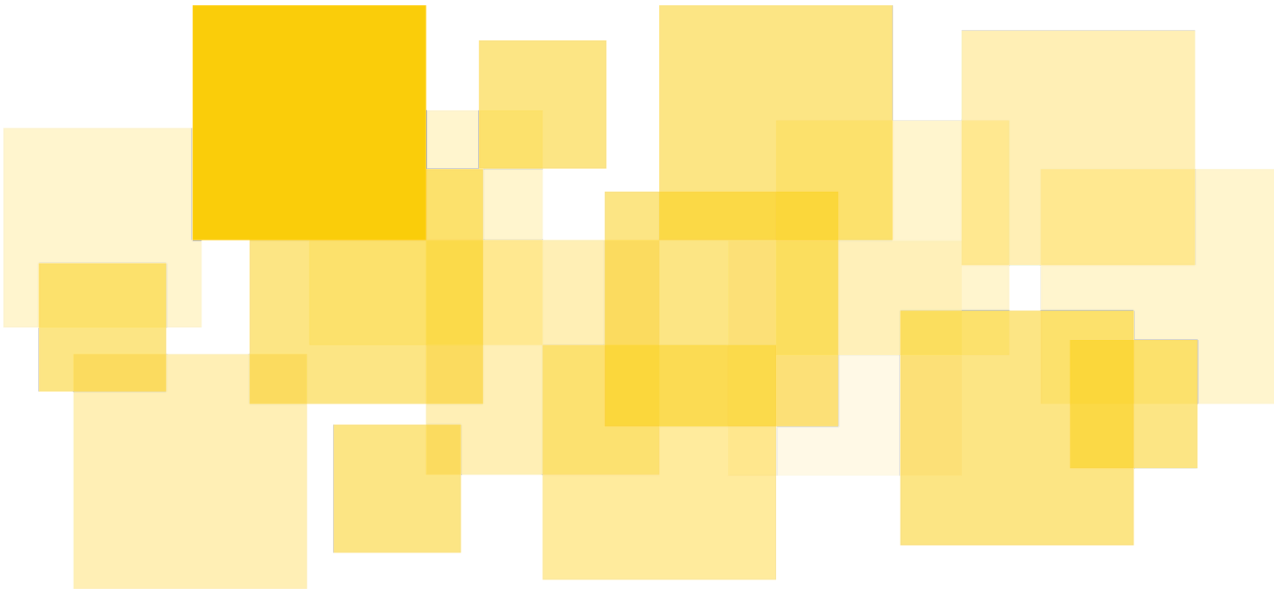
Security Audit Report

Matrixdock RWA Audit Stellar

Delivered: July 2, 2026



runtime verification

Table of Contents

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Executive Summary

Matrixdock engaged Runtime Verification Inc. to conduct a 2-week review of the Matrixdock Gold (XAUm) protocol and code, followed by 1 week of remediation support. The audit began on June 8, 2026, with the objective of evaluating the correctness and security of the protocol, reviewing existing quality assurance measures, identifying potential vulnerabilities, and providing recommendations for improvements in terms of code, testing, and documentation.

The audit's primary focus was on validating the role-based system that governs the XAUm token contract, with a secondary focus on the BullionMinter (minter) contract primarily in regard to how the two components interact. Each role in the protocol is granted specific capabilities, and ensuring that the bounds on each actor's actions are correctly managed is crucial to the protocol's proper functioning.

Runtime Verification's audit process began with a design review, where we familiarized ourselves with the protocol's role structure and contract architecture by reviewing documentation and the existing codebase, with the objective of identifying gaps in the role and permission model and providing recommendations on how to address them. Using the understanding developed during the design review, we conducted a manual code review of the contract implementation, comparing the implemented role capabilities and bounds against the intended design and security best practices. This was followed by a one-week period of remediation support, during which Runtime Verification worked with the Matrixdock team to address identified issues and verify proposed fixes.

Scope

The scope of this audit was limited to the code contained in the xaum-stellar directory of the public GitHub repository provided by the Matrixdock team, located at RWA-Contracts, with the fixed commit hash `458a5cbfcd65eef84685f27d02c6fddb0faf4946`. The repository was cloned at commit `458a5cbfcd65eef84685f27d02c6fddb0faf4946` on the main branch, which served as the initial revision under review. The elements within scope are divided into two core components, both implemented as Soroban smart contracts for the Stellar network:

- `./xaum-stellar/contracts/xaum`: Contains the implementation of the XAUm token contract, including its role-based access control system (Owner, Operator, Revoker, Pending Owner, Token Holder, Spender, and Blocked User roles), balance and allowance management, token metadata, minting and burning logic, the blocklist, timelocked governance actions, and the forced transfer (clawback) functionality. Approximately 1,522 lines of Rust across 8 source files, excluding tests.
- `./xaum-stellar/contracts/minter`: Contains the implementation of the BullionMinter contract, including the configuration of the minting and redeeming gateways, the Pool A and Pool B account destinations, and the Requestor-driven minting and redemption request flows. Approximately 627 lines of Rust across 6 source files, excluding tests.

Other implementations of the XAUm token on EVM-compatible chains (Ethereum, BNB Chain), the Matrixdock Gold NFT packing/unpacking functionality, cross-chain messaging infrastructure, frontend logic, deployment infrastructure, off-chain tooling, and third-party integrations (including KYC providers and gold reserve/allocation data feeds) are explicitly excluded from this engagement.

As findings were identified during the review, the client submitted several commits addressing reported issues. These remediation commits were also reviewed to verify that the identified issues were appropriately resolved prior to report finalization. The latest reviewed commit is `86fe22765f221922e67c0f48dfe2752c44d51493`.

Methodology

Runtime Verification followed a structured methodology to evaluate the correctness and security of the Matrixdock Gold (XAUm) smart contracts within the agreed two-week engagement, followed by a one-week remediation period. Although manual review cannot guarantee the discovery of all vulnerabilities, our approach was designed to maximize coverage and identify the most impactful risks within the available timeline.

Design Review and Role/Actor Mapping

The engagement began with documentation and scope intake, followed by a high-level design review of the XAUm protocol, focusing on the lifecycle of the xaum token contract and the minter (BullionMinter) contract, and on how the two components interact during minting and redemption.

The Matrixdock contracts use a role-based system to correctly execute their functionality and implement their business logic. Each role is enabled with specific capabilities in each of the contracts, and the correct management of bounds for actions of each actor within the protocol is crucial to its proper functioning. Aiming to validate this, we first built an understanding of the roles available in each contract.

Threat Analysis from the Design Review

Building on the role and invariant mapping, we conducted a dedicated threat analysis of the protocol. This included generating invariants for each privileged operation, running Almanax scans of the codebase using the context developed during the design review, and evaluating financial and business-logic risk, with particular attention to the protocol's logic, the implications of compromised or misused roles, and dependencies on external inputs such as the mint budget.

Manual Code Review

After establishing the protocol model, we performed a line-by-line review of the Soroban Rust codebase, covering both in-scope contracts. This included examining:

- Correctness of the xaum contract's privileged operations, including minting, burning, mint budget management, the blocklist, and the timelocked governance and clawback mechanisms;
- Correctness of the xaum contract's core accounting logic, including balances, allowances, and supply management;
- Soroban-specific concerns across both contracts, including storage layout, event emission, and the existing test suites;
- The minter (BullionMinter) contract's gateway configuration and the Pool A / Pool B minting and redemption request flows initiated by the Requestor role;
- Authorization and permission boundaries across both contracts, verified against the role and trust model identified during the design review;
- Edge cases, boundary conditions, and potential state inconsistencies.

Execution paths and state transitions were evaluated against the invariants identified earlier, including a final invariant check based on the findings of the low-level code review, to ensure alignment between the intended design and the implemented behavior.

Tool-Assisted Analysis

To supplement manual review, we incorporated targeted automated analysis using Almanax for AI-assisted bug pattern detection and anomaly surfacing across the codebase, with findings analyzed and triaged alongside the manual review results. Standard Rust tooling (including `cargo audit`, `cargo outdated`, and `cargo clippy`) was additionally run over the codebase to surface dependency and code-quality issues.

Finally, we conducted rounds of internal discussions with security experts over the code and platform design to verify possible exploitation vectors and identify improvements for the analyzed contracts.

Findings Consolidation and Reporting

In the final stage of the review, we consolidated all findings from the manual review, the threat analysis, and the Almanax scans, revisited any pending notes raised during earlier phases, and proofread the resulting report prior to delivery.

Roles Description and Analysis

The Matrixdock contracts use a role-based system to correctly execute their functionality and implement their business logic. Each role is enabled with specific capabilities in each of the contracts, and the correct management of bounds for actions of each actor within the protocol is crucial to its proper functioning. Aiming to validate this, we first have to understand what roles are available for each contract.

`xaum` token contract roles

- **Owner**: top governance authority, managing all other roles and contracts' lifecycle. Can transfer ownership, set and revoke other roles, control time delays, manage contract upgrades, and seize/clawback assets through a forced transfer functionality.
- **Operator**: minting/compliance authority. Can control the mint budget, partially manage the tokens in circulation through its mint and burn functionalities, and manage the tokens' blocklist.
- **Revoker**: a safety role that can cancel pending timelocked actions prior to their effect taking place. Cannot revoke actions directly related to itself or owner-controlled functionalities.
- **Pending Owner**: it is a transient actor, more than a proper role. It is the individual who must accept ownership of the contract in case it is transferred to them.
- **Token Holder**: another transient actor and, to all intents and purposes, is considered the end user of the platforms. It is the individual who sends, receives, and delegates token amounts to other entities' usage.
- **Spender**: transient actor capable of manipulating delegated tokens by a token holder.
- **Blocked users**: addresses flagged by the operator and barred from exercising the full activities of a token holder.

`minter` (BullionMinter) contract

- **Owner**: sole privileged role of the protocol. Configures the minting and redeeming gateways.
- **Pool Accounts (A and B)**: destination address for tokens used for minting (Pool A), and for redeeming (Pool B).
- **Requestor**: the `minter` 's user, who requests the minting and/or redeeming operations, incurring the authorization of transfers to the respective pools.

Trust/Authority Summary

Regarding the core and sensitive functions of the protocol, a summary of the trust and authorization permissions of non-transient actors per functionality can be seen in the table below:

Capability	Token Owner	Token Operator	Token Revoker	Minter Owner	Token Holder
Mint / change budget		✓			
Burn / redeem		✓			
Blocklist		✓			
Forced transfer (clawback)	✓				
Set operator/revoker, delays	✓				
Cancel pending mint/delay/operator			✓		
Upgrade contract	✓			✓	
Configure pools / accepted tokens				✓	

Business Logic and Expected Flow of Operations

To better understand the protocol's business logic, we need to define the expected flow of operations that correctly represent it. The expected actions can be broken down into:

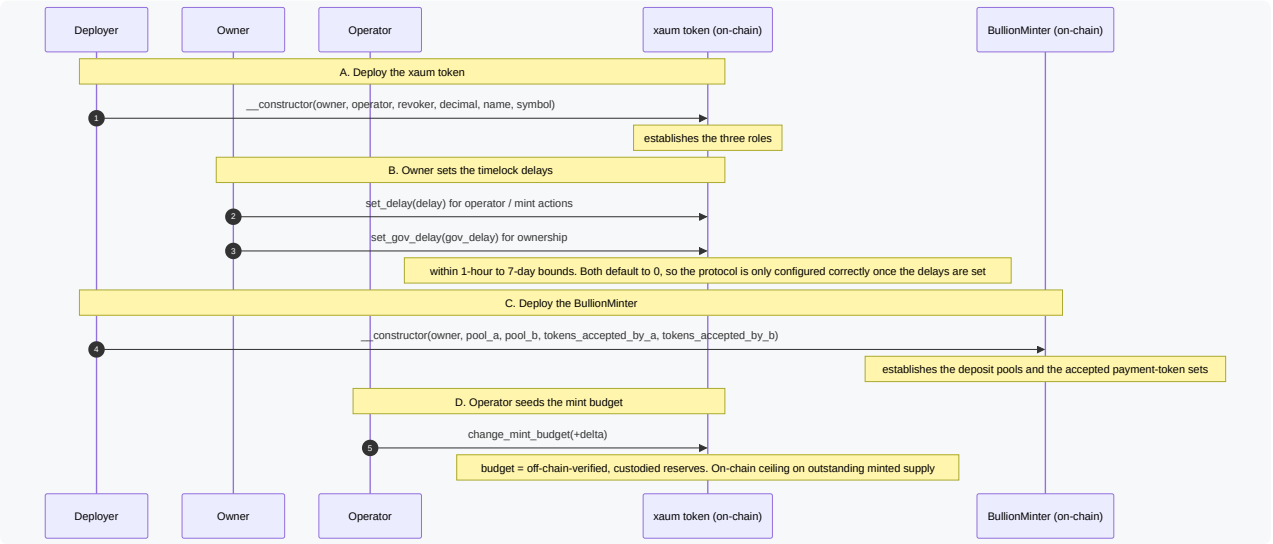
1. Setup & initialization (one-time, by deployer/owner)

A. Deploy the xaum token with `__constructor(owner, operator, revoker, decimal, name, symbol)`. This establishes the three roles.

B. Owner sets the timelock delays using `set_delay` for operator/mint actions and `set_gov_delay` for ownership, within the 1-hour to 7-day bounds. This step matters because both default to 0, so the protocol is only configured correctly once the delays are set.

C. Deploy the BullionMinter with `__constructor(owner, pool_a, pool_b, tokens_accepted_by_a, tokens_accepted_by_b)`, which establishes the deposit pools and the accepted payment-token sets.

D. The operator seeds the `mint_budget` via `change_mint_budget(+delta)`. The budget represents gold reserves that have been verified and custodied off-chain and are therefore authorized for minting. It acts as the on-chain ceiling on outstanding minted supply.



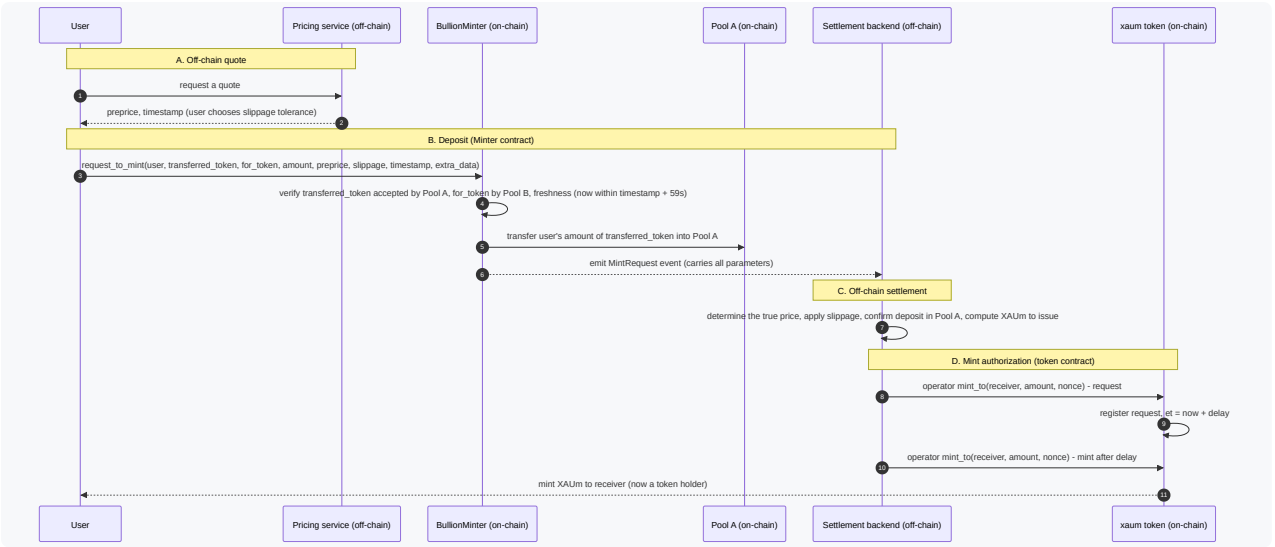
2. Mint flow (token holder acquires XAUm)

A. Off-chain quote. The user requests a quote from the pricing service and receives a preprice and a timestamp, plus a slippage tolerance they choose.

B. Deposit (Minter contract). User calls `request_to_mint(user, transferred_token, for_token, amount, preprice, slippage, timestamp, extra_data)`. The contract verifies `transferred_token` is accepted by Pool A and `for_token` by Pool B, checks the quote is fresh (`now <= timestamp + 59s`), transfers the user's amount of `transferred_token` into Pool A, and emits a `MintRequest` event carrying all the parameters.

C. Off-chain settlement. The backend observes the `MintRequest` event, determines the true price on its own, applies the user's slippage tolerance, confirms the deposit landed in Pool A, and computes the correct amount of XAUm to issue.

D. Mint authorization (token contract). The operator role calls `mint_to(receiver, amount, nonce)` twice to request and, after the expected delay, mint XAUm tokens, issuing them to a now token holder.



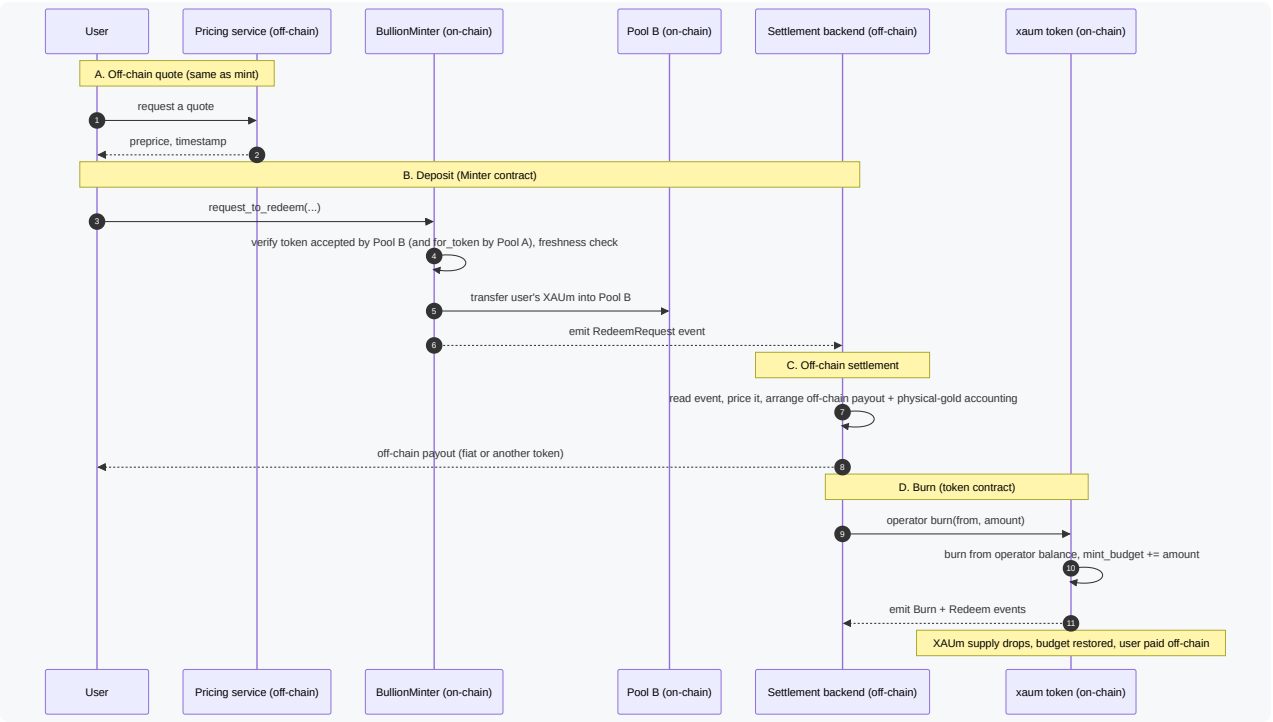
3. Redeem flow: user converts XAUm back

A. Off-chain quote, same as mint.

B. Deposit (Minter contract). User calls `request_to_redeem()`. The contract verifies the token is accepted by Pool B (and `for_token` by Pool A), checks freshness, transfers the user's XAUm into Pool B, and emits `RedeemRequest`.

