has the `owner` argument, which has a different name than the corresponding `from` argument specified in SEP-41. Similarly, the `owner` argument of the `approve` function has a different name than its counterpart from the `approve` function specified in the SEP-41.

Consider renaming the `owner` arguments of the generated `allowance` and `approve` functions so that they match their counterparts specified in the SEP-41 standard.

Update: *Resolved in pull request #181 at commit 0e00607.*

N-04 Rephrasing Suggestions

Throughout the codebase, multiple opportunities to rephrase comments to improve readability were identified:

- This comment could be rephrased so that the word "crate" is placed after "stellar_upgradeable". Furthermore, the quotes around the `stellar_upgradeable` crate name could be removed.
- This comment could be modified so that instead of saying "where can also be found" it says "where you can also find".
- This comment along with other similar comments referring to `token_id` could be rephrased to indicate that `token_id` refers to the ID of the token to be burned, not directly to the token.
- This comment could be rephrased so that "most likely" is moved to the beginning of the second sentence.

Consider modifying the comments mentioned above in order to improve the readability and clarity of the codebase.

Update: *Resolved in pull request #212 at commit 97cb96b.*

N-05 Rollback Logic in `Migratable` is Not Strictly Necessary

The `Migratable` trait specifies the `rollback` function which is meant to contain specific logic for when a rollback is needed after a contract upgrade. It is possible to use the default implementation of this function, but the users have to provide the implementation of the `_rollback` function from the `MigratableInternal` trait.

However, it is possible to include the entire rollback-specific logic in the `_migrate` function of the new contract implementation. In such a case, instead of first calling `rollback` and then upgrading a contract, it is possible to first upgrade it and then call the `migrate` function on the new implementation, hence, there is no need of requiring the `_rollback` function to be additionally implemented. Removing the rollback-specific logic from the `Migratable` and `MigratableInternal` traits has the additional benefit of simplifying the migration logic as it would only require holding two migration states.

Consider removing the explicit rollback logic from `Migratable` and `MigratableInternal` traits in order to make the code less error-prone and easier to use for the developers.

Update: *Resolved in pull request #211 at commit ddaab82.*

N-06 Insufficient Documentation

The codebase contains extensive documentation describing the usage and behavior of the implemented functions. However, some functions would benefit from additional inline documentation explaining their behavior in more detail and the documentation of some other functions could be expanded. Some examples are:

- The `token_id_to_string` function could include a comment explaining that the number 48 represents the '0' ASCII character.
- The `set_owner_for` function implemented for the `Consecutive` type is called for a token with `token_id + 1` when the `update` function is executed on a token having an ID of `token_id`. As a result, when a token with `token_id` is transferred or minted, the TTL of the storage entry containing the owner of the token with the `token_id + 1` ID is extended. This design decision could be explained in a comment.
- The `token_uri` functions implemented for the `Consecutive` type and for the `Base` type invoke the `owner_of` function and ignore the returned result. This call verifies that a token with the given ID exists, as otherwise, the `owner_of` function will panic. This could be explained in a comment since the reason for calling the `owner_of` function is not readily apparent.
- The `add_to_owner_enumeration` function implemented for the `Enumerable` type assumes that it is called after the balance has been manipulated. However, it is possible that if a token transfer has been made to the current owner, potentially through `transfer_from`, their balance will not change and the function will incorrectly override the tokens list for the owner. Although this situation cannot happen with the current implementations of the `transfer` and `transfer_from` functions for `Enumerable` because the code would first panic at the `remove_from_owner_enumeration` call, this edge case should nonetheless be included in the documentation so that developers using the codebase are aware of it and can avoid it when overriding `transfer` and `transfer_from` implementations.
- The documentation of the `non_sequential_mint` function implemented for the `Enumerable` type could include a warning similar to the one for the one from the `mint` function implemented for the `Base` type. A similar warning could be included in the documentation of the `sequential_mint` function implemented for the `Enumerable` type.
- The documentation for the `approve_for_all` function, the `approve_for_owner` function and the `approve` function could mention that the `live_until_ledger`

argument is implicitly bounded by the maximum allowed TTL extension for a temporary storage entry and specifying a higher value will cause the code to panic.

Consider adding more documentation in order to improve the clarity and understandability of the codebase.

Update: *Resolved in pull request #212 at commit 97cb96b.*

N-07 Refactoring Suggestions

The `token_uri` function implemented for the `Consecutive` type calls the `owner_of` function in order to validate that the given token exists. However, the returned value is not used later on, which means that the code which determines the owner is unnecessarily executed. A similar logic is present in the `token_uri` function implemented for the `Base` type, but the `owner_of` function implemented for that type consists of a very simple logic, so no redundant operations are executed in that case.

Consider refactoring the `owner_of` function of the `Consecutive` type so that the token existence validation is put into a helper function, which could then also be called inside the `token_uri` function.

Update: *Resolved in pull request #197 at commit 7ba6f21.*

N-08 No Explicit Approval-Revoking Ability

It is possible to revoke approval for all tokens for a given operator by calling the `approve_for_all` function with 0 supplied as the `live_until_ledger` argument. Similar logic related to revoking approvals is also present for the `approve` function of the fungible token implementation. However, it is not possible to revoke an approval this way for a non-fungible token, and in order to revoke an approval, users have to approve a token to themselves.

To be consistent with other functions in the codebase, consider allowing users to explicitly revoke NFT approvals by supplying 0 as the value for the `live_until_ledger` parameter of the `approve` function.

Update: *Resolved in pull request #174 at commit 407988b.*

Conclusion

The newly introduced extensions and utilities equip developers with tools that make contract development more efficient and flexible by enabling easy derivation of default function implementations.

During the audit, several design suggestions were provided to simplify the code and improve performance. Although the codebase includes a comprehensive testing suite, some issues were not detected due to incomplete test coverage—certain execution paths were untested, and default settings (e.g., a ledger sequence number of 0) were used in some cases. It is recommended to improve test coverage across all code branches and ensure that the testing environment closely mirrors the real blockchain environment.

The Soroban Contracts team was highly responsive throughout the audit, actively discussing design choices and thoroughly addressing all questions.