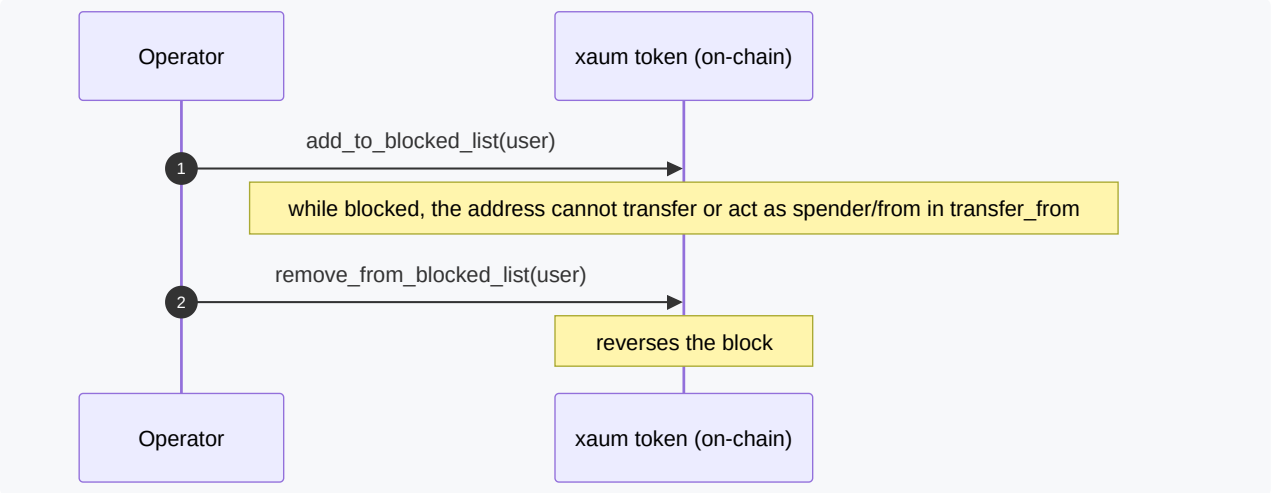
C. Off-chain settlement. The backend reads the event, prices it, and arranges the off-chain payout (fiat or another token) to the user along with the physical-gold accounting.

D. Burn (Token contract). The operator calls `burn(from, amount)`, which burns tokens from the operator's own balance (the XAUm that ended up under operator or pool control), credits `mint_budget` back by amount so the reserves are again available to mint, and emits Burn and Redeem events. The XAUm supply drops, the budget is restored, and the user is paid off-chain.

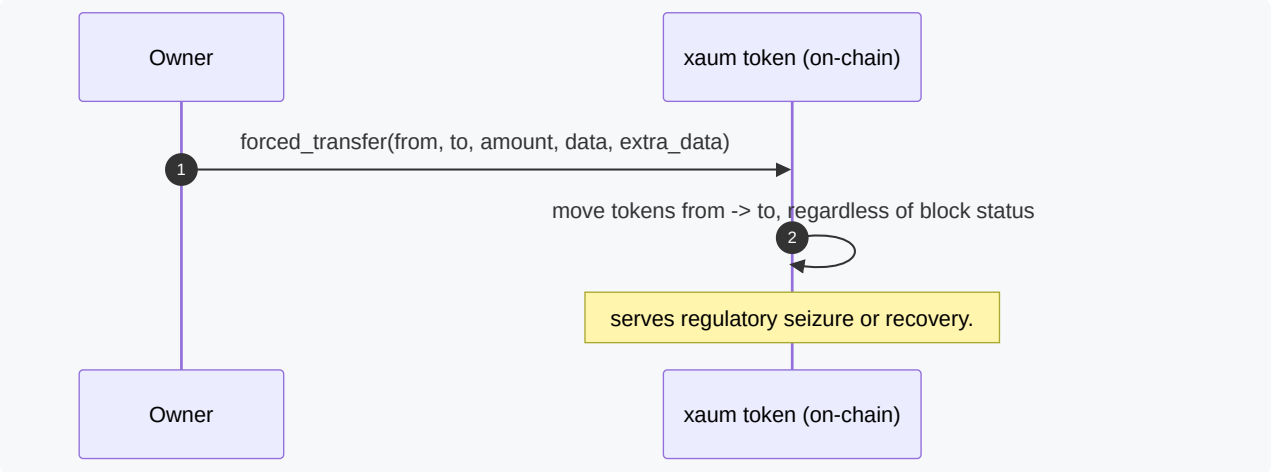


4. Compliance flows (operator/owner, as needed)

A. Blocklisting. When compliance flags an address, the operator calls `add_to_blocked_list(user)`. From then on, that address cannot transfer or act as a spender/from in `transfer_from`. `remove_from_blocked_list` reverses it.



B. Seizure/clawback. The owner calls `forced_transfer(from, to, amount, data, extra_data)` to move tokens regardless of block status. This serves regulatory seizure or recovery, with `data` / `extra_data` capturing the justification for audit trails.

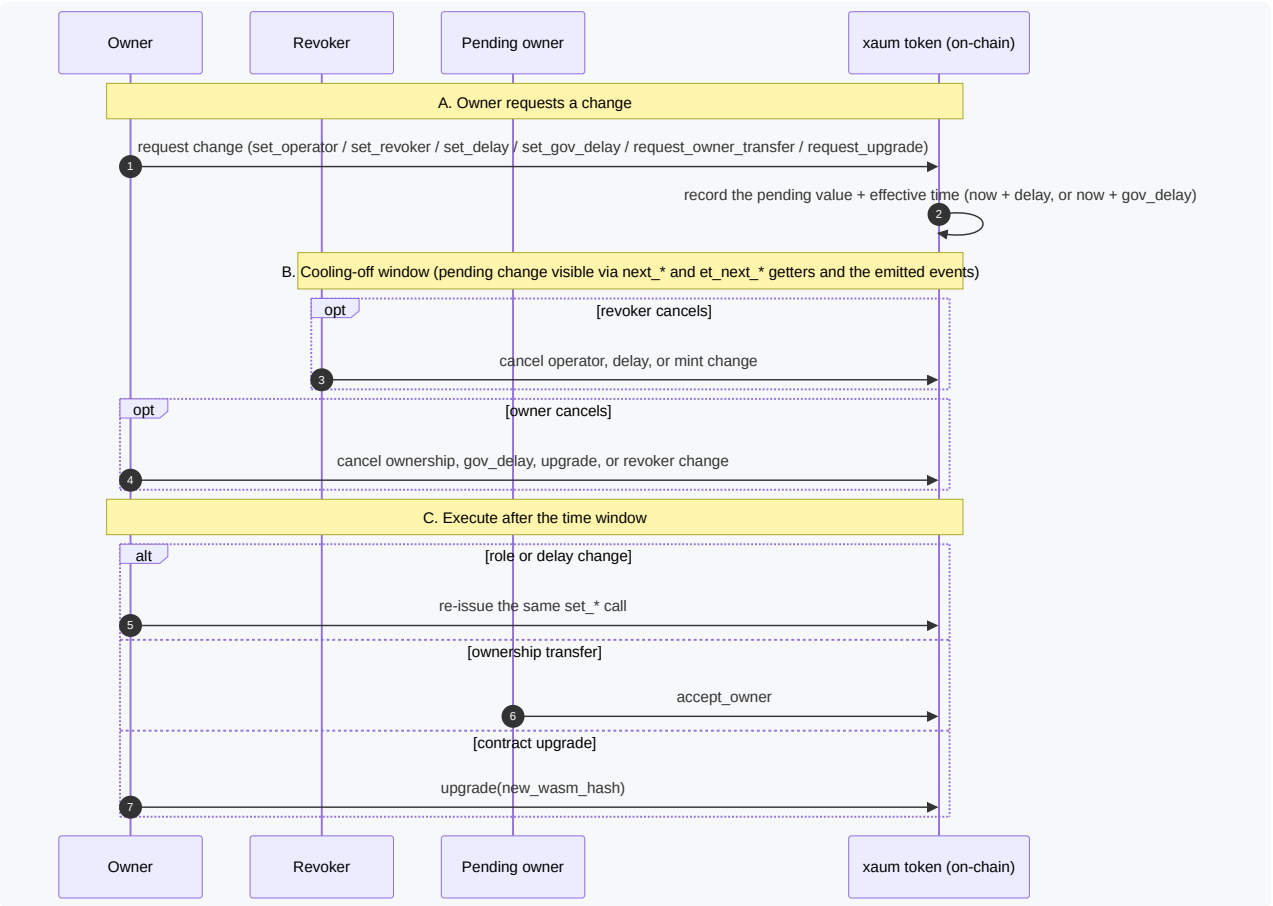


5. Governance flows (the timelock and revoker pattern)

A. Owner requests a change: new operator, new revoker, new delay, new `gov_delay`, ownership transfer, or contract upgrade. This records the pending value and an effective time (`now` + `delay` or `now` + `gov_delay`).

B. Cooling-off window. During the delay, observers can see the pending change through the `next_*` and `et_next_*` getters and the emitted events. The revoker can cancel operator, delay, and mint changes. The owner can cancel the governance-level ones: ownership, gov_delay, upgrade, and revoker.

C. On role and delay changes, after the time window is spent, the owner may re-issue the same `set_*` call. In case of ownership transfer, the pending owner may call accept_owner. For a contract upgrade, the owner calls `upgrade(new_wasm_hash)`, changing the contract WASM.



6. Ordinary token usage: Tokens can be used as any SEP-41 token independently of the mint and redeem machinery, with acknowledgment of their particularities in the form of features (blocklist, forced transfer).

Although not mentioned in the steps above, for time-locked, non-owner-exclusive operations in the Token Contract, the revoker can prevent them from coming to fruition by using its authority to essentially cancel updates to data it does not agree to.

Invariants

The following invariants capture the properties that must hold for the XAUm-Stellar contracts (token and minter) to behave correctly and safely. Each is a condition the protocol is expected to preserve at all times, regardless of the order or combination of operations, and together they define the boundary between intended behavior and a potential vulnerability. They are organized by domain (supply and balances, mint budget, access control, timelocks, compliance, and the minter), and serve as the reference against which the code is reviewed. Any execution path that can violate one of these properties is a finding.

Supply & balance accounting

- **I-1.** `total_supply` equals the sum of all account balances at all times.
- **I-2.** No balance is ever negative, and no transfer succeeds when `amount > balance(from)`.
- **I-3.** `total_supply` increases only via `mint_to` (second call) and decreases only via `burn`.
- **I-4.** Every amount entering any entry point is non-negative (`check_nonnegative_amount`).
- **I-5.** Arithmetic on balances, supply, and budget never wraps. Overflow must trap or error, never silently produce a wrong value.

Mint budget (reserve backing)

- **I-6.** `mint_budget >= 0` at all times.
- **I-7.** A mint of `amount` strictly decreases `mint_budget` by `amount` ; a burn of `amount` increases it by `amount`.
- **I-8.** No mint executes when `amount > mint_budget`.
- **I-9.** `mint_budget + total_supply` only changes via `change_mint_budget` (i.e., mint/burn conserve the sum, and only the operator's explicit budget change moves it).

Minting process

- **I-10.** No mint executes without a prior registered request for the exact same `(receiver, amount, nonce)` triple.
- **I-11.** A mint request executes only after its effective time has passed (`et < now`).
- **I-12.** A revoked mint request cannot be executed without being re-registered (and re-waiting the delay).
- **I-13.** Each `(receiver, amount, nonce)` request, once executed, is consumed.

Allowances

- **I-14.** `transfer_from` succeeds only if allowance ≥ amount, and decreases the allowance by exactly `amount`.
- **I-15.** An expired allowance (`expiration_ledger < current ledger`) spends as zero.
- **I-16.** A positive allowance can never be written with an expiration ledger in the past.

Access control

- **I-17.** Only the **operator** can: `mint_to` , `burn` , `change_mint_budget` , block/unblock addresses.
- **I-18.** Only the **owner** can: `forced_transfer` , set operator/revoker/delays, request/execute/revoke upgrades, request/revoke ownership transfer.
- **I-19.** Only the **revoker** can revoke pending mint requests, delay changes, and operator changes.
- **I-20.** Only the **pending owner** can `accept_owner` , and only after `gov_delay` has elapsed.
- **I-21.** `burn_from` always rejects (disabled).

Timelock & governance

- **I-22.** No privileged change (role, delay, upgrade, ownership) takes effect before `now > et`, where `et = request_time + delay` (or `gov_delay`).
- **I-23.** `MIN_DELAY (1h) <= delay, gov_delay <= MAX_DELAY (7d)` after any successful `set_*`.
- **I-24.** At most one pending request per category exists at a time.
- **I-25.** A pending request, once revoked, has no residual effect. Execution after revocation is impossible.
- **I-26.** The upgrade executed is byte-identical to the one requested (`new_wasm_hash` must match the stored hash).
- **I-27.** Every version of the contract retains the upgrade functions (else the contract may brick permanently).

Blocklist (compliance)

- **I-28.** A blocked address cannot be `from` in `transfer`, or `spender` / `from` in `transfer_from`.
- **I-29.** Only `forced_transfer` (owner) can move a blocked address's funds.
- **I-30.** Block/unblock is idempotent and takes effect immediately.

Minter (BullionMinter)

- **I-31.** `request_to_mint` accepts deposits only in tokens accepted by Pool A, targeting tokens accepted by Pool B (and the mirror for `request_to_redeem`).
- **I-32.** The user's deposit transfers to the configured pool **before** the request event is emitted; no event without a deposit.
- **I-33.** A request with a stale quote (`now > timestamp + 59s`) is rejected.
- **I-34.** Only the owner can change pools, accepted-token sets, or trigger upgrades.
- **I-35.** The minter never holds funds itself. Deposits go directly from the user to the pool.

Storage & liveness

- **I-36.** Role keys (`Owner`, `Operator`, `Revoker`) and balances always survive as instance/persistent storage. TTLs bumped on access.
- **I-37.** Pending-state TTLs always exceed the maximum delay window (e.g., `PENDING_TTL_LEDGERS` 10d > `MAX_DELAY` 7d), so a pending request cannot expire before it becomes executable.
- **I-38.** Paired pending keys (`NewWasmHash` + `EtNextUpgrade`, `PendingOwner` + `EtNextOwner`, etc.) are always written/removed together.

Events

- **I-39.** Every state-changing operation emits its corresponding event. No silent state changes.

Findings

This section contains all issues identified during the audit that could lead to unintended behavior, security vulnerabilities, or failure to enforce the protocol's intended logic. Each issue is documented with a description, potential impact, and recommended remediation steps.

[A1] Missing Delay Initialization in Constructor Bypasses Timelock

Severity: Low | Difficulty: High | Recommended Action: Fix Code | Acknowledged by client

Description

The XAUm token contract was designed to rely on two configurable timelock delays to protect sensitive operations: `delay` guards operator-level actions (e.g., minting, role changes) and `gov_delay` guards governance actions (e.g., ownership transfer). These delays force a mandatory waiting period between when a privileged action is requested and when it can be executed, giving the team a window to detect and cancel anything malicious or mistaken. The problem is that the constructor never initializes either value. Because the storage readers fall back to 0 when no value is present, both delays are effectively 0 from the moment the contract is deployed, meaning every "timelocked" action can actually be executed immediately.

This creates a dangerous window right after deployment, where the protections the design depends on are simply not in effect. Until the owner manually calls `set_delay` and `set_gov_delay`, anyone who controls (or compromises) the owner key can upgrade the contract, transfer ownership, or change roles with no cooling-off period and no opportunity for the revoker or the team to intervene. Worse, the setter functions are themselves time-locked using the current delay value, so the very first call to configure them is also unprotected. The protection only becomes real after a separate, manual, post-deployment step that is easy to forget or delay.

Recommendation

We recommend initializing both delays inside the `__constructor` itself, either as constructor parameters or as safe constants, and enforcing the intended bounds (`MIN_DELAY` of 1 hour through `MAX_DELAY` of 7 days) at that point. This guarantees the timelock is active from the very first moment the contract exists, removing the unprotected window entirely and eliminating the reliance on a manual follow-up step.

Status

The client has acknowledged this finding. Although in agreement with the recommendation above, their position is to make the owner's deployment and configuration of the smart contract more flexible, while maintaining the current initialization design.

[A2] Missing Nonce-Consumption Check Allows Mint Request Replay

Severity: Medium | Difficulty: High | Recommended Action: Fix Design | Partially addressed by client

Description

The `mint_to` function identifies each mint request by a hash of `(receiver, amount, nonce)`. The nonce exists to make each request unique and, by convention, single-use. However, the contract only stores a request while it is pending: once a mint executes, the request entry is deleted, and the contract keeps no record that the nonce was ever used. Because nothing marks a nonce as "spent," the exact same `(receiver, amount, nonce)` combination can be submitted again, re-registered, and minted a second time (and a third, and so on).

The practical impact is that nonces provide no replay protection. If the off-chain minting backend treats a nonce as a one-time identifier, which is the usual reason to include one, then any duplicated, retried, or replayed operator submission of the same request will mint the tokens again rather than being safely rejected as a duplicate. Each replay still requires operator authorization and available mint budget, so this is not an unauthorized-minting bug, but it does mean the on-chain contract does not enforce the one-mint-per-nonce guarantee the design appears to assume, delegating that responsibility entirely onto the off-chain system.

Recommendation

When security is concerned, contracts should be self-sufficient, with no delegation of responsibility to off-chain elements. To address this finding, persist a record of consumed nonces instead of removing all traces of a request after it executes. Concretely, when a mint completes, write a permanent "used" marker (for example, keyed by the `(receiver, amount, nonce)` hash, or by a per-receiver nonce value) to persistent storage, and have `mint_to` reject any request whose nonce is already marked used before registering or executing it. Alternatively, use a monotonically increasing nonce per receiver.

Status

The client has acknowledged this finding. The handling of nonces and prevention of exploits related to it will be handled in the off-chain settlement process.

[A3] Mint/Redeem Requests Are Not Validated and Offer No On-Chain Recourse

Severity: Low | Difficulty: Medium | Recommended Action: Fix Design | Partially addressed by client

Description

`request_to_mint` and `request_to_redeem` accept `preprice` and `slippage` as caller-supplied parameters but perform no assertions over them whatsoever, including basic sanity checks such as non-zero values or an upper bound on slippage. The functions validate only that the tokens are in the accepted sets and that the timestamp is not stale, and then immediately transfer the user's deposit into the pool. Because the deposit is moved before, and independently of, any settlement, and because settlement happens entirely off-chain (outside this audit's scope), the contract has no on-chain means of detecting or rejecting an incorrect, invalid, or unreasonable `preprice` / `slippage` pairing.

The consequence is borne by users who interact with the contract incorrectly. A user who submits malformed or unreasonable values has already irrevocably transferred their deposit token into the pool, yet will not receive their XAUm (mint) or payout (redeem) until and unless the off-chain settlement process resolves the request. The minter provides no on-chain cancellation, refund, or expiry path, so such a deposit is locked with no on-chain recourse, and its resolution depends entirely on off-chain behavior we cannot observe or guarantee. This represents a temporary loss of funds for users who interact with the minter contract using incorrect parameters.

References:

- `request_to_mint` transfers the deposit with no validation of `preprice` / `slippage`:

  Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/minter/src/contract.rs

  Line 197 to 207 in 458a5cb

```
197     pub fn request_to_mint(
198         env: Env,
199         user: Address,
200         transferred_token: Address,
201         for_token: Address,
202         amount: i128,
203         preprice: u128,
204         slippage: u128,
205         timestamp: u64,
206         extra_data: Bytes,
207     ) {
```

- `request_to_redeem`, same pattern:

Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/minter/src/contract.rs

Line 243 to 253 in 458a5cb

```
243     pub fn request_to_redeem(
244         env: Env,
245         user: Address,
246         transferred_token: Address,
247         for_token: Address,
248         amount: i128,
249         preprice: u128,
250         slippage: u128,
251         timestamp: u64,
252         extra_data: Bytes,
253     ) {
```

Recommendation

Evaluate adding on-chain sanity checks on the request parameters to reject obviously invalid input early (for example, require a non-zero `preprice` and bound `slippage` within a sensible maximum), so the most common user errors fail before any funds move. Additionally, offer a user the option to retract their deposit if their parameters are unreasonable.

Status

The client has acknowledged this finding. The behavior described above should manifest if users don't interact with the contract through the client's interface and invoke the contract directly; in this case, any potential damage can be considered self-inflicted. Additionally, the off-chain settlement mechanisms will handle such scenarios.

Informative Findings

This section includes observations that are not directly exploitable, but highlight areas for improvement in code clarity, maintainability, or best practices. While not critical, addressing these can strengthen the system's overall robustness.

[B1] Effective-time Readers Round-trip `Option<u64>` Through a 0 Sentinel

Severity: Informative | Recommended Action: Fix Code | Addressed by client

Description

Several effective-time readers in both `state.rs` files use `unwrap_or(0)` to fold "key absent" into `0`, then use `0` as a "no pending request" sentinel, conflating a zero value with a missing one. But `None` already expresses that, and the matching getters immediately convert the value back to `Option`: the code effectively round-trips `Option → 0 → Option`, which is a sign `Option<u64>` was the right type to begin with.

An example of this behavior:

Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/xaum/src/state.rs

Line 55 to 60 in 458a5cb

```rust
pub fn read_et_next_owner(env: &Env) -> u64 {
    env.storage()
        .temporary()
        .get(&DataKey::EtNextOwner)
        .unwrap_or(0)
}
```

And its getter, which undoes the sentinel:

Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/xaum/src/contract.rs

Line 530 to 535 in 458a5cb

```rust
    pub fn et_next_owner(env: Env) -> Option<u64> {
        match state::read_et_next_owner(&env) {
            0 => None,
            val => Some(val),
        }
    }
```

Recommendation

Have these readers return `Option<u64>` and drop the `unwrap_or(0)`. Call sites then can use `.is_some()` instead of `!= 0`, and the getters can return the reader directly.

Status

Fixed in https://github.com/Matrixdock-RWA/RWA-Contracts/commit/86fe22765f221922e67c0f48dfe2752c44d51493.

[B2] Pending Request Payload and Effective Time can Desync

Severity: Informative | Recommended Action: Fix Code | Acknowledged by client

Description

Pending governance and upgrade requests are stored as two independent storage keys that have to be kept in sync by hand: a payload (the new Wasm hash, the next owner/operator/revoker, the next delay) and its effective time `et_*`. They are set together, removed together, and TTL-extended together, but nothing enforces that they stay aligned. The invariant "the `et_*` key exists iff the payload key exists" lives only in convention.

Exemplifying the above, the upgrade hash and its effective time are separate slots, each with its own read/write/remove and TTL handling:

Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/xaum/src/state.rs

Line 331 to 337 in 458a5cb

```
331 pub fn write_next_upgrade_wasm_hash(env: &Env, new_wasm_hash: &BytesN<32>) {
332     let key = DataKey::NewWasmHash;
333     env.storage().temporary().set(&key, new_wasm_hash);
334     env.storage()
335         .temporary()
336         .extend_ttl(&key, PENDING_TTL_LEDGERS, PENDING_TTL_LEDGERS);
337 }
```

Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/xaum/src/state.rs

Line 350 to 356 in 458a5cb

```
350 pub fn write_et_next_upgrade(env: &Env, et: u64) {
351     let key = DataKey::EtNextUpgrade;
352     env.storage().temporary().set(&key, &et);
353     env.storage()
354         .temporary()
355         .extend_ttl(&key, PENDING_TTL_LEDGERS, PENDING_TTL_LEDGERS);
356 }
```

Because the two keys are separate, the consuming code checks the presence of one value and then plainly unwraps the other, relying on the hand-maintained invariant. The reader has to reason about why the `unwrap()` cannot panic, and the justification is only a comment:

Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/xaum/src/contract.rs

Line 184 to 188 in 458a5cb

```
184         let et = state::read_et_next_operator(&env); // u64, 0 = none
185
186         if et != 0 {
187             // next_operator always be Some if et != 0, no need to check
188             if next_operator.unwrap() != new_operator {
```

Recommendation

Bundling each pending request into a single storage slot holding a struct (e.g., `PendingUpgrade { wasm_hash, et }`) makes the two impossible to desync: one `set`, one `remove`, one TTL extension, payload, and `et` are always present or absent together. This applies across both `state.rs` modules.

Status

The client has acknowledged this finding. Precautions for preventing this issue from manifesting will be handled off-chain.

[B3] `MintBudget` and `TotalSupply` are lazily initialized, not set at construction

Severity: Informative | Recommended Action: Fix Code | Acknowledged by client

Description

`__constructor` never writes the instance slots `MintBudget` and `TotalSupply`. Each is created only on its first mutation, and the readers default an absent slot to zero via `unwrap_or(0)`.

This is correct (an absent slot and a stored `0` are observationally identical, and `0` is the right initial value), but it diverges from the other instance slots `Owner`, `Operator`, and `Revoker`, which `__constructor` always sets and reads with `unwrap()`. The contract thus carries two idioms for instance state and an absent-vs-`0` distinction that readers must track.

Recommendation

Initialize both slots to `0` in `__constructor` and switch the readers to `unwrap()`. Since `0` is in range for both quantities, this is behavior-preserving: every instance slot becomes eagerly set, and the reader idiom is unified. Cost is one extra write per slot at construction, which is negligible.

Status

The client has acknowledged this finding.

[B4] Revoker Cannot Protect the Protocol From Incorrect Owner Updates

Severity: Informative | Recommended Action: Fix Code | Acknowledged by client

Description

As previously explored in the Roles Description and Analysis section, the Revoker is the protocol's circuit-breaker for pending time-locked actions: during the cooling-off window, it can cancel a pending operator change (`revoke_next_operator`), a pending delay change (`revoke_next_delay`), and a pending mint request (`revoke_mint_request`). The contract upgrade, however, is excluded from the revoker's authority. `revoke_next_upgrade` requires owner authorization, so the only actor that can abort a pending upgrade is the same owner that requested it. This leaves the single most powerful operation in the system, one capable of replacing the entire contract logic, including the role model and the timelock mechanism itself, without an independent party able to stop it during its delay window.

Differential analysis against the EVM reference implementation (out of scope, used here only as evidence of the intended design) shows that `revokeNextUpgrade` is gated `onlyRevoker` , i.e., the revoker is explicitly the role meant to abort a pending upgrade. The Stellar port keeps the upgrade on the operational `delay` (consistent with the reference) but removes the revoker's power over it, so the cooling-off window for upgrades has no independent circuit-breaker. The practical consequence is that if the owner key is compromised or misused to queue a malicious upgrade, the revoker, the role designed to intervene in exactly this situation, cannot act, and the entire defense against a bad upgrade collapses onto the owner key that initiated it. This is a weaker safety posture than the reference design it is based on.

Similarly, `revoke_next_owner` , which cancels a pending ownership transfer during its `gov_delay` cooling-off window, is gated to the owner (`read_owner().require_auth()`). Regarding authorization, the same reasoning applies to this function as well.

Such behavior can also be identified in the owner-gated operations of equal functionality in the XAUm token.

References:

- Stellar, `revoke_next_upgrade` gated to the owner:

  Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/xaum/src/contract.rs

  Line 100 to 104 in 458a5cb

```
100     pub fn revoke_next_upgrade(env: Env) {
101         let owner = state::read_owner(&env);
102         owner.require_auth();
103
104         bump_instance(&env);
```

- EVM reference, `revokeNextUpgrade` gated to the revoker:

  Matrixdock-RWA/RWA-Contracts/xaum-evm/contracts/Delayable.sol

  Line 62 to 65 in 458a5cb

```
62     function revokeNextUpgrade() public onlyRevoker {
63         etNextUpgradeToAndCall = 0;
64         emit NextUpgradeRevoked(nextUpgradeToAndCallDataHash);
65     }
```

- Stellar, `revoke_next_owner` gated to the owner:

Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/xaum/src/contract.rs

Line 164 to 167 in 458a5cb

```rust
    pub fn revoke_next_owner(env: Env) {
        let owner = state::read_owner(&env);
        owner.require_auth();

```

- EVM reference, `revokeOwnershipTransfer` is also `onlyOwner`:

Matrixdock-RWA/RWA-Contracts/xaum-evm/contracts/Ownable2StepTimeLockUpgradeable.sol

Line 68 to 74 in 458a5cb

```solidity
    function revokeOwnershipTransfer() public virtual onlyOwner {
        Ownable2StepTimeLockStorage storage $ = _getOwnable2StepTimeLockStorage();
        address pending = $._pendingOwner;
        delete $._pendingOwner;
        delete $._etNextOwner;
        emit OwnershipTransferRevoked(pending);
    }
```

Recommendation

Grant the revoker authority to cancel a pending upgrade by changing `revoke_next_upgrade` to require revoker authorization, aligning the Stellar implementation with the EVM reference and with the revoker's existing power over the other time-locked, non-owner-exclusive actions. The same should be considered on the `revoke_next_owner` function.

Status

This finding has been acknowledged by the client, and the elaborated behavior is intended by design.

[B5] `TokenError` variants `NotOwner`, `NotOperator`, `NotRevoker` defined but never raised

Severity: Informative | Recommended Action: Fix Code | Acknowledged by client

Description

The `TokenError` variants `NotOwner` (10), `NotOperator` (11), and `NotRevoker` (12) are defined but never raised.

Role gating is enforced via `require_auth` on the role address (`read_<role>()` + `require_auth()`), so unauthorized callers fail with a host authorization error, not these custom errors.

An example:

Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/xaum/src/contract.rs

Line 426 to 427 in 458a5cb

```
426         let operator = state::read_operator(&env);
427         operator.require_auth();
```

The three variants thus only appear in the `enum` definition:

Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/xaum/src/error.rs

Line 10 to 12 in 458a5cb

```
10      NotOwner = 10,
11      NotOperator = 11,
12      NotRevoker = 12,
```

Recommendation

Remove the unused variants, or mark them as currently unraised if kept for ABI stability.

Status

The client has acknowledged this finding.

[B6] One-Sided Freshness Check Accepts Arbitrarily Future-Dated Timestamps

Severity: Informative | Recommended Action: Fix Code | Acknowledged by client

Description

`request_to_mint` and `request_to_redeem` are meant to reject stale requests, but the freshness check is one-sided. It only enforces `now > timestamp + DELAY_MAX` (with `DELAY_MAX = 59` seconds), which bounds how far in the *past* the supplied `timestamp` may be and does nothing to bound how far in the future it may be. As a result, a caller can submit a request with a `timestamp` set arbitrarily far ahead (for example, ten years from now) and it will pass the check indefinitely, defeating the intended freshness guard. The practical impact is limited because the `timestamp` is self-reported by the caller in the first place (a caller can always pass `timestamp = now`).

Compounding the limited usefulness of the check, the validated `timestamp` is not included in the emitted `MintRequest` / `RedeemRequest` event. Off-chain services that consume these events, therefore, cannot perform their own freshness validation against the value the contract checked, so the on-chain check neither enforces a meaningful bound on-chain nor propagates the timestamp for off-chain enforcement. This part is mostly informative, but it means the freshness mechanism provides little real protection in either domain.

References:

- One-sided freshness check in `request_to_mint`:

  Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/minter/src/contract.rs

  Line 221 to 224 in 458a5cb

```
221         let now = env.ledger().timestamp();
222         if now > timestamp + DELAY_MAX {
223             panic_with_error!(env, MinterError::InvalidTimestamp);
224         }
```

- Same check in `request_to_redeem`:

  Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/minter/src/contract.rs

  Line 267 to 270 in 458a5cb

```
267         let now = env.ledger().timestamp();
268         if now > timestamp + DELAY_MAX {
269             panic_with_error!(env, MinterError::InvalidTimestamp);
270         }
```

- `MintRequest` event emitted without the `timestamp` field:

Matrixdock-RWA/RWA-Contracts/xaum-stellar/contracts/minter/src/contract.rs

Line 230 to 240 in 458a5cb

```
230         MintRequest {
231             transferred_token,
232             for_token,
233             requestor: user,
234             pool,
235             amount,
236             preprice,
237             slippage,
238             extra_data,
239         }
240         .publish(&env);
```

Recommendation

Make the check two-sided, rejecting both stale and future-dated timestamps (for example, require `timestamp <= now` and `now - timestamp <= DELAY_MAX`, optionally allowing a small clock-skew margin in the future direction), and include the `timestamp` in the emitted events so off-chain consumers can apply their own validation. More fundamentally, because the timestamp is supplied by the caller and unauthenticated, the freshness guard provides weak assurance regardless of the bounds.

Status

The client has acknowledged this finding. The timestamp is not used off-chain; there's no need currently to add it to broadcasted events. Additionally, off-chain validations during the settlement process will add security layers to the management of tokens handled by the contracts in scope.

Appendix

The following appendix provides reference documentation for the two Soroban contracts under investigation, inferred as a by-product of the audit process. For both contracts, we document their storage and functions, along with assumptions, invariants, function postconditions, and supporting proofs.

The descriptions are based on commit https://github.com/Matrixdock-RWA/RWA-Contracts/tree/458a5cbfcd65eef84685f27d02c6fddb0faf4946, and describe the behavior of that revision only.

`BullionMinter` Contract Overview

On-chain request gateway for minting and redeeming a tokenized asset, with fulfillment performed off-chain. Users call `request_to_mint` or `request_to_redeem`, which validate the input and target tokens against the per-pool accepted-token sets, transfer the input tokens into the relevant pool account, and emit a `MintRequest` / `RedeemRequest` event that an off-chain operator acts on; the contract itself never mints, burns, or holds asset accounting. It maintains two pool accounts (`PoolAccountA` for minting inputs, `PoolAccountB` for redeeming inputs) and the corresponding accepted-token sets (`AcceptedByA`, `AcceptedByB`), all owner-configurable. A single owner (`Owner`) governs configuration, and both ownership transfers and contract upgrades are time-locked behind a `DELAY_SETTING` delay with explicit request / accept-or-execute / revoke steps.

Invariants

Name	Property	Proof
`owner_always_set`	`Owner` is always set.	Established by `__constructor`, which always sets `Owner`. `accept_owner` overwrites it (with a set `PendingOwner`) but never unsets it. No other function mutates the slot.
`pool_account_a_always_set`	`PoolAccountA` is always set.	Established by `__constructor`, which always sets `PoolAccountA`. `set_pool_account_a` overwrites it but never unsets it. No other function mutates the slot.
`pool_account_b_always_set`	`PoolAccountB` is always set.	Established by `__constructor`, which always sets `PoolAccountB`. `set_pool_account_b` overwrites it but never unsets it. No other function mutates the slot.
`pending_upgrade_consistent`	`NewWasmHash` and `EtNextUpgrade` are either both set or both unset.	Established by `request_upgrade`, which always sets both. Preserved by `upgrade` and `revoke_next_upgrade`, which always unset both together. No other function mutates either slot. Both slots are temporary entries, always written in the same transaction with the same TTL (`PENDING_TTL_LEDGERS`), so they expire together.
`pending_owner_consistent`	`PendingOwner` and `EtNextOwner` are either both set or both unset.	Established by `request_owner_transfer`, which always sets both. Preserved by `accept_owner` and `revoke_next_owner`, which always unset both together. No other function mutates either slot. Both slots are temporary entries, always written in the same transaction with the same TTL (`PENDING_TTL_LEDGERS`), so they expire together.
`et_next_upgrade_nonzero_when_set`	When `EtNextUpgrade` is set, its value is non-zero; the zero value is reserved as the "absent" sentinel and is never stored.	The sole writer of `EtNextUpgrade` is `request_upgrade`, which stores `now + DELAY_SETTING`. As `DELAY_SETTING` is `43200` and `now` is non-negative, the stored value is at least `43200`, hence non-zero, by `assume_no_overflow`.
`et_next_owner_nonzero_when_set`	When `EtNextOwner` is set, its value is non-zero; the zero value is reserved as the "absent" sentinel and is never stored.	The sole writer of `EtNextOwner` is `request_owner_transfer`, which stores `now + DELAY_SETTING`. As `DELAY_SETTING` is `43200` and `now` is non-negative, the stored value is at least `43200`, hence non-zero, by `assume_no_overflow`.
`pending_upgrade_deadline_immutable`	While an upgrade is pending, its effective time `EtNextUpgrade` is never modified in place: between the request that sets it and the execute or revoke that clears it, the value is constant.	`EtNextUpgrade` is set only by `request_upgrade` and removed only by `upgrade` and `revoke_next_upgrade`; no function overwrites an already-set slot. A repeat request is rejected by the `PendingRequestExists` guard, which detects the pending request because `et_next_upgrade_nonzero_when_set` makes a set slot read as non-zero. So changing the deadline requires revoking and re-requesting, and the time-lock fixed at request time cannot be shortened or extended after the fact.
`pending_owner_deadline_immutable`	While an ownership transfer is pending, its effective time `EtNextOwner` is never modified in place: between the request that sets it and the accept or revoke that clears it, the value is constant.	`EtNextOwner` is set only by `request_owner_transfer` and removed only by `accept_owner` and `revoke_next_owner`; no function overwrites an already-set slot. A repeat request is rejected by the `PendingRequestExists` guard, which detects the pending request because `et_next_owner_nonzero_when_set` makes a set slot read as non-zero. So changing the deadline requires revoking and re-requesting, and the time-lock fixed at request time cannot be shortened or extended after the fact.

Constants

Name	Type	Value	Description
`DELAY_MAX`	`u64`	`59`	Maximum allowed staleness, in seconds, of the caller-supplied `timestamp` and `timestamp` in `request_to_mint` and `request_to_redeem`, respectively. A request is rejected with `InvalidTimestamp` if the current ledger time exceeds the timestamp by more than this.
`DELAY_SETTING`	`u64`	`12 * 3600`	Duration of the time-lock, in seconds (12 hours). When an upgrade or ownership transfer is requested, the effective time is set to `now + DELAY_SETTING`; the request cannot be executed until the ledger time passes that point. Applies to `request_upgrade` (via `EtNextUpgrade`) and `request_owner_transfer` (via `EtNextOwner`).
`DAY_IN_LEDGERS`	`u32`	`17280`	Approximate number of ledgers produced in one day, assuming a ledger roughly every 5 seconds (86400 / 5 = 17280). Used as the base unit for the TTL constants `PENDING_TTL_LEDGERS`, `INSTANCE_BUMP_AMOUNT`, and `INSTANCE_LIFETIME_THRESHOLD`.
`PENDING_TTL_LEDGERS`	`u32`	`3 * DAY_IN_LEDGERS`	TTL, in ledgers, applied to the temporary storage slots backing a pending request: `PendingOwner`, `EtNextOwner`, `NewWasmHash`, and `EtNextUpgrade`. Each is written with both the threshold and extend-to arguments set to this value. At ~3 days it sits well above the 12-hour `DELAY_SETTING` time-lock, so a pending request cannot expire before it becomes executable.
`INSTANCE_BUMP_AMOUNT`	`u32`	`30 * DAY_IN_LEDGERS`	Target TTL, in ledgers (~30 days), to which the instance storage TTL is extended. Passed as the extend-to argument of `extend_ttl`, paired with the threshold `INSTANCE_LIFETIME_THRESHOLD`: whenever a state-changing function bumps instance TTL and it has dropped to or below that threshold, it is bumped back up to this value.
`INSTANCE_LIFETIME_THRESHOLD`	`u32`	`INSTANCE_BUMP_AMOUNT - DAY_IN_LEDGERS`	Threshold TTL, in ledgers, that triggers an instance-storage bump. Passed as the threshold argument of `extend_ttl`, paired with the extend-to value `INSTANCE_BUMP_AMOUNT`: when the instance TTL has dropped to or below this value, it is extended back to `INSTANCE_BUMP_AMOUNT`. Set one day below the bump target, so a bump only occurs once the TTL has decayed by at least a day, rather than on every call.

Storage

Name	Type	Lifetime	Description
`Owner`	`Address`	`instance`	Address of the current contract owner. The owner is the primary privileged role: it configures the pool accounts (`set_pool_account_a`, `set_pool_account_b`) and the accepted-token sets (`set_accepted_by_a`, `set_accepted_by_b`), and it initiates and revokes the time-locked contract upgrades (`request_upgrade`, `revoke_next_upgrade`, then `upgrade`) and ownership transfers (`request_owner_transfer`, `revoke_next_owner`).
`PendingOwner`	`Address`	`temporary`	Address of the requested new owner, set during a pending ownership transfer. This role has a single privilege: once the time-lock has elapsed, it executes the transfer by calling `accept_owner` (the current `Owner` cannot execute it), at which point it becomes the new owner.
`EtNextOwner`	`u64`	`temporary`	Effective time (Unix seconds) after which the pending ownership transfer may be executed; the time-lock on ownership transfers. Set to `now + DELAY_SETTING` by `request_owner_transfer`; until the ledger time passes it, `accept_owner` reverts with `TooEarlyToExecute`.
`NewWasmHash`	`BytesN<32>`	`temporary`	Wasm hash of the pending upgrade request.
`EtNextUpgrade`	`u64`	`temporary`	Effective time (Unix seconds) after which the pending upgrade may be executed; the time-lock on contract upgrades. Set to `now + DELAY_SETTING` by `request_upgrade`; until the ledger time passes it, `upgrade` reverts with `TooEarlyToExecute`.
`PoolAccountA`	`Address`	`instance`	Address of pool A, which accepts tokens for minting.
`PoolAccountB`	`Address`	`instance`	Address of pool B, which accepts tokens for redeeming.
`AcceptedByA`	`Set<Address>`	`instance`	Set of token addresses accepted by pool A as input for minting. Note that in this description, `Set` is an abstraction: the actual implementation maps keys of the form `AcceptedByA(Address)` to `()`, where membership is the existence of the entry.
`AcceptedByB`	`Set<Address>`	`instance`	Set of token addresses accepted by pool B as input for redeeming. Note that in this description, `Set` is an abstraction: the actual implementation maps keys of the form `AcceptedByB(Address)` to `()`, where membership is the existence of the entry.

Events

Event	Fields	Description
`UpgradeRequested`	`owner: Address` — The current owner, who authorized the request. `new_wasm_hash: BytesN<32>` — Wasm hash the contract is requested to upgrade to. `effective_time: u64` — Earliest time the upgrade may be executed (Unix timestamp, seconds).	Emitted by `request_upgrade` when a time-locked upgrade is requested.
`ContractUpgraded`	`owner: Address` — The current owner, who authorized the upgrade. `new_wasm_hash: BytesN<32>` — Wasm hash the contract was upgraded to.	Emitted by `upgrade` when a pending time-locked upgrade is executed.
`UpgradeRevoked`	`owner: Address` — The current owner, who authorized the revocation. `new_wasm_hash: BytesN<32>` — Wasm hash of the upgrade that was cancelled.	Emitted by `revoke_next_upgrade` when a pending upgrade is cancelled.
`OwnerTransferRequested`	`owner: Address` — The current owner, who authorized the request. `pending_owner: Address` — The requested new owner. `effective_time: u64` — Earliest time the transfer may be executed (Unix timestamp, seconds).	Emitted by `request_owner_transfer` when a time-locked ownership transfer is requested.
`OwnerTransferred`	`old_owner: Address` — The owner prior to the transfer. `new_owner: Address` — The owner after the transfer, who authorized it.	Emitted by `accept_owner` when a pending ownership transfer is executed.
`OwnerRevoked`	—	Emitted by `revoke_next_owner` when a pending ownership transfer is cancelled.
`SetPoolAccountA`	`pool: Address` — The new address of pool A.	Emitted by `set_pool_account_a` when the address of pool A is set.
`SetPoolAccountB`	`pool: Address` — The new address of pool B.	Emitted by `set_pool_account_b` when the address of pool B is set.
`SetAcceptedByA`	`token: Address` *(topic)* — The token that was added or removed. `accepted: bool` — `true` if the token was added to the set, `false` if removed.	Emitted by `set_accepted_by_a` when a token is added to or removed from the set accepted by pool A.
`SetAcceptedByB`	`token: Address` *(topic)* — The token that was added or removed. `accepted: bool` — `true` if the token was added to the set, `false` if removed.	Emitted by `set_accepted_by_b` when a token is added to or removed from the set accepted by pool B.
`MintRequest`	`transferred_token: Address` *(topic)* — The token the user transferred in as input for minting. `for_token: Address` *(topic)* — The token the user wishes to mint. `requestor: Address` *(topic)* — The user who submitted the request and authorized the transfer. `pool: Address` — The address of pool A, which received the transferred tokens. `amount: i128` — The amount of `transferred_token` transferred. `preprice: u128` — Caller-supplied reference price for the request. `slippage: u128` — Caller-supplied slippage tolerance for the request. `extra_data: Bytes` — Caller-supplied opaque data passed through with the request.	Emitted by `request_to_mint` when a user successfully submits a mint request, after the input tokens have been transferred to pool A.

`RedeemRequest`	`transferred_token: Address` *(topic)* — The token the user transferred in as input for redeeming. `for_token: Address` *(topic)* — The token the user wishes to redeem for. `requestor: Address` *(topic)* — The user who submitted the request and authorized the transfer. `pool: Address` — The address of pool B, which received the transferred tokens. `amount: i128` — The amount of `transferred_token` transferred. `preprice: u128` — Caller-supplied reference price for the request. `slippage: u128` — Caller-supplied slippage tolerance for the request. `extra_data: Bytes` — Caller-supplied opaque data passed through with the request.	Emitted by `request_to_redeem` when a user successfully submits a redeem request, after the input tokens have been transferred to pool B.

Errors

Error	Code	Description
`NegativeAmountNotAllowed`	`1`	Raised by `request_to_mint` and `request_to_redeem` when the caller-supplied `amount`, respectively `amount` is negative.
`NoPendingOwner`	`10`	Raised by `accept_owner` when there is no pending ownership transfer to accept.
`InvalidTokenForMinting`	`20`	Raised by `request_to_mint` when the transferred-in token is not accepted by pool A as input for minting.
`InvalidTokenForRedeeming`	`21`	Raised by `request_to_redeem` when the transferred-in token is not accepted by pool B as input for redeeming.
`InvalidForToken`	`22`	Raised by `request_to_mint` when the `for_token` (the token to be minted) is not accepted by pool B, and by `request_to_redeem` when the `for_token` (the token to be redeemed for) is not accepted by pool A.
`InvalidTimestamp`	`30`	Raised by `request_to_mint` or `request_to_redeem` when the caller-supplied `timestamp`, respectively `timestamp` is too stale: the current ledger time exceeds it by more than `DELAY_MAX` seconds.
`TooEarlyToExecute`	`31`	Raised by `upgrade` or `accept_owner` when the time-lock has not yet elapsed: the current ledger time has not passed the effective time recorded when the request was made.
`PendingRequestExists`	`32`	Raised by `request_upgrade` or `request_owner_transfer` when a request of that kind is already pending; the existing one must be executed or revoked before a new one can be made.
`InvalidWasmHash`	`40`	Raised by `upgrade` when the supplied `new_wasm_hash` does not match the Wasm hash of the pending upgrade request.
`NoPendingUpgrade`	`41`	Raised by `revoke_next_upgrade` when there is no pending upgrade to cancel.

Functions

`__constructor`

Categories	`constructor`
Authorization	—
Parameters	`env: Env` `owner: Address` — Initial address of the contract owner. `pool_a: Address` — Initial address of pool A, which accepts tokens for minting `pool_b: Address` — Initial address of pool B, which accepts tokens for redeeming `tokens_accepted_by_a: Vec<Address>` — Initial list of tokens that pool A accepts as input for minting. `tokens_accepted_by_b: Vec<Address>` — Initial list of tokens that pool B accepts as input for redeeming.
Returns	`()`
Reads	—
Mutates	`Owner` — Set to `owner`. `PoolAccountA` — Set to `pool_a`. `PoolAccountB` — Set to `pool_b`. `AcceptedByA` — Each `token` in `tokens_accepted_by_a` is added to the set. (I.e. each corresponding key `AcceptedByA(token)` is added to the storage with value `()`.) `AcceptedByB` — Each `token` in `tokens_accepted_by_b` is added to the set. (I.e. each corresponding key `AcceptedByB(token)` is added to the storage with value `()`.)
Raises	—
Emits	—

Initializes the contract with an owner, two pool accounts, and the sets of tokens accepted by each pool.

`request_upgrade`

Categories	`upgrade`
Authorization	`Owner`
Parameters	`env: Env` `new_wasm_hash: BytesN<32>` — Hash of the new contract's Wasm bytecode to upgrade to.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `EtNextUpgrade` — To check that no upgrade request is already pending.
Mutates	`NewWasmHash` — Set to `new_wasm_hash`. Extends the TTL to `PENDING_TTL_LEDGERS` (if it would otherwise expire sooner). `EtNextUpgrade` — Set to `now + DELAY_SETTING`. Extends the TTL to `PENDING_TTL_LEDGERS` (if it would otherwise expire sooner).
Raises	`PendingRequestExists` — If `EtNextUpgrade` is set to a non-zero value.
Emits	`UpgradeRequested` — On success.

Requests a time-locked contract upgrade. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it otherwise would expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`pending_upgrade_established`	On success, a pending upgrade exists: `NewWasmHash` equals `new_wasm_hash` and `EtNextUpgrade` equals `now + DELAY_SETTING`.	After the `PendingRequestExists` guard rules out an already-pending request, the function writes `new_wasm_hash` to `NewWasmHash` and `now + DELAY_SETTING` to `EtNextUpgrade` before returning.

`upgrade`

Categories	`upgrade`
Authorization	`Owner`
Parameters	`env: Env` `new_wasm_hash: BytesN<32>` — Hash of the new contract's Wasm bytecode to upgrade to.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `NewWasmHash` — To verify it is set to `new_wasm_hash`. `EtNextUpgrade` — To check that it is set to a timestamp lower than the current one.
Mutates	`NewWasmHash` — Unset after the upgrade is executed. `EtNextUpgrade` — Unset after the upgrade is executed.
Raises	`InvalidWasmHash` — If `NewWasmHash` is not set to `new_wasm_hash`. `TooEarlyToExecute` — If `EtNextUpgrade` is set to a timestamp at least the current one.
Emits	`ContractUpgraded` — On success.

Executes a pending time-locked contract upgrade, replacing the contract's Wasm bytecode with `NewWasmHash`. Clears the pending upgrade state on success.

Calls

Soroban host (`env.deployer()`)	`update_current_contract_wasm`	Replaces the contract's Wasm bytecode with `new_wasm_hash`. Takes effect after the current invocation completes: the executing code finishes running as the old version.

Postconditions

Name	Property	Proof
`timelock_respected`	On success, the current ledger timestamp is strictly greater than the effective time that was stored in `EtNextUpgrade`.	Guarded by the `TooEarlyToExecute` check. The slot is necessarily set at that point: the `InvalidWasmHash` check has already passed, so `NewWasmHash` is set, and `pending_upgrade_consistent` then implies `EtNextUpgrade` is set.
`wasm_replaced`	On success, the contract's Wasm bytecode is replaced with `new_wasm_hash`, taking effect after the current invocation completes.	Via `update_current_contract_wasm`; see the corresponding calls entry.
`pending_upgrade_cleared`	On success, `NewWasmHash` and `EtNextUpgrade` are unset.	Both are removed unconditionally before the function returns.

`revoke_next_upgrade`

Categories	`upgrade`
Authorization	`Owner`
Parameters	`env: Env`
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `NewWasmHash` — To verify a pending upgrade exists.
Mutates	`NewWasmHash` — Unset on success. `EtNextUpgrade` — Unset on success.
Raises	`NoPendingUpgrade` — If `NewWasmHash` is not set.
Emits	`UpgradeRevoked` — On success.

Cancels a pending contract upgrade. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`pending_upgrade_cleared`	On success, `NewWasmHash` and `EtNextUpgrade` are unset.	After the `NoPendingUpgrade` guard (which requires `NewWasmHash` to be set) passes, both slots are removed unconditionally before returning.

`request_owner_transfer`

Categories	`ownership`
Authorization	`Owner`
Parameters	`env: Env` `new_owner: Address` — Address of the requested new owner.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `EtNextOwner` — To verify that no ownership transfer is already pending.
Mutates	`PendingOwner` — Set to `new_owner` . Extends the TTL to `PENDING_TTL_LEDGERS` (if it would otherwise expire sooner). `EtNextOwner` — Set to `now + DELAY_SETTING` . Extends the TTL to `PENDING_TTL_LEDGERS` (if it would otherwise expire sooner).
Raises	`PendingRequestExists` — If `EtNextOwner` is set to a non-zero value.
Emits	`OwnerTransferRequested` — On success.

Requests a time-locked ownership transfer. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`pending_transfer_established`	On success, a pending ownership transfer exists: `PendingOwner` equals `new_owner` and `EtNextOwner` equals `now + DELAY_SETTING` .	After the `PendingRequestExists` guard rules out an already-pending transfer, the function writes `new_owner` to `PendingOwner` and `now + DELAY_SETTING` to `EtNextOwner` , unconditionally, before returning.

`accept_owner`

Categories	`ownership`
Authorization	`PendingOwner`
Parameters	`env: Env`
Returns	`()`
Reads	`PendingOwner` — To verify a transfer is pending, to authenticate the caller as the pending owner, to set `Owner` , and to set as the `new_owner` field of the emitted `OwnerTransferred` . `EtNextOwner` — To check that the time-lock has elapsed (i.e. it is set to a timestamp lower than the current one). `Owner` — To set as the `old_owner` field of the emitted `OwnerTransferred` .
Mutates	`Owner` — Set to `PendingOwner` (the accepted new owner). `PendingOwner` — Unset on success. `EtNextOwner` — Unset on success.
Raises	`NoPendingOwner` — If `PendingOwner` is not set. `TooEarlyToExecute` — If `EtNextOwner` is set to a timestamp at least the current one.
Emits	`OwnerTransferred` — On success.

Executes a pending time-locked ownership transfer, setting `Owner` to the pending owner. Clears the pending transfer state on success. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than

`INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`timelock_respected`	On success, the current ledger timestamp is strictly greater than the effective time that was stored in `EtNextOwner` .	Guarded by the `TooEarlyToExecute` check. The slot is necessarily set at that point: the `NoPendingOwner` check has already passed, so `PendingOwner` is set, and `pending_owner_consistent` then implies `EtNextOwner` is set.
`ownership_transferred`	On success, `Owner` is set to the address that was stored in `PendingOwner` .	Set before the function returns.
`pending_transfer_cleared`	On success, `PendingOwner` and `EtNextOwner` are unset.	Both are removed unconditionally before the function returns.

`revoke_next_owner`

Categories	`ownership`
Authorization	`Owner`
Parameters	`env: Env`
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	`PendingOwner` — Unset. `EtNextOwner` — Unset.
Raises	—
Emits	`OwnerRevoked` — On success.

Cancels a pending ownership transfer, unconditionally clearing the pending transfer state. Does not require a transfer to actually be pending.

Postconditions

Name	Property	Proof
`pending_transfer_cleared`	On success, `PendingOwner` and `EtNextOwner` are unset.	Both are unset unconditionally.

`set_pool_account_a`

Categories	`configuration`
Authorization	`Owner`
Parameters	`env: Env` `pool: Address` — New address of pool A.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	`PoolAccountA` — Set to `pool` .
Raises	—
Emits	`SetPoolAccountA` — On success.

Sets the address of pool A. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`pool_account_a_set`	On success, `PoolAccountA` is `pool`.	Direct write before the function returns.

`set_pool_account_b`

Categories	`configuration`
Authorization	`Owner`
Parameters	`env: Env` `pool: Address` — New address of pool B.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	`PoolAccountB` — Set to `pool`.
Raises	—
Emits	`SetPoolAccountB` — On success.

Sets the address of pool B. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`pool_account_b_set`	On success, `PoolAccountB` is `pool`.	Direct write before the function returns.

`set_accepted_by_a`

Categories	`configuration`
Authorization	`Owner`
Parameters	`env: Env` `token: Address` — Token address to add or remove. `accepted: bool` — If `true`, `token` is added to the set, otherwise it is removed.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	`AcceptedByA` — If `accepted` is `true`, `token` is added to the set (the key `AcceptedByA(token)` is set to `()`), otherwise it is removed.
Raises	—
Emits	`SetAcceptedByA` — On success.

Adds or removes `token` from the set of tokens accepted by pool A. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`membership_reflects_accepted`	On success, `token` is in `AcceptedByA` iff `accepted` is `true`.	The two branches set or remove the key `AcceptedByA(token)` accordingly, unconditionally, before the function returns.

`set_accepted_by_b`

Categories	`configuration`
Authorization	`Owner`
Parameters	`env: Env` `token: Address` — Token address to add or remove. `accepted: bool` — If `true`, `token` is added to the set, otherwise it is removed.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	`AcceptedByB` — If `accepted` is `true`, `token` is added to the set (the key `AcceptedByB(token)` is set to `()`), otherwise it is removed.
Raises	—
Emits	`SetAcceptedByB` — On success.

Adds or removes `token` from the set of tokens accepted by pool B. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`membership_reflects_accepted`	On success, `token` is in `AcceptedByB` iff `accepted` is `true`.	The two branches set or remove the key `AcceptedByB(token)` accordingly, unconditionally, before the function returns.

`request_to_mint`

Categories	`core`
Authorization	`user`
Parameters	`env: Env` `user: Address` — The account requesting the mint. Must authorize the call, and the token transfer is made from this account. `transferred_token: Address` — The token transferred in as input for minting. Must be accepted by pool A. `for_token: Address` — The token the user wishes to mint. Must be accepted by pool B. `amount: i128` — Amount of `transferred_token` to transfer to pool A. Must be non-negative. `preprice: u128` — Caller-supplied reference price, emitted in `MintRequest` and not otherwise used on-chain. `slippage: u128` — Caller-supplied slippage tolerance, emitted in `MintRequest` and not otherwise used on-chain. `timestamp: u64` — Caller-supplied timestamp (Unix seconds) used for the freshness check. Must be within `DELAY_MAX` seconds of the current ledger time. `extra_data: Bytes` — Caller-supplied opaque data, emitted in `MintRequest` and not otherwise used on-chain.
Returns	`()`
Reads	`AcceptedByA` — To check that `transferred_token` is accepted by pool A as input for minting. `AcceptedByB` — To check that `for_token` is accepted by pool B. `PoolAccountA` — To get the destination address for the token transfer.
Mutates	—
Raises	`NegativeAmountNotAllowed` — If `amount` is negative. `InvalidTokenForMinting` — If `transferred_token` is not accepted by pool A. `InvalidForToken` — If `for_token` is not accepted by pool B. `InvalidTimestamp` — If the current ledger time exceeds `timestamp` by more than `DELAY_MAX` seconds.
Emits	`MintRequest` — On success.

Submits a request to mint `for_token` by transferring `amount` of `transferred_token` into pool A. Validates that the input token is accepted by pool A, that the target token is accepted by pool B, and that the supplied timestamp is fresh; then transfers the input tokens from the user to pool A and emits a `MintRequest` event for off-chain fulfillment. The mint itself is not performed on-chain by this function. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Calls

The input token contract, at the address `transferred_token`.	`transfer`	Transfers `amount` of `transferred_token` from `user` to pool A (`PoolAccountA`). Reverts the whole invocation if the user has insufficient balance or has not authorized the transfer.

Postconditions

Name	Property	Proof
`tokens_transferred`	On success, `amount` of `transferred_token` has been transferred from `user` to pool A (`PoolAccountA`).	Via the input token contract's `transfer`; see the calls entry. Reached only after the amount, token-validity, and timestamp checks have passed.
`own_storage_unchanged`	The function does not modify any of the contract's own storage slots (only the instance TTL is bumped and an event is emitted).	No write or remove on any slot occurs in the body; the only state effect is the external token transfer.

`request_to_redeem`

Categories	`core`
Authorization	`user`
Parameters	`env: Env` `user: Address` — The account requesting the redemption. Must authorize the call, and the token transfer is made from this account. `transferred_token: Address` — The token transferred in as input for redeeming. Must be accepted by pool B. `for_token: Address` — The token the user wishes to redeem for. Must be accepted by pool A. `amount: i128` — Amount of `transferred_token` to transfer to pool B. Must be non-negative. `preprice: u128` — Caller-supplied reference price, emitted in `RedeemRequest` and not otherwise used on-chain. `slippage: u128` — Caller-supplied slippage tolerance, emitted in `RedeemRequest` and not otherwise used on-chain. `timestamp: u64` — Caller-supplied timestamp (Unix seconds) used for the freshness check. Must be within `DELAY_MAX` seconds of the current ledger time. `extra_data: Bytes` — Caller-supplied opaque data, emitted in `RedeemRequest` and not otherwise used on-chain.
Returns	`()`
Reads	`AcceptedByB` — To check that `transferred_token` is accepted by pool B as input for redeeming. `AcceptedByA` — To check that `for_token` is accepted by pool A. `PoolAccountB` — To get the destination address for the token transfer.
Mutates	—
Raises	`NegativeAmountNotAllowed` — If `amount` is negative. `InvalidTokenForRedeeming` — If `transferred_token` is not accepted by pool B. `InvalidForToken` — If `for_token` is not accepted by pool A. `InvalidTimestamp` — If the current ledger time exceeds `timestamp` by more than `DELAY_MAX` seconds.
Emits	`RedeemRequest` — On success.

Submits a request to redeem `transferred_token` for `for_token` by transferring `amount` of `transferred_token` into pool B. Validates that the input token is accepted by pool B, that the target token is accepted by pool A, and that the supplied timestamp is fresh; then transfers the input tokens from the user to pool B and emits a `RedeemRequest` event for off-chain fulfillment. The redemption itself is not performed on-chain by this function. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Calls

The input token contract, at the address `transferred_token`.	`transfer`	Transfers `amount` of `transferred_token` from `user` to pool B (`PoolAccountB`). Reverts the whole invocation if the user has insufficient balance or has not authorized the transfer.

Postconditions

Name	Property	Proof
`tokens_transferred`	On success, `amount` of `transferred_token` has been transferred from `user` to pool B (`PoolAccountB`).	Via the input token contract's `transfer`; see the calls entry. Reached only after the amount, token-validity, and timestamp checks have passed.
`own_storage_unchanged`	The function does not modify any of the contract's own storage slots (only the instance TTL is bumped and an event is emitted).	No write or remove on any slot occurs in the body; the only state effect is the external token transfer.

`owner`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Address` — The current owner address.
Reads	`Owner` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the current owner address.

Postconditions

Name	Property	Proof
`owner_return_value`	The result is `Owner` .	Direct read. The slot is set per `owner_always_set` , so the read cannot fail.

`pending_owner`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<Address>` — The pending owner address, or `None` if no transfer is pending.
Reads	`PendingOwner` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the address of the requested new owner, or `None` if no ownership transfer is pending.

Postconditions

Name	Property	Proof
`pending_owner_return_value`	The result is `PendingOwner` if set, otherwise `None` .	Direct read.

`et_next_owner`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<u64>` — Effective time of the pending ownership transfer (Unix timestamp, seconds), or `None` if no transfer is pending.
Reads	`EtNextOwner` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the effective time after which the pending ownership transfer can be executed, or `None` if no transfer is pending.

Postconditions

Name	Property	Proof
`et_next_owner_return_value`	The result is `Some(EtNextOwner)` if the slot is set to a non-zero value, otherwise `None`.	An absent value or the `0` sentinel maps to `None`, any other value to `Some`.

`pool_account_a`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Address` — The current address of pool A.
Reads	`PoolAccountA` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the address of pool A.

Postconditions

Name	Property	Proof
`pool_account_a_return_value`	The result is `PoolAccountA`.	Direct read. The slot is set per `pool_account_a_always_set`, so the read cannot fail.

`pool_account_b`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Address` — The current address of pool B.
Reads	`PoolAccountB` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the address of pool B.

Postconditions

Name	Property	Proof
`pool_account_b_return_value`	The result is `PoolAccountB`.	Direct read. The slot is set per `pool_account_b_always_set`, so the read cannot fail.

`is_accepted_by_a`

Categories	`view`
Authorization	—
Parameters	`env: Env` `token: Address` — Token address to check.
Returns	`bool` — `true` iff `token` is in `AcceptedByA`.
Reads	`AcceptedByA` — To check membership of `token`.
Mutates	—
Raises	—
Emits	—

Returns whether `token` is accepted by pool A as input for minting.

Postconditions

Name	Property	Proof
`is_accepted_by_a_return_value`	The result is `true` iff `token` is in `AcceptedByA`.	Direct existence check of the key `AcceptedByA(token)` (`has`).

`is_accepted_by_b`

Categories	`view`
Authorization	—
Parameters	`env: Env` `token: Address` — Token address to check.
Returns	`bool` — `true` iff `token` is in `AcceptedByB`.
Reads	`AcceptedByB` — To check membership of `token`.
Mutates	—
Raises	—
Emits	—

Returns whether `token` is accepted by pool B as input for redeeming.

Postconditions

Name	Property	Proof
`is_accepted_by_b_return_value`	The result is `true` iff `token` is in `AcceptedByB`.	Direct existence check of the key `AcceptedByB(token)` (`has`).

`next_upgrade_wasm_hash`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<BytesN<32>>` — The Wasm hash of the pending upgrade, or `None` if no upgrade is pending.
Reads	`NewWasmHash` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the Wasm hash of the pending upgrade, or `None` if no upgrade is pending.

Postconditions

Name	Property	Proof
`next_upgrade_wasm_hash_return_value`	The result is `NewWasmHash` if set, otherwise `None`.	Direct read.

`et_next_upgrade`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<u64>` — Effective time of the pending upgrade (Unix timestamp, seconds), or `None` if no upgrade is pending.
Reads	`EtNextUpgrade` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the effective time after which the pending upgrade can be executed, or `None` if no upgrade is pending.

Postconditions

Name	Property	Proof
`et_next_upgrade_return_value`	The result is `Some(EtNextUpgrade)` if the slot is set to a non-zero value, otherwise `None`.	An absent value or the `0` sentinel maps to `None`, any other value to `Some`.

`Token` Contract Overview

Xaum is a Stellar/Soroban smart token for a real-world asset. It implements the standard token interface (balances, transfers, allowances) via `transfer`, `transfer_from`, `approve`, `balance`, and `allowance`, alongside real-world-asset controls: time-locked issuance, redemption by burning, an address block list, owner-forced transfers, and an upgrade path. Token supply is tracked by `TotalSupply` and per-holder `Balance` entries; the standard `soroban-token-sdk` metadata (name, symbol, decimals) is fixed at construction and immutable thereafter.

Three roles hold privileges. The owner is the primary administrator: it sets the operational and governance delays, appoints the operator and revoker, performs forced transfers, and drives contract upgrades and ownership transfers. The operator is the issuance role: it mints (via `mint_to`), burns on behalf of redeeming customers (via `burn`), maintains the mint budget, and manages the block list. The revoker is a narrow safety role: it can cancel pending operator and delay changes (`revoke_next_operator`, `revoke_next_delay`) and revoke pending mint requests (`revoke_mint_request`). Ownership itself transfers through a two-step, time-locked request/accept handshake (`request_owner_transfer` then `accept_owner`).

Sensitive changes use a request/execute time-lock. A request records a pending value together with an effective time of `now + delay`; the change can be executed only once that time has passed, and may be cancelled before then. Operator, revoker, and delay changes, contract upgrades, and mints are gated by the operational delay `Delay`; governance-delay changes and ownership transfers are gated by the governance delay `GovDelay`. Both delays are uninitialized at construction and read as `0` until first set, so every time-lock is initially inactive — a request can be executed as soon as the next ledger — until the owner configures the relevant delay.

The mint budget `MintBudget` is not a security boundary against the operator. Both `change_mint_budget` and `mint_to` are operator-authed, so the operator can raise its own budget arbitrarily (no cap, no owner approval, no time-lock on the budget itself) and then mint against it; equivalently, the operator alone controls the quantity `TotalSupply + MintBudget`, which minting and burning leave unchanged and only `change_mint_budget` moves. The budget is therefore an internal accounting and accidental-over-mint rail the operator maintains for itself, not a check on a malicious or compromised operator. This is presumably by design — the operator is the trusted minter, issuing against off-chain reserves.

The block list is a spend-side restriction only. A blocked address cannot send tokens (as `from` in `transfer` / `transfer_from`) or act as a `spender` in `transfer_from`, but it can still receive tokens, be minted to, have allowances set over its tokens, and have its balance moved by the owner via `forced_transfer`, which bypasses the block list by design.

Invariants

Name	Property	Proof
`owner_always_set`	`Owner` is always set.	Established by `__constructor`, which always sets `Owner`. `accept_owner` overwrites it (with a set `PendingOwner`) but never unsets it. No other function writes the slot, and there is no remover for it.
`pending_owner_consistent`	`PendingOwner` and `EtNextOwner` are either both set or both unset.	Established by `request_owner_transfer`, which always sets both. Preserved by `accept_owner` and `revoke_next_owner`, which always unset both together. No other function mutates either slot. Both slots are temporary entries, always written in the same transaction with the same TTL (`PENDING_TTL_LEDGERS`), so they expire together.
`et_next_owner_nonzero_when_set`	When `EtNextOwner` is set, its value is non-zero; the zero value is reserved as the "absent" sentinel and is never stored.	The sole writer of `EtNextOwner` is `request_owner_transfer`, which stores `now + gov_delay`, where `now` is the ledger timestamp and `gov_delay` is `GovDelay` if it is set and `0` otherwise. Because `GovDelay` is not initialized by `__constructor` and stays unset until a governance delay change first takes effect, the addend gives no positive lower bound, so non-zeroness rests on `now`. By `assume_no_overflow` the stored value is the exact sum, hence `>= now` since `gov_delay >= 0`; and by `assume_positive_ledger_time` `now >= 1`. Therefore the stored value is at least `1`, never the `0` sentinel.
`operator_always_set`	`Operator` is always set.	Established by `__constructor`, which always sets `Operator`. `set_operator` overwrites it (on its execute path, with the requested new operator) but never unsets it. No other function writes the slot, and there is no remover for it.
`next_operator_consistent`	`NextOperator` and `EtNextOperator` are either both set or both unset.	Established by `set_operator`, which on its request path always sets both. Preserved by `set_operator` on its execute path and by `revoke_next_operator`, which always unset both together. No other function mutates either slot. Both slots are temporary entries, always written in the same transaction with the same TTL (`PENDING_TTL_LEDGERS`), so they expire together.
`et_next_operator_nonzero_when_set`	When `EtNextOperator` is set, its value is non-zero; the zero value is reserved as the "absent" sentinel and is never stored.	The sole writer of `EtNextOperator` is `set_operator` on its request path, which stores `now + delay`, where `now` is the ledger timestamp and `delay` is `Delay` if it is set and `0` otherwise. Because `Delay` is not initialized by `__constructor` and stays unset until a delay change first takes effect, the addend gives no positive lower bound, so non-zeroness rests on `now`. By `assume_no_overflow` the stored value is the exact sum, hence `>= now` since `delay >= 0`; and by `assume_positive_ledger_time` `now >= 1`. Therefore the stored value is at least `1`, never the `0` sentinel.
`revoker_always_set`	`Revoker` is always set.	Established by `__constructor`, which always sets `Revoker`. `set_revoker` overwrites it (on its execute path, with the requested new revoker) but never unsets it. No other function writes the slot, and there is no remover for it.
`next_revoker_consistent`	`NextRevoker` and `EtNextRevoker` are either both set or both unset.	Established by `set_revoker`, which on its request path always sets both. Preserved by `set_revoker` on its execute path and by `revoke_next_revoker`, which always unset both together. No other function mutates either slot. Both slots are temporary entries, always written in the same transaction with the same TTL (`PENDING_TTL_LEDGERS`), so they expire together.
`et_next_revoker_nonzero_when_set`	When `EtNextRevoker` is set, its value is non-zero; the zero value is reserved as the "absent" sentinel and is never stored.	The sole writer of `EtNextRevoker` is `set_revoker` on its request path, which stores `now + delay`, where `now` is the ledger timestamp and `delay` is `Delay` if it is set and `0` otherwise. Because `Delay` is not initialized by `__constructor` and stays unset until a delay change first takes effect, the addend gives no positive lower bound, so non-zeroness rests on `now`. By `assume_no_overflow` the stored value is the exact sum, hence `>= now` since `delay >= 0`; and by `assume_positive_ledger_time` `now >= 1`. Therefore the stored value is at least `1`, never the `0` sentinel.
`pending_upgrade_consistent`	`NewWasmHash` and `EtNextUpgrade` are either both set or both unset.	Established by `request_upgrade`, which always sets both. Preserved by `upgrade` and `revoke_next_upgrade`, which always unset both together. No other function mutates either slot. Both slots are temporary entries, always written in the same transaction with the same TTL (`PENDING_TTL_LEDGERS`), so they expire together.
`et_next_upgrade_nonzero_when_set`	When `EtNextUpgrade` is set, its value is non-zero; the zero value is reserved as the "absent" sentinel and is never stored.	The sole writer of `EtNextUpgrade` is `request_upgrade`, which stores `now + delay`, where `now` is the ledger timestamp and `delay` is `Delay` if it is set and `0` otherwise. Because `Delay` is not initialized by `__constructor` and stays unset until a delay change first takes effect, the addend gives no positive lower bound, so non-zeroness rests on `now`. By `assume_no_overflow` the stored value is the exact sum, hence `>= now` since `delay >= 0`; and by `assume_positive_ledger_time` `now >= 1`. Therefore the stored value is at least `1`, never the `0` sentinel.
`next_delay_within_bounds`	If `NextDelay` is set, then `MIN_DELAY <= NextDelay <= MAX_DELAY`. In particular, `NextDelay != 0`.	`NextDelay` is only set by `set_delay`, which stores the caller-supplied `new_delay` only after rejecting `new_delay < MIN_DELAY` with `DelayTooSmall` and `new_delay > MAX_DELAY` with `DelayTooLarge`. And since `MIN_DELAY > 0`, we get non-zeroness of the storage slot. The slot is otherwise absent: removed when the change takes effect, is revoked, or expires. Therefore it never holds an out-of-range value.
`et_next_delay_nonzero_when_set`	When `EtNextDelay` is set, its value is non-zero; the zero value is reserved as the "absent" sentinel and is never stored.	The sole writer of `EtNextDelay` is `set_delay` on its request path, which stores `now + delay`, where `now` is the ledger timestamp and `delay` is `Delay` if it is set and `0` otherwise. Because `Delay` is not initialized by `__constructor` and stays unset until a delay change first takes effect, the addend gives no positive lower bound, so non-zeroness rests on `now`. By `assume_no_overflow` the stored value is the exact sum, hence `>= now` since `delay >= 0`; and by `assume_positive_ledger_time` `now >= 1`. Therefore the stored value is at least `1`, never the `0` sentinel.
`next_delay_consistent`	`NextDelay` and `EtNextDelay` are either both set or both unset.	Established by `set_delay`, which on its request path always sets both. Preserved by `set_delay` on its execute path and by `revoke_next_delay`, which always unset both together. No other function mutates either slot. Both slots are temporary entries, always written in the same transaction with the same TTL (`PENDING_TTL_LEDGERS`), so they expire together.

delay_within_bounds	If Delay is set, then MIN_DELAY <= Delay <= MAX_DELAY. In particular, Delay != 0.	Delay is written only by set_delay, on its execute path, which stores new_delay. That path runs only when EtNextDelay is set to a non-zero value, so by next_delay_consistent, slot NextDelay is set, and the guard preceding the write rejects the call with PendingRequestExists unless new_delay == NextDelay. By next_delay_within_bounds we have MIN_DELAY <= NextDelay <= MAX_DELAY, therefore after the update MIN_DELAY <= Delay <= MAX_DELAY. And since MIN_DELAY > 0, we get non-zeroness of the storage slot.
delay_never_unset	Delay is not initialized by __constructor and is absent (reading as 0 via unwrap_or(0)) until the first delay change takes effect via set_delay. Once set, it is never unset.	The sole writer of Delay is set_delay, on its execute path. Notably, __constructor does not initialize Delay, so the slot is absent until that path first runs. Delay lives in instance storage and no function removes it (there is no remover), so once set it remains set permanently.
next_gov_delay_within_bounds	If NextGovDelay is set, then MIN_DELAY <= NextGovDelay <= MAX_DELAY. In particular, NextGovDelay != 0.	NextGovDelay is only set by set_gov_delay, which stores the caller-supplied new_delay only after rejecting new_delay < MIN_DELAY with DelayTooSmall and new_delay > MAX_DELAY with DelayTooLarge. And since MIN_DELAY > 0, we get non-zeroness of the storage slot. The slot is otherwise absent: removed when the change takes effect, is revoked, or expires.
et_next_gov_delay_nonzero_when_set	When EtNextGovDelay is set, its value is non-zero; the zero value is reserved as the "absent" sentinel and is never stored.	The sole writer of EtNextGovDelay is set_gov_delay on its request path, which stores now + delay, where now is the ledger timestamp and delay is GovDelay if it is set and 0 otherwise. Because GovDelay is not initialized by __constructor and stays unset until a governance delay change first takes effect, the addend gives no positive lower bound, so non-zeroness rests on now. By assume_no_overflow the stored value is the exact sum, hence >= now since delay >= 0; and by assume_positive_ledger_time now >= 1. Therefore the stored value is at least 1, never the 0 sentinel.
next_gov_delay_consistent	NextGovDelay and EtNextGovDelay are either both set or both unset.	Established by set_gov_delay, which on its request path always sets both. Preserved by set_gov_delay on its execute path and by revoke_next_gov_delay, which always unset both together. No other function mutates either slot. Both slots are temporary entries, always written in the same transaction with the same TTL (PENDING_TTL_LEDGERS), so they expire together.
gov_delay_within_bounds	If GovDelay is set, then MIN_DELAY <= GovDelay <= MAX_DELAY. In particular, GovDelay != 0.	GovDelay is written only by set_gov_delay, on its execute path, which stores new_delay. That path runs only when EtNextGovDelay is set to a non-zero value, so by next_gov_delay_consistent, slot NextGovDelay is set, and the guard preceding the write rejects the call with PendingRequestExists unless new_delay == NextGovDelay. By next_gov_delay_within_bounds we have MIN_DELAY <= NextGovDelay <= MAX_DELAY, therefore after the update MIN_DELAY <= GovDelay <= MAX_DELAY. And since MIN_DELAY > 0, we get non-zeroness of the storage slot.
gov_delay_never_unset	GovDelay is not initialized by __constructor and is absent (reading as 0 via unwrap_or(0)) until the first governance delay change takes effect via set_gov_delay. Once set, it is never unset.	The sole writer of GovDelay is set_gov_delay, on its execute path. Notably, __constructor does not initialize GovDelay, so the slot is absent until that path first runs. GovDelay lives in instance storage and no function removes it (there is no remover), so once set it remains set permanently.
balance_never_unset	No function unsets a Balance entry: once a balance has been written for an address, the entry is never removed (an address with no entry reads as 0).	Balance entries are written only by write_balance (a plain set), reached through update_balance from transfer, transfer_from, mint_to, burn, and forced_transfer. No function removes a Balance entry (there is no remover), so once written an entry persists; it may be overwritten (including to 0) but never deleted.
balance_nonneg	Every balance stored in Balance is non-negative.	The only functions that write a Balance entry are transfer, transfer_from, forced_transfer, mint_to, and burn (cf. balance_never_unset). An absent entry reads as 0, the base case. Assume every stored balance is non-negative before a call; each writer preserves this on success. transfer: by balance_transferred_distinct the sender becomes b_from - amount and the recipient b_to + amount (each relative to its current value, an absent entry counting as 0), and by self_transfer_unchanged the balance is unchanged when sender and recipient coincide. On success neither NegativeAmountNotAllowed nor InsufficientBalance was raised, so amount >= 0 and amount <= b_from; hence b_from - amount >= 0, and b_to + amount >= 0 since b_to >= 0 by hypothesis. The unchanged case stays non-negative by hypothesis. transfer_from: identically, by balance_transferred_distinct and self_transfer_unchanged, with the same NegativeAmountNotAllowed and InsufficientBalance guards giving amount >= 0 and amount <= b_from, so both written balances are >= 0. forced_transfer: identically, by balance_transferred_distinct and self_transfer_unchanged, with the same NegativeAmountNotAllowed and InsufficientBalance guards, so both written balances are >= 0. mint_to: a Balance entry is written only on the execute path, where mint_executed increases the receiver balance by amount (relative to its current value, an absent entry counting as 0); success implies no NegativeAmountNotAllowed, so amount >= 0, and the result is >= 0 since the prior balance is >= 0 by hypothesis. The request path (request_recorded) writes no balance. burn: by operator_balance_burned the operator balance becomes b - amount (relative to its current value, an absent entry counting as 0); success implies no InsufficientBalance, so amount <= b, hence b - amount >= 0. By induction over all calls, every stored balance is non-negative.

`allowance_nonneg`	Every allowance amount stored in `Allowance` is non-negative.	The only functions that write an `Allowance` entry are `approve` and `transfer_from`. An absent entry reads as `0`, the base case. Assume every stored allowance amount is non-negative before a call; each writer preserves this on success. `approve`: by `allowance_set` the entry for the key (`from`, `spender`) is set to `AllowanceValue { amount: amount, expiration_ledger: expiration_ledger }`, so the stored amount is `amount`. On success no `NegativeAmountNotAllowed` was raised, so `amount >= 0`. `transfer_from`: by `allowance_decremented`, for `amount > 0` the stored amount of the entry for (`from`, `spender`) becomes `a - amount`, where `a` is its prior amount (`>= 0` by hypothesis, or `0` if absent or expired); on success no `InsufficientAllowance` was raised, so `a >= amount` and `a - amount >= 0`. For `amount == 0` the same postcondition gives `Allowance` unchanged, staying `>= 0` by hypothesis. By induction over all calls, every stored allowance amount is non-negative.
`mint_budget_nonneg`	If `MintBudget` is set, then `MintBudget >= 0`.	`MintBudget` starts unset (reading as `0` via `unwrap_or(0)`) and is written by exactly three functions. `change_mint_budget` stores `MintBudget + delta` only after rejecting a negative result with `MintBudgetNotEnough`, so the stored value is `>= 0`. `burn` increases it by the burned `amount`, which it has checked to be non-negative (rejecting negatives with `NegativeAmountNotAllowed`), preserving non-negativity. `mint_to` decreases it by the minted `amount` on its execute path, but only after the guard rejecting `amount > MintBudget` with `MintBudgetNotEnough`, so `amount <= MintBudget` and the result stays `>= 0`. No other function writes the slot, so by induction it is non-negative whenever set.
`mint_budget_never_unset`	`MintBudget` is not initialized by `__constructor` and is absent (reading as `0` via `unwrap_or(0)`) until first written by `change_mint_budget`, `burn`, or `mint_to`. Once set, it is never unset.	The slot lives in instance storage and is written only by `change_mint_budget`, `burn`, and `mint_to`. `__constructor` does not initialize it, so it is absent until one of those first runs. No function removes the slot (there is no remover), so once set it remains set permanently.
`total_supply_nonneg`	If `TotalSupply` is set, then `TotalSupply >= 0`.	`TotalSupply` starts unset (reading as `0` via `unwrap_or(0)`) and is written on only two paths. `mint_to` increases it by the minted `amount`, which it has already checked to be non-negative (rejecting negatives with `NegativeAmountNotAllowed`), so non-negativity is preserved. `burn` decreases it by the burned `amount`, but only after the guard rejecting `amount > TotalSupply` with `TotalSupplyUnderflow`, so `amount <= TotalSupply` and the result stays `>= 0`. No other function writes the slot, so by induction it is non-negative whenever set.
`total_supply_never_unset`	`TotalSupply` is not initialized by `__constructor` and is absent (reading as `0` via `unwrap_or(0)`) until first written by `mint_to` or `burn`. Once set, it is never unset.	The slot lives in instance storage and is written only by `mint_to` and `burn` (the mint and burn paths of `update_balance`). `__constructor` does not initialize it, so it is absent until one of those paths first runs. No function removes the slot (there is no remover), so once set it remains set permanently.
`total_supply_set`	If `TotalSupply` is set, then there exists a `Balance` entry.	`TotalSupply` is written only by `mint_to` and `burn` (cf. `total_supply_never_unset`); before either first runs it is unset, so the implication holds vacuously. Each call that writes `TotalSupply` also writes a `Balance` entry in the same call, and that entry persists by `balance_never_unset`. `mint_to`: on the execute path, `mint_executed` increases `TotalSupply` by `amount` and increases the balance of `receiver` by `amount` (relative to its current value, counting an absent entry as `0`); the latter writes the receiver's `Balance` entry, so an entry exists. The request path (`request_recorded`) writes no `TotalSupply`. `burn`: `total_supply_decreased` decreases `TotalSupply`, and in the same call `operator_balance_burned` guarantees the operator has a `Balance` entry afterward. So whenever `TotalSupply` is written, a `Balance` entry exists, and by `balance_never_unset` it is never removed thereafter. Hence whenever `TotalSupply` is set, a `Balance` entry exists.

`total_supply_eq_sum_of_balances`	If `TotalSupply` is unset then every `Balance` entry is `0`. If it is set, then `TotalSupply` equals the sum of all `Balance` entries.	Let `TS` denote `TotalSupply` read via `unwrap_or(0)` (its stored value if set, `0` if unset) and `S` the sum of the stored values of all `Balance` entries. We show `TS == S` holds after every call that mutates either storage slot. The two cases of the property then follow immediately. If `TotalSupply` is unset, then `TS == 0`, so `S == 0`; by `balance_nonneg` every stored balance is `>= 0`, and a sum of non-negative values is `0` only if each term is `0`, so every `Balance` entry is `0`. If `TotalSupply` is set, then `TS` is its stored value, which by `TS == S` equals the sum of all stored balances. Claim: `TS == S`, by induction over calls. Base case: `__constructor` does not initialize `TotalSupply` (cf. `total_supply_never_unset`) and writes no `Balance` entry, so `TS == 0 == S`. Inductive step: assume `TS == S` before a call. Only `transfer`, `transfer_from`, `forced_transfer`, `mint_to`, and `burn` write `TotalSupply` or any `Balance` entry (cf. `total_supply_never_unset` and `balance_never_unset`); every other function leaves both `TS` and `S` unchanged, preserving the equality. For each writer, on success the change in `TS` equals the change in `S`: `transfer`: by `total_supply_unchanged` `TS` is unchanged. By `balance_transferred_distinct` the sender's balance falls by the transferred amount and the recipient's rises by the same amount (when the two are distinct), so the two changes cancel; by `self_transfer_unchanged` the single shared entry is unchanged (when they coincide). No other entry is touched, so `S` is unchanged. Hence `ΔTS == ΔS == 0`. `transfer_from`: identically to `transfer`, by `total_supply_unchanged` `TS` is unchanged, and by `balance_transferred_distinct` and `self_transfer_unchanged` `S` is unchanged (the sender and recipient changes cancel when distinct, the shared entry is unchanged when they coincide). Hence `ΔTS == ΔS == 0`. `forced_transfer`: identically to `transfer`, by `total_supply_unchanged` `TS` is unchanged, and by `balance_transferred_distinct` and `self_transfer_unchanged` `S` is unchanged. Hence `ΔTS == ΔS == 0`. `mint_to`: on the request path (`request_recorded`) neither `TotalSupply` nor any `Balance` entry is written, so both are unchanged. On the execute path `mint_executed` increases `TotalSupply` by `amount` and the receiver balance by `amount`, touching no other entry; hence `ΔTS == ΔS == amount`. `burn`: `total_supply_decreased` decreases `TotalSupply` by `amount` and `operator_balance_burned` decreases the operator balance by `amount`, touching no other entry; hence `ΔTS == ΔS == -amount`. In every case `ΔTS == ΔS`, so `TS == S` is preserved. By induction `TS == S` holds after every call.
`supply_budget_sum_changed_only_by_change_mint_budget`	The sum `TotalSupply + MintBudget` (each read via `unwrap_or(0)`) is changed only by `change_mint_budget`, which changes it by `delta`. Every other function leaves it unchanged.	Write `B` for `TotalSupply + MintBudget`, each read via `unwrap_or(0)`. The only functions that write `TotalSupply` or `MintBudget` are `mint_to`, `burn`, and `change_mint_budget` (cf. `total_supply_never_unset`, whose only writers of `TotalSupply` are `mint_to` and `burn`, and `mint_budget_never_unset`, whose only writers of `MintBudget` are `change_mint_budget`, `burn`, and `mint_to`). Every other function writes neither slot, so leaves `B` unchanged. It remains to show `mint_to` and `burn` leave `B` unchanged, while `change_mint_budget` changes it by `delta`. `mint_to`: on the request path (`request_recorded`) neither slot is written, so `B` is unchanged. On the execute path `mint_executed` increases `TotalSupply` by `amount` and decreases `MintBudget` by `amount`, so the two changes cancel and `B` is unchanged. `burn`: `total_supply_decreased` decreases `TotalSupply` by `amount` and `mint_budget_increased` increases `MintBudget` by `amount`, so the two changes cancel and `B` is unchanged. `change_mint_budget`: `mint_budget_updated` sets `MintBudget` to its prior value plus `delta`, and `TotalSupply` is not mutated by it, so `B` changes by exactly `delta`. Hence `B` is changed only by `change_mint_budget`, by `delta`.
`metadata_immutable`	`Metadata` is always set, and its value never changes after construction.	Established by `__constructor`, which always writes `Metadata` (after the `decimal > 18` guard). `__constructor` is the sole writer and there is no remover, so once the contract is constructed the slot is permanently set and its value is immutable.
`decimal_at_most_18`	The `decimal` field of `Metadata` is at most `18`.	The sole writer of `Metadata` is `__constructor`, which rejects `decimal > 18` with `InvalidDecimal` before writing. By `metadata_immutable` the value never changes afterward, so the field remains `<= 18`.

Constants

Name	Type	Value	Description
`MIN_DELAY`	`u64`	`3600`	The minimum permitted value, in seconds, for the operational `Delay` and the governance `GovDelay`. Equal to one hour. Enforced by `set_delay` and `set_gov_delay`, which reject a smaller value with `DelayTooSmall`.
`MAX_DELAY`	`u64`	`3600 * 24 * 7`	The maximum permitted value, in seconds, for the operational `Delay` and the governance `GovDelay`. Equal to seven days. Enforced by `set_delay` and `set_gov_delay`, which reject a larger value with `DelayTooLarge`.
`DAY_IN_LEDGERS`	`u32`	`17280`	The number of ledgers in a day, used as the unit for the TTL (time-to-live) constants below. At roughly 5 seconds per ledger this is one day; the bump and threshold amounts are expressed as multiples of it.
`PENDING_TTL_LEDGERS`	`u32`	`10 * DAY_IN_LEDGERS`	The TTL bump applied to a pending-change entry when it is written: `PendingOwner` and `EtNextOwner`; `NextOperator` and `EtNextOperator`; `NextRevoker` and `EtNextRevoker`; `NextDelay` and `EtNextDelay`; `NextGovDelay` and `EtNextGovDelay`; and the upgrade slots `NewWasmHash` and `EtNextUpgrade`. Equal to 10 days, comfortably above the `MAX_DELAY` of 7 days, so a pending change cannot expire before its effective time becomes reachable.
`MINT_REQUEST_TTL_LEDGERS`	`u32`	`20 * DAY_IN_LEDGERS`	The TTL bump applied to a pending `MintRequest` entry when it is recorded by `mint_to` on the request path. Equal to 20 days, giving the operator a wide window to execute a registered mint request before the entry expires.
`INSTANCE_BUMP_AMOUNT`	`u32`	`30 * DAY_IN_LEDGERS`	The amount, in ledgers, by which the contract instance TTL is extended when bumped (the instance storage holds all instance-scoped state, including the configuration and pending-change slots). Equal to 30 days. Applied on essentially every entry point via the instance bump, gated by `INSTANCE_LIFETIME_THRESHOLD`.
`INSTANCE_LIFETIME_THRESHOLD`	`u32`	`INSTANCE_BUMP_AMOUNT - DAY_IN_LEDGERS`	The threshold, in ledgers, gating the instance TTL bump: the instance TTL is extended to `INSTANCE_BUMP_AMOUNT` only when its remaining lifetime would otherwise fall below this value. Equal to `INSTANCE_BUMP_AMOUNT` minus one day, so the bump is a no-op until the instance is within a day of expiry relative to the bump amount, avoiding a storage write on every call.
`BALANCE_BUMP_AMOUNT`	`u32`	`30 * DAY_IN_LEDGERS`	The amount, in ledgers, by which a `Balance` entry TTL is extended when the entry is written. Equal to 30 days. Gated by `BALANCE_LIFETIME_THRESHOLD`.
`BALANCE_LIFETIME_THRESHOLD`	`u32`	`BALANCE_BUMP_AMOUNT - DAY_IN_LEDGERS`	The threshold, in ledgers, gating the `Balance` entry TTL bump: a balance entry TTL is extended to `BALANCE_BUMP_AMOUNT` only when its remaining lifetime would otherwise fall below this value. Equal to `BALANCE_BUMP_AMOUNT` minus one day.
`ALLOW_BLOCK_EXTEND_AMOUNT`	`u32`	`30 * DAY_IN_LEDGERS`	The amount, in ledgers, by which a `Blocked` entry TTL is extended when an address is added to the block list. Equal to 30 days. Gated by `ALLOW_BLOCK_TTL_THRESHOLD`.
`ALLOW_BLOCK_TTL_THRESHOLD`	`u32`	`ALLOW_BLOCK_EXTEND_AMOUNT - DAY_IN_LEDGERS`	The threshold, in ledgers, gating the `Blocked` entry TTL bump: a block-list entry TTL is extended to `ALLOW_BLOCK_EXTEND_AMOUNT` only when its remaining lifetime would otherwise fall below this value. Equal to `ALLOW_BLOCK_EXTEND_AMOUNT` minus one day.

Storage

Name	Type	Lifetime	Description
`Allowance`	`Map<(Address, Address), AllowanceValue>`	`temporary`	Spending allowances, keyed by `(owner, spender)`: how many tokens the spender may move on the owner's behalf, together with an expiration ledger. An entry is set by `approve` (which rejects a negative amount with `NegativeAmountNotAllowed`, and a past expiration for a positive amount with `InvalidExpirationLedger`) and decremented by `transfer_from` as the spender draws on it (an insufficient live allowance reverts with `InsufficientAllowance`). An absent or expired entry reads as a zero allowance; a positive allowance is surfaced by the `allowance` getter and lives until its expiration ledger.
`Balance`	`Map<Address, i128>`	`persistent`	Per-address token balances, keyed by holder address. Entries are created or updated by the token movers: `transfer`, `transfer_from`, and `forced_transfer` move a balance between two addresses; `mint_to` credits the receiver; `burn` debits the operator. An address with no entry reads as `0`, and entries are never removed once written (a balance may be set to `0` but the entry persists).
`MintRequest`	`Map<BytesN<32>, u64>`	`temporary`	Pending mint requests, keyed by a request id (the `sha256` of the receiver, amount, and nonce) mapping to the effective time after which the request may be executed. An entry is created by `mint_to` on the request path (set to `now + delay`, the operational `Delay` if set, else `0`) and removed when the request is executed (the execute path of `mint_to`) or cancelled (`revoke_mint_request`). The presence of an entry is what distinguishes the two paths of `mint_to`: absent means the call records a new request, present means it executes the matching one (reverting with `TooEarlyToExecute` before the effective time). The entry TTL is extended to `MINT_REQUEST_TTL_LEDGERS` when recorded.
`Blocked`	`Set<Address>`	`persistent`	The set of blocked addresses. Membership is by key existence (a present entry always maps to `true`); `add_to_blocked_list` inserts an address and `remove_from_blocked_list` removes it (both operator-gated), and `is_blocked` queries membership. Blocking is a spend-side restriction: a blocked address cannot be the sender (`from`) in `transfer` or `transfer_from`, nor the `spender` in `transfer_from` — each such call raises `UserBlocked`. It does not otherwise restrict the address: a blocked address can still receive tokens (the recipient is never checked in `transfer` or `transfer_from`), be minted to via `mint_to`, have an allowance set over its tokens via `approve`, and have its balance moved by the operator via `forced_transfer` (which bypasses the block list by design).
`NewWasmHash`	`BytesN<32>`	`temporary`	The Wasm hash of the requested contract upgrade during a pending upgrade. Set by `request_upgrade` alongside `EtNextUpgrade`, and cleared when the upgrade is executed (`upgrade`) or cancelled (`revoke_next_upgrade`). On execution the supplied hash must match this pending value, else `upgrade` reverts with `InvalidWasmHash`, which also covers the no-pending-upgrade case (an unset slot matches no hash), so `upgrade` does not raise `NoPendingUpgrade`.
`EtNextUpgrade`	`u64`	`temporary`	Effective time (Unix seconds) after which a pending contract upgrade may be executed; the time-lock on upgrades. Set by `request_upgrade` to `now + delay` (the operational `Delay` if set, else `0`) alongside `NewWasmHash`, and cleared when the upgrade is executed (`upgrade`) or cancelled (`revoke_next_upgrade`). Until the ledger time passes it, `upgrade` reverts with `TooEarlyToExecute`. A non-zero reading means an upgrade is pending; `request_upgrade` reverts with `PendingRequestExists` if one already is.
`TotalSupply`	`i128`	`instance`	The total number of tokens in circulation. Increased by `mint_to` when a mint executes and decreased by `burn` when tokens are burned; no other function writes it. Never initialized at construction: until the first mint it reads as `0`.
`MintBudget`	`i128`	`instance`	The remaining authorized mint allowance: the number of tokens the operator may still mint. Decreased by `mint_to` when a mint executes (a mint exceeding it reverts with `MintBudgetNotEnough`), increased by `burn` when tokens are burned, and adjusted up or down by the operator via `change_mint_budget` (an adjustment that would drive it below zero reverts with `MintBudgetNotEnough`). Never initialized at construction: until first set it reads as `0`.
`Owner`	`Address`	`instance`	Address of the current contract owner, the primary privileged role. Set at construction by `__constructor` and thereafter only by `accept_owner` when a transfer completes. The owner configures the operational and governance delays (`set_delay`, `set_gov_delay`) and the operator and revoker roles (`set_operator`, `set_revoker`), force-transfers tokens bypassing the block list (`forced_transfer`), and initiates and revokes the time-locked contract upgrades (`request_upgrade`, `revoke_next_upgrade`, then `upgrade`) and ownership transfers (`request_owner_transfer`, `revoke_next_owner`). It also revokes pending revoker and gov-delay changes (`revoke_next_revoker`, `revoke_next_gov_delay`).
`PendingOwner`	`Address`	`temporary`	Address of the requested new owner during a pending ownership transfer. Set by `request_owner_transfer` alongside `EtNextOwner`, and cleared when the transfer completes (`accept_owner`) or is cancelled (`revoke_next_owner`). Its sole privilege: once the time-lock has elapsed it executes the transfer by calling `accept_owner` (the current `Owner` cannot execute it), becoming the new owner.
`EtNextOwner`	`u64`	`temporary`	Effective time (Unix seconds) after which a pending ownership transfer may be executed; the time-lock on ownership transfers. Set by `request_owner_transfer` to `now + gov_delay` (the governance `GovDelay` if set, else `0`) alongside `PendingOwner`, and cleared when the transfer completes (`accept_owner`) or is cancelled (`revoke_next_owner`). Until the ledger time passes it, `accept_owner` reverts with `TooEarlyToExecute`. A non-zero reading means a transfer is pending; a `0` (or unset) reading means none is.
`Delay`	`u64`	`instance`	The operational time-lock delay, in seconds: the waiting period between requesting and executing an operator change (`set_operator`), a revoker change (`set_revoker`), a delay change (`set_delay`), an upgrade (`request_upgrade` then `upgrade`), and a mint (`mint_to`). Set only by `set_delay` when a change completes, and never initialized at construction — until first set it reads as `0`, so those time-locks are initially inactive (a request may be executed in the same ledger). When set, it is between `MIN_DELAY` and `MAX_DELAY`.
`NextDelay`	`u64`	`temporary`	The requested new operational delay, in seconds, during a pending delay change. Set by `set_delay` on the request path alongside `EtNextDelay`, and cleared when the change is committed (the execute path of `set_delay`) or cancelled (`revoke_next_delay`). On the execute path the supplied delay must match this pending value, else `set_delay` reverts with `PendingRequestExists`. When set, it lies between `MIN_DELAY` and `MAX_DELAY` (the bound is checked by `set_delay` before recording the request).
`EtNextDelay`	`u64`	`temporary`	Effective time (Unix seconds) after which a pending delay change may be executed; the time-lock on operational delay changes. Set by `set_delay` on the request path to `now + delay` (the current operational `Delay` if set, else `0`) alongside `NextDelay`, and cleared when the change is committed (the execute path of `set_delay`) or cancelled (`revoke_next_delay`). Until the ledger time passes it, the execute path of `set_delay` reverts with `TooEarlyToExecute`. It also drives the request/execute dispatch: a `0` (or unset) reading takes the request path, a non-zero reading the execute path.
`Operator`	`Address`	`instance`	Address of the current operator, the role responsible for minting and redemption. Set at construction by `__constructor` and thereafter only by `set_operator` when a change completes. The operator mints tokens through the time-locked request/execute flow (`mint_to`), burns tokens from its own balance on behalf of redeeming customers (`burn`), adjusts the mint budget (`change_mint_budget`), revokes pending mint requests (`revoke_mint_request`), and manages the block list (`add_to_blocked_list`, `remove_from_blocked_list`).

`NextOperator`	`Address`	`temporary`	Address of the requested new operator during a pending operator change. Set by `set_operator` on the request path alongside `EtNextOperator`, and cleared when the change is committed (the execute path of `set_operator`) or cancelled (`revoke_next_operator`). On the execute path the supplied operator must match this pending value, else `set_operator` reverts with `PendingRequestExists`.
`EtNextOperator`	`u64`	`temporary`	Effective time (Unix seconds) after which a pending operator change may be executed; the time-lock on operator changes. Set by `set_operator` on the request path to `now + delay` (the operational `Delay` if set, else `0`) alongside `NextOperator`, and cleared when the change is committed (the execute path of `set_operator`) or cancelled (`revoke_next_operator`). Until the ledger time passes it, the execute path of `set_operator` reverts with `TooEarlyToExecute`. It also drives the request/execute dispatch: a `0` (or unset) reading takes the request path, a non-zero reading the execute path.
`Revoker`	`Address`	`instance`	Address of the current revoker, a narrowly scoped role whose only privilege is to cancel pending operational changes: it revokes a pending operator change (`revoke_next_operator`) or a pending delay change (`revoke_next_delay`). Set at construction by `__constructor` and thereafter only by `set_revoker` when a change completes. Pending revoker changes themselves are revoked by the `Owner` (`revoke_next_revoker`), not the revoker.
`NextRevoker`	`Address`	`temporary`	Address of the requested new revoker during a pending revoker change. Set by `set_revoker` on the request path alongside `EtNextRevoker`, and cleared when the change is committed (the execute path of `set_revoker`) or cancelled (`revoke_next_revoker`). On the execute path the supplied revoker must match this pending value, else `set_revoker` reverts with `PendingRequestExists`.
`EtNextRevoker`	`u64`	`temporary`	Effective time (Unix seconds) after which a pending revoker change may be executed; the time-lock on revoker changes. Set by `set_revoker` on the request path to `now + delay` (the operational `Delay` if set, else `0`) alongside `NextRevoker`, and cleared when the change is committed (the execute path of `set_revoker`) or cancelled (`revoke_next_revoker`). Until the ledger time passes it, the execute path of `set_revoker` reverts with `TooEarlyToExecute`. It also drives the request/execute dispatch: a `0` (or unset) reading takes the request path, a non-zero reading the execute path.
`GovDelay`	`u64`	`instance`	The governance time-lock delay, in seconds: the waiting period between requesting and executing a governance delay change (`set_gov_delay`) and an ownership transfer (`request_owner_transfer` then `accept_owner`). Set only by `set_gov_delay` when a change completes, and never initialized at construction — until first set it reads as `0`, so those time-locks are initially inactive (a request may be executed in the same ledger). When set, it is between `MIN_DELAY` and `MAX_DELAY`.
`NextGovDelay`	`u64`	`temporary`	The requested new governance delay, in seconds, during a pending governance delay change. Set by `set_gov_delay` on the request path alongside `EtNextGovDelay`, and cleared when the change is committed (the execute path of `set_gov_delay`) or cancelled (`revoke_next_gov_delay`). On the execute path the supplied governance delay must match this pending value, else `set_gov_delay` reverts with `PendingRequestExists`. When set, it lies between `MIN_DELAY` and `MAX_DELAY` (the bound is checked by `set_gov_delay` before recording the request).
`EtNextGovDelay`	`u64`	`temporary`	Effective time (Unix seconds) after which a pending governance delay change may be executed; the time-lock on governance delay changes. Set by `set_gov_delay` on the request path to `now + gov_delay` (the current governance `GovDelay` if set, else `0`) alongside `NextGovDelay`, and cleared when the change is committed (the execute path of `set_gov_delay`) or cancelled (`revoke_next_gov_delay`). Until the ledger time passes it, the execute path of `set_gov_delay` reverts with `TooEarlyToExecute`. It also drives the request/execute dispatch: a `0` (or unset) reading takes the request path, a non-zero reading the execute path.
`Metadata`	`TokenMetadata`	`instance`	The token metadata: its display name, symbol, and decimal precision, held in the `soroban-token-sdk` `TokenMetadata` structure and surfaced by the `name`, `symbol`, and `decimals` getters. Written once by `__constructor` (which rejects a decimal above 18 with `InvalidDecimal`) and never modified afterward, so the name, symbol, and decimals are immutable for the life of the contract.

Events

Event	Fields	Description
`UpgradeRequested`	`owner: Address` — The owner who requested the upgrade and authorized the call. `new_wasm_hash: BytesN<32>` — The proposed new contract Wasm hash. `effective_time: u64` — The earliest time at which the upgrade may be executed.	Emitted by `request_upgrade` when a contract upgrade is requested, recording the proposed Wasm hash and its effective time.
`ContractUpgraded`	`owner: Address` — The owner who executed the upgrade and authorized the call. `new_wasm_hash: BytesN<32>` — The Wasm hash the contract was upgraded to.	Emitted by `upgrade` when a pending upgrade is executed and the contract Wasm is replaced.
`UpgradeRevoked`	`owner: Address` — The owner who revoked the pending upgrade and authorized the call. `new_wasm_hash: BytesN<32>` — The Wasm hash of the upgrade that was pending and has now been cancelled.	Emitted by `revoke_next_upgrade` when a pending upgrade is cancelled.
`OwnerTransferRequested`	`owner: Address` — The current owner who requested the transfer and authorized the call. `pending_owner: Address` — The proposed new owner. `effective_time: u64` — The earliest time at which the transfer may be accepted.	Emitted by `request_owner_transfer` when an ownership transfer is requested, recording the proposed new owner and its effective time.
`OwnerTransferred`	`old_owner: Address` — The owner prior to the transfer. `new_owner: Address` — The owner after the transfer, who authorized it.	Emitted by `accept_owner` when a pending ownership transfer is executed.
`OwnerRevoked`	—	Emitted by `revoke_next_owner` when a pending ownership transfer is cancelled.
`SetOperatorRequest`	`current_operator: Address` — The operator at the time of the request. `next_operator: Address` — The proposed new operator. `effective_time: u64` — The earliest time at which the change may be executed.	Emitted by `set_operator` on the request path when an operator change is requested, recording the proposed operator and its effective time.
`SetOperatorEffected`	`operator: Address` — The new operator.	Emitted by `set_operator` on the execute path when a pending operator change is committed.
`OperatorRevoked`	—	Emitted by `revoke_next_operator` when a pending operator change is cancelled.
`SetRevokerRequest`	`current_revoker: Address` — The revoker at the time of the request. `next_revoker: Address` — The proposed new revoker. `effective_time: u64` — The earliest time at which the change may be executed.	Emitted by `set_revoker` on the request path when a revoker change is requested, recording the proposed revoker and its effective time.
`SetRevokerEffected`	`revoker: Address` — The new revoker.	Emitted by `set_revoker` on the execute path when a pending revoker change is committed.
`RevokerRevoked`	—	Emitted by `revoke_next_revoker` when a pending revoker change is cancelled.
`SetDelayRequest`	`current_delay: u64` — The operational delay at the time of the request, in seconds. `next_delay: u64` — The proposed new operational delay, in seconds. `effective_time: u64` — The earliest time at which the change may be executed.	Emitted by `set_delay` on the request path when a delay change is requested, recording the proposed delay and its effective time.
`SetDelayEffected`	`delay: u64` — The new operational delay, in seconds.	Emitted by `set_delay` on the execute path when a pending delay change is committed.
`DelayRevoked`	—	Emitted by `revoke_next_delay` when a pending delay change is cancelled.
`SetGovDelayRequest`	`current_delay: u64` — The governance delay at the time of the request, in seconds. `next_delay: u64` — The proposed new governance delay, in seconds. `effective_time: u64` — The earliest time at which the change may be executed.	Emitted by `set_gov_delay` on the request path when a governance delay change is requested, recording the proposed governance delay and its effective time.

`SetGovDelayEffected`	`delay: u64` — The new governance delay, in seconds.	Emitted by `set_gov_delay` on the execute path when a pending governance delay change is committed.
`GovDelayRevoked`	—	Emitted by `revoke_next_gov_delay` when a pending governance delay change is cancelled.
`ChangeMintBudget`	`delta: i128` — The signed change applied to `MintBudget`; positive increases it, negative decreases it.	Emitted by `change_mint_budget` when the operator adjusts the mint budget.
`MintRequest`	`receiver: Address` *(topic)* — The address that will receive the minted tokens if the request is executed. `amount: i128` — The amount to be minted. `nonce: u64` — The caller-supplied nonce distinguishing this request from otherwise-identical ones. `et: u64` — The effective time: the earliest time at which the request may be executed.	Emitted by `mint_to` on the request path when a mint request is recorded, before it can be executed.
`MintEffected`	`receiver: Address` *(topic)* — The address that received the minted tokens. `amount: i128` — The amount minted. `nonce: u64` — The nonce of the executed request.	Emitted by `mint_to` on the execute path when a pending mint request is executed and the tokens are minted.
`MintRequestRevoked`	`req: BytesN<32>` — The request id of the cancelled mint request (the `sha256` of the receiver, amount, and nonce).	Emitted by `revoke_mint_request` when a pending mint request is cancelled.
`Redeem`	`customer: Address` *(topic)* — The customer on whose behalf the redemption is recorded (the `from` argument). The tokens are burned from the operator balance, not from this address. `amount: i128` — The amount redeemed (and burned from the operator balance).	Emitted by `burn` when tokens are burned on behalf of a redeeming customer, recording the redemption against that customer.
`ForcedTransfer`	`from: Address` *(topic)* — The address the tokens were seized from. `to: Address` *(topic)* — The address the tokens were moved to. `amount: i128` — The amount transferred. `data: String` — Caller-supplied data passed through with the transfer (`data`). `extra_data: String` — Caller-supplied data passed through with the transfer (`extra_data`).	Emitted by `forced_transfer` when the owner forcibly moves tokens between two addresses, bypassing the block list.
`BlockPlaced`	`user: Address` *(topic)* — The address that was added to the block list.	Emitted by `add_to_blocked_list` when an address is added to the block list.
`BlockReleased`	`user: Address` *(topic)* — The address that was removed from the block list.	Emitted by `remove_from_blocked_list` when an address is removed from the block list.
`Transfer`	`from: Address` *(topic)* — The address the tokens moved from. `to: Address` *(topic)* — The address the tokens moved to. `to_muxed_id: Option<u64>` — The multiplexing id of the recipient, set only by `transfer` from a muxed destination; `transfer_from` and `forced_transfer` always leave it `None`. `amount: i128` — The amount transferred.	The standard `soroban-token-sdk transfer` event, emitted by `transfer`, `transfer_from`, and `forced_transfer` when tokens move between two addresses.
`Approve`	`from: Address` *(topic)* — The address that owns the tokens and granted the allowance. `spender: Address` *(topic)* — The address granted permission to spend on behalf of `from`. `amount: i128` — The allowance amount that was set. `expiration_ledger: u32` — The ledger sequence through which the allowance remains valid.	The standard `soroban-token-sdk` approval event, emitted by `approve` when an allowance is set.

`Burn`	`from: Address` *(topic)* — The address the tokens were burned from, which is the operator (not the redeeming customer; the customer is recorded in `Redeem`). `amount: i128` — The amount burned.	The standard `soroban-token-sdk` burn event, emitted by `burn` alongside `Redeem` when tokens are burned.
`MintWithAmountOnly`	`to: Address` *(topic)* — The address that received the minted tokens. `amount: i128` — The amount minted.	The standard `soroban-token-sdk` mint event, emitted by `mint_to` on the execute path alongside `MintEffected` when tokens are minted.

Errors

Error	Code	Description
`NegativeAmountNotAllowed`	`1`	Raised by `approve`, `transfer`, `transfer_from`, `forced_transfer`, `mint_to`, or `burn` when the caller-supplied amount is negative. Zero is permitted; only values below zero are rejected.
`NotOwner`	`10`	Not raised by any function in this contract.
`NotOperator`	`11`	Not raised by any function in this contract.
`NotRevoker`	`12`	Not raised by any function in this contract.
`NoPendingOwner`	`13`	Raised by `accept_owner` when there is no pending ownership transfer to accept.
`DelayTooSmall`	`20`	Raised by `set_delay` or `set_gov_delay` when the caller-supplied new delay is below `MIN_DELAY`. The bound is checked on every call, before the request/execute dispatch.
`TooEarlyToExecute`	`21`	Raised by `set_operator`, `set_revoker`, `set_delay`, `set_gov_delay`, `mint_to`, `upgrade`, or `accept_owner` when a pending request is executed before its effective time: the current ledger time has not passed the effective time recorded when the request was made.
`PendingRequestExists`	`22`	Raised when a change is requested while one is already pending. In `request_upgrade` and `request_owner_transfer` any pending request blocks a new one (`EtNextUpgrade`, respectively `EtNextOwner`, is non-zero). In `set_operator`, `set_revoker`, `set_delay`, and `set_gov_delay` it is raised on the request path when a request is pending for a value other than the one supplied (supplying the pending value instead dispatches to the execute path).
`DelayTooLarge`	`23`	Raised by `set_delay` or `set_gov_delay` when the caller-supplied new delay exceeds `MAX_DELAY`. The bound is checked on every call, before the request/execute dispatch.
`MintBudgetNotEnough`	`30`	Raised by `mint_to` on the execute path when `amount` exceeds `MintBudget`, and by `change_mint_budget` when the resulting budget would be negative (the prior `MintBudget` plus `delta` is below zero). In both cases `MintBudget` would otherwise drop below zero.
`UserBlocked`	`40`	Raised by `transfer` when `from` is on the block list, and by `transfer_from` when `spender` or `from` is on the block list. Blocking is spend-side only: a blocked address can still receive tokens, be minted to, have an allowance set over its tokens, and be moved by `forced_transfer` (which bypasses the block list).
`InsufficientAllowance`	`50`	Raised by `transfer_from` when `amount` is positive and exceeds the live allowance of `spender` over `from` (an absent or expired allowance reads as zero, so any positive amount exceeds it).
`InvalidExpirationLedger`	`51`	Raised by `approve` when `amount` is positive and `expiration_ledger` is below the current ledger sequence (a live allowance may not be set to expire in the past). For a zero amount the expiration is unconstrained.
`InsufficientBalance`	`60`	Raised by `transfer`, `transfer_from`, or `forced_transfer` when the amount to move exceeds the sender balance, and by `burn` when the amount to burn exceeds the operator balance. The debited account must hold at least the amount before its balance is reduced.
`InvalidDecimal`	`70`	Raised by `__constructor` when the supplied decimal precision exceeds 18.
`InvalidWasmHash`	`71`	Raised by `upgrade` when `new_wasm_hash` does not match the pending upgrade hash in `NewWasmHash`. This covers both a mismatched hash and the absence of any pending upgrade (an unset `NewWasmHash` cannot equal the supplied hash), so `upgrade` does not separately raise `NoPendingUpgrade`.
`NoPendingUpgrade`	`72`	Raised by `revoke_next_upgrade` when there is no pending upgrade to revoke (`NewWasmHash` is unset). This is the only function that raises it; `upgrade` folds the no-pending case into `InvalidWasmHash` instead.
`TotalSupplyUnderflow`	`81`	Raised by `burn` when the amount to burn exceeds `TotalSupply`. Defensive and not expected to trigger: by `total_supply_eq_sum_of_balances` the operator balance is at most `TotalSupply`, so the prior `InsufficientBalance` check (amount at most the operator balance) already guarantees the amount is at most `TotalSupply`.
`NotSupported`	`90`	Raised by `burn_from`, which is an unsupported stub: every call reverts unconditionally with this error. The standard `TokenInterface::burn_from` is intentionally not implemented by this token.

Functions

`__constructor`

Categories	`constructor`
Authorization	—
Parameters	`env: Env` `owner: Address` — Initial address of the contract owner. `operator: Address` — Initial address of the operator. `revoker: Address` — Initial address of the revoker. `decimal: u32` — Number of decimals used to represent token amounts. Must be at most `18`. `name: String` — Token name. `symbol: String` — Token symbol.
Returns	`()`
Reads	—
Mutates	`Owner` — Set to `owner`. `Operator` — Set to `operator`. `Revoker` — Set to `revoker`. `Metadata` — Set to a `TokenMetadata` with the given `decimal`, `name`, and `symbol`.
Raises	`InvalidDecimal` — If `decimal` > 18.
Emits	—

Initializes the contract with an owner, operator, revoker, and token metadata (decimals, name, symbol).

`request_upgrade`

Categories	`upgrade`
Authorization	`Owner`
Parameters	`env: Env` `new_wasm_hash: BytesN<32>` — Hash of the new contract's Wasm bytecode to upgrade to.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `EtNextUpgrade` — To check that no upgrade request is already pending. `Delay` — To compute the effective time.
Mutates	`NewWasmHash` — Set to `new_wasm_hash`. Extends the TTL to `PENDING_TTL_LEDGERS` (if it would otherwise expire sooner). `EtNextUpgrade` — Set to `now + delay`, where `delay` is `Delay` if set and `0` otherwise. Extends the TTL to `PENDING_TTL_LEDGERS` (if it would otherwise expire sooner).
Raises	`PendingRequestExists` — If `EtNextUpgrade` is set to a non-zero value.
Emits	`UpgradeRequested` — On success.

Requests a time-locked contract upgrade. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`pending_upgrade_established`	On success, a pending upgrade exists: `NewWasmHash` equals `new_wasm_hash` and `EtNextUpgrade` equals `now + delay`, where `delay` is `Delay` if set and `0` otherwise.	After the `PendingRequestExists` guard rules out an already-pending request, the function writes `new_wasm_hash` to `NewWasmHash` and `now + delay` to `EtNextUpgrade` before returning.

`upgrade`

Categories	`upgrade`
Authorization	`Owner`
Parameters	`env: Env` `new_wasm_hash: BytesN<32>` — Hash of the new contract's Wasm bytecode to upgrade to.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `NewWasmHash` — To verify it is set to `new_wasm_hash`. `EtNextUpgrade` — To check that it is set to a timestamp lower than the current one.
Mutates	`NewWasmHash` — Unset after the upgrade is executed. `EtNextUpgrade` — Unset after the upgrade is executed.
Raises	`InvalidWasmHash` — If `NewWasmHash` is not set to `new_wasm_hash`. `TooEarlyToExecute` — If `EtNextUpgrade` is set to a timestamp at least the current one.
Emits	`ContractUpgraded` — On success.

Executes a pending time-locked contract upgrade, replacing the contract's Wasm bytecode with `NewWasmHash`. Clears the pending upgrade state on success.

Calls

Soroban host (`env.deployer()`)	`update_current_contract_wasm`	Replaces the contract's Wasm bytecode with `new_wasm_hash`. Takes effect after the current invocation completes: the executing code finishes running as the old version.

Postconditions

Name	Property	Proof
`timelock_respected`	On success, the current ledger timestamp is strictly greater than the effective time that was stored in `EtNextUpgrade`.	Guarded by the `TooEarlyToExecute` check. The slot is necessarily set at that point: the `InvalidWasmHash` check has already passed, so `NewWasmHash` is set, and `pending_upgrade_consistent` then implies `EtNextUpgrade` is set.
`wasm_replaced`	On success, the contract's Wasm bytecode is replaced with `new_wasm_hash`, taking effect after the current invocation completes.	Via `update_current_contract_wasm`; see the corresponding calls entry.
`pending_upgrade_cleared`	On success, `NewWasmHash` and `EtNextUpgrade` are unset.	Both are removed unconditionally before the function returns.

`revoke_next_upgrade`

Categories	`upgrade`
Authorization	`Owner`
Parameters	`env: Env`
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `NewWasmHash` — To verify a pending upgrade exists.
Mutates	`NewWasmHash` — Unset on success. `EtNextUpgrade` — Unset on success.
Raises	`NoPendingUpgrade` — If `NewWasmHash` is not set.
Emits	`UpgradeRevoked` — On success.

Cancels a pending contract upgrade. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`pending_upgrade_cleared`	On success, `NewWasmHash` and `EtNextUpgrade` are unset.	After the `NoPendingUpgrade` guard (which requires `NewWasmHash` to be set) passes, both slots are removed unconditionally before returning.

`request_owner_transfer`

Categories	`ownership`
Authorization	`Owner`
Parameters	`env: Env` `new_owner: Address` — Address of the requested new owner.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `EtNextOwner` — To verify that no ownership transfer is already pending. `GovDelay` — To compute the effective time.
Mutates	`PendingOwner` — Set to `new_owner`. Extends the TTL to `PENDING_TTL_LEDGERS` (if it would otherwise expire sooner). `EtNextOwner` — Set to `now + gov_delay`, where `gov_delay` is `GovDelay` if set and `0` otherwise. Extends the TTL to `PENDING_TTL_LEDGERS` (if it would otherwise expire sooner).
Raises	`PendingRequestExists` — If `EtNextOwner` is set to a non-zero value.
Emits	`OwnerTransferRequested` — On success.

Requests a time-locked ownership transfer. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`pending_transfer_established`	On success, a pending ownership transfer exists: `PendingOwner` equals `new_owner` and `EtNextOwner` equals `now + gov_delay`, where `gov_delay` is `GovDelay` if set and `0` otherwise.	After the `PendingRequestExists` guard rules out an already-pending transfer, the function writes `new_owner` to `PendingOwner` and `now + gov_delay` (the governance delay: `GovDelay` if set and `0` otherwise) to `EtNextOwner` before returning.

`accept_owner`

Categories	`ownership`
Authorization	`PendingOwner`
Parameters	`env: Env`
Returns	`()`
Reads	`PendingOwner` — To verify a transfer is pending, to authenticate the caller as the pending owner, to set `Owner`, and to set as the `new_owner` field of the emitted `OwnerTransferred`. `EtNextOwner` — To check that the time-lock has elapsed (i.e. it is set to a timestamp lower than the current one). `Owner` — To set as the `old_owner` field of the emitted `OwnerTransferred`.
Mutates	`Owner` — Set to `PendingOwner` (the accepted new owner). `PendingOwner` — Unset on success. `EtNextOwner` — Unset on success.
Raises	`NoPendingOwner` — If `PendingOwner` is not set. `TooEarlyToExecute` — If `EtNextOwner` is set to a timestamp at least the current one.
Emits	`OwnerTransferred` — On success.

Executes a pending time-locked ownership transfer, setting `Owner` to the pending owner. Clears the pending transfer state on success. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`timelock_respected`	On success, the current ledger timestamp is strictly greater than the effective time that was stored in `EtNextOwner`.	Guarded by the `TooEarlyToExecute` check. The slot is necessarily set at that point: the `NoPendingOwner` check has already passed, so `PendingOwner` is set, and `pending_owner_consistent` then implies `EtNextOwner` is set.
`ownership_transferred`	On success, `Owner` is set to the address that was stored in `PendingOwner`.	Set before the function returns.
`pending_transfer_cleared`	On success, `PendingOwner` and `EtNextOwner` are unset.	Both are removed unconditionally before the function returns.

`revoke_next_owner`

Categories	`ownership`
Authorization	`Owner`
Parameters	`env: Env`
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	`PendingOwner` — Unset. `EtNextOwner` — Unset.
Raises	—
Emits	`OwnerRevoked` — On success.

Cancels a pending ownership transfer, unconditionally clearing the pending transfer state. Does not require a transfer to actually be pending.

Postconditions

Name	Property	Proof
`pending_transfer_cleared`	On success, `PendingOwner` and `EtNextOwner` are unset.	Both are unset unconditionally.

`set_operator`

Categories	`configuration`
Authorization	`Owner`
Parameters	`env: Env` `new_operator: Address` — The proposed new operator. On the request path it is recorded as pending; on the execute path it must match the pending request and becomes the new `Operator`.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `EtNextOperator` — To determine whether a request is pending (non-zero) and, on the execute path, whether its effective time has elapsed. `NextOperator` — On the execute path, to verify the pending request is for `new_operator`. `Operator` — To record the current operator in the emitted `SetOperatorRequest` on the request path. `Delay` — To compute the effective time.
Mutates	`Operator` — On the execute path, set to `new_operator`. `NextOperator` — On the request path, set to `new_operator` (TTL extended to `PENDING_TTL_LEDGERS`); on the execute path, unset. `EtNextOperator` — On the request path, set to `now + delay`, where `delay` is `Delay` if set and `0` otherwise (TTL extended to `PENDING_TTL_LEDGERS`); on the execute path, unset.
Raises	`PendingRequestExists` — If a request is already pending (`EtNextOperator` non-zero) for an operator other than `new_operator`. `TooEarlyToExecute` — If a request for `new_operator` is pending but its effective time `EtNextOperator` has not yet elapsed (`>= now`).
Emits	`SetOperatorRequest` — On the request path. `SetOperatorEffected` — On the execute path.

Sets the operator through a two-step, time-locked process, dispatching on whether a request is already pending (indicated by a non-zero `EtNextOperator`). On the request path, i.e. with no request pending, it records `new_operator` as the pending operator

with effective time `now + delay` and emits `SetOperatorRequest`. On the execute path, i.e. with a request already pending for `new_operator` whose effective time has passed, it commits the change (sets `Operator`), clears the pending state, and emits `SetOperatorEffected`. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`request_recorded`	If `EtNextOperator` is unset or `0` at entry, then on success `NextOperator == new_operator`, `EtNextOperator == now + delay` (where `delay` is `Delay` if set and `0` otherwise), and `Operator` is unchanged.	The function dispatches on `read_et_next_operator()`, which yields `EtNextOperator` if set and `0` otherwise. An unset or `0` slot gives `0`, taking the request branch. The request branch is unconditional: it writes `new_operator` to `NextOperator` and `now + delay` to `EtNextOperator`, and does not touch `Operator`, before returning.
`change_committed`	If `EtNextOperator` is non-zero at entry, then on success `Operator == new_operator`, and `NextOperator` and `EtNextOperator` are unset.	The function dispatches on `read_et_next_operator()`, which yields `EtNextOperator` if set and `0` otherwise. A non-zero slot gives a non-zero value, taking the execute branch. By `next_operator_consistent` `NextOperator` is then also set, so reading the pending operator does not fault. The execute branch requires that pending `NextOperator == new_operator` (else `PendingRequestExists`) and the effective time to have elapsed (else `TooEarlyToExecute`); on success it writes `new_operator` to `Operator` and removes both `NextOperator` and `EtNextOperator` before returning.
`timelock_respected`	On success via the execute path (`EtNextOperator` non-zero at entry), the current ledger timestamp is strictly greater than the effective time that was stored in `EtNextOperator`.	The execute branch raises `TooEarlyToExecute` when `et >= now`, so success requires `et < now`: the current timestamp strictly exceeds the stored effective time.

`set_revoker`

Categories	`configuration`
Authorization	`Owner`
Parameters	`env: Env` `new_revoker: Address` — The proposed new revoker. On the request path it is recorded as pending; on the execute path it must match the pending request and becomes the new `Revoker`.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `EtNextRevoker` — To determine whether a request is pending (non-zero) and, on the execute path, whether its effective time has elapsed. `NextRevoker` — On the execute path, to verify the pending request is for `new_revoker`. `Revoker` — To record the current revoker in the emitted `SetRevokerRequest` on the request path. `Delay` — To compute the effective time.
Mutates	`Revoker` — On the execute path, set to `new_revoker`. `NextRevoker` — On the request path, set to `new_revoker` (TTL extended to `PENDING_TTL_LEDGERS`); on the execute path, unset. `EtNextRevoker` — On the request path, set to `now + delay`, where `delay` is `Delay` if set and `0` otherwise (TTL extended to `PENDING_TTL_LEDGERS`); on the execute path, unset.
Raises	`PendingRequestExists` — If a request is already pending (`EtNextRevoker` non-zero) for a revoker other than `new_revoker`. `TooEarlyToExecute` — If a request for `new_revoker` is pending but its effective time `EtNextRevoker` has not yet elapsed (`>= now`).
Emits	`SetRevokerRequest` — On the request path. `SetRevokerEffected` — On the execute path.

Sets the revoker through a two-step, time-locked process, dispatching on whether a request is already pending (indicated by a non-zero `EtNextRevoker`). On the request path, i.e. with no request pending, it records `new_revoker` as the pending revoker with effective time `now + delay` and emits `SetRevokerRequest`. On the execute path, i.e. with a request already pending for `new_revoker` whose effective time has passed, it commits the change (sets `Revoker`), clears the pending state, and emits `SetRevokerEffected`. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`request_recorded`	If `EtNextRevoker` is unset or `0` at entry, then on success `NextRevoker == new_revoker`, `EtNextRevoker == now + delay` (where `delay` is `Delay` if set and `0` otherwise), and `Revoker` is unchanged.	The function dispatches on `read_et_next_revoker()`, which yields `EtNextRevoker` if set and `0` otherwise. An unset or `0` slot gives `0`, taking the request branch. The request branch is unconditional: it writes `new_revoker` to `NextRevoker` and `now + delay` to `EtNextRevoker`, and does not touch `Revoker`, before returning.
`change_committed`	If `EtNextRevoker` is non-zero at entry, then on success `Revoker == new_revoker`, and `NextRevoker` and `EtNextRevoker` are unset.	The function dispatches on `read_et_next_revoker()`, which yields `EtNextRevoker` if set and `0` otherwise. A non-zero slot gives a non-zero value, taking the execute branch. By `next_revoker_consistent` `NextRevoker` is then also set, so reading the pending revoker does not fault. The execute branch requires that pending `NextRevoker == new_revoker` (else `PendingRequestExists`) and the effective time to have elapsed (else `TooEarlyToExecute`); on success it writes `new_revoker` to `Revoker` and removes both `NextRevoker` and `EtNextRevoker` before returning.
`timelock_respected`	On success via the execute path (`EtNextRevoker` non-zero at entry), the current ledger timestamp is strictly greater than the effective time that was stored in `EtNextRevoker`.	The execute branch raises `TooEarlyToExecute` when `et >= now`, so success requires `et < now`: the current timestamp strictly exceeds the stored effective time.

`set_delay`

Categories	`configuration`
Authorization	`Owner`
Parameters	`env: Env` `new_delay: u64` — The proposed new delay, in seconds. `MIN_DELAY <= new_delay <= MAX_DELAY` must hold. On the request path it is recorded as pending; on the execute path it must match the pending request and becomes the new `Delay`.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `EtNextDelay` — To determine whether a request is pending (non-zero) and, on the execute path, whether its effective time has elapsed. `NextDelay` — On the execute path, to verify the pending request is for `new_delay`. `Delay` — On the request path, to record the current delay in the emitted `SetDelayRequest` and to compute the effective time.
Mutates	`Delay` — On the execute path, set to `new_delay`. `NextDelay` — On the request path, set to `new_delay` (TTL extended to `PENDING_TTL_LEDGERS`); on the execute path, unset. `EtNextDelay` — On the request path, set to `now + delay`, where `delay` is `Delay` if set and `0` otherwise (TTL extended to `PENDING_TTL_LEDGERS`); on the execute path, unset.
Raises	`DelayTooSmall` — If `new_delay < MIN_DELAY`. `DelayTooLarge` — If `new_delay > MAX_DELAY`. `PendingRequestExists` — If a request is already pending (`EtNextDelay` non-zero) for a delay other than `new_delay`. `TooEarlyToExecute` — If a request for `new_delay` is pending but its effective time `EtNextDelay` has not yet elapsed (`>= now`).
Emits	`SetDelayRequest` — On the request path. `SetDelayEffected` — On the execute path.

Sets the delay through a two-step, time-locked process, dispatching on whether a request is already pending (indicated by a non-zero `EtNextDelay`). On every call, `MIN_DELAY <= new_delay <= MAX_DELAY` must hold. On the request path, i.e. with no request pending, it records `new_delay` as the pending delay with effective time `now + delay` (the current `Delay` if set or `0` otherwise) and emits `SetDelayRequest`. On the execute path, i.e. with a request already pending for `new_delay` whose effective time has passed, it commits the change (sets `Delay`), clears the pending state, and emits `SetDelayEffected`. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`request_recorded`	If `EtNextDelay` is unset or `0` at entry, then on success `NextDelay == new_delay` (with `MIN_DELAY <= new_delay <= MAX_DELAY`), `EtNextDelay == now + delay` (where `delay` is `Delay` if set and `0` otherwise), and `Delay` is unchanged.	Before dispatching, the function rejects `new_delay < MIN_DELAY` (`DelayTooSmall`) and `new_delay > MAX_DELAY` (`DelayTooLarge`), so success implies `new_delay` is within bounds. It then dispatches on `read_et_next_delay()` , which yields `EtNextDelay` if set and `0` otherwise. An unset or `0` slot gives `0` , taking the request branch, which is unconditional: it writes `new_delay` to `NextDelay` and `now + delay` to `EtNextDelay` , and does not touch `Delay` , before returning.
`change_committed`	If `EtNextDelay` is non-zero at entry, then on success `Delay == new_delay` (with `MIN_DELAY <= new_delay <= MAX_DELAY`), and `NextDelay` and `EtNextDelay` are unset.	Before dispatching, the function rejects `new_delay < MIN_DELAY` (`DelayTooSmall`) and `new_delay > MAX_DELAY` (`DelayTooLarge`), so success implies `new_delay` is within bounds. It then dispatches on `read_et_next_delay()` , which yields `EtNextDelay` if set and `0` otherwise. A non-zero slot gives a non-zero value, taking the execute branch. By `next_delay_consistent` , `NextDelay` is then also set, so reading the pending delay does not fault. The execute branch requires that pending `NextDelay == new_delay` (else `PendingRequestExists`) and the effective time to have elapsed (else `TooEarlyToExecute`); on success it writes `new_delay` to `Delay` and removes both `NextDelay` and `EtNextDelay` before returning.
`timelock_respected`	On success via the execute path (`EtNextDelay` non-zero at entry), the current ledger timestamp is strictly greater than the effective time that was stored in `EtNextDelay` .	The execute branch raises `TooEarlyToExecute` when `et >= now` , so success requires `et < now` : the current timestamp strictly exceeds the stored effective time.

`set_gov_delay`

Categories	`configuration`
Authorization	`Owner`
Parameters	`env: Env` `new_delay: u64` — The proposed new governance delay, in seconds. `MIN_DELAY <= new_delay <= MAX_DELAY` must hold. On the request path it is recorded as pending; on the execute path it must match the pending request and becomes the new `GovDelay` .
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `EtNextGovDelay` — To determine whether a request is pending (non-zero) and, on the execute path, whether its effective time has elapsed. `NextGovDelay` — On the execute path, to verify the pending request is for `new_delay` . `GovDelay` — On the request path, to record the current governance delay in the emitted `SetGovDelayRequest` and to compute the effective time.
Mutates	`GovDelay` — On the execute path, set to `new_delay` . `NextGovDelay` — On the request path, set to `new_delay` (TTL extended to `PENDING_TTL_LEDGERS`); on the execute path, unset. `EtNextGovDelay` — On the request path, set to `now + delay` , where `delay` is `GovDelay` if set and `0` otherwise (TTL extended to `PENDING_TTL_LEDGERS`); on the execute path, unset.
Raises	`DelayTooSmall` — If `new_delay < MIN_DELAY` . `DelayTooLarge` — If `new_delay > MAX_DELAY` . `PendingRequestExists` — If a request is already pending (`EtNextGovDelay` non-zero) for a governance delay other than `new_delay` . `TooEarlyToExecute` — If a request for `new_delay` is pending but its effective time `EtNextGovDelay` has not yet elapsed (`>= now`).
Emits	`SetGovDelayRequest` — On the request path. `SetGovDelayEffected` — On the execute path.

Sets the governance delay through a two-step, time-locked process, dispatching on whether a request is already pending (indicated by a non-zero `EtNextGovDelay`). On every call, `MIN_DELAY <= new_delay <= MAX_DELAY` must hold. On the request path, i.e. with no request pending, it records `new_delay` as the pending governance delay with effective time `now + delay` (the current `GovDelay` if set or `0` otherwise) and emits `SetGovDelayRequest` . On the execute path, i.e. with a request already pending for `new_delay` whose effective time has passed, it commits the change (sets `GovDelay`), clears the pending state, and emits `SetGovDelayEffected` . Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`request_recorded`	If `EtNextGovDelay` is unset or `0` at entry, then on success `NextGovDelay == new_delay` (with `MIN_DELAY <= new_delay <= MAX_DELAY`), `EtNextGovDelay == now + delay` (where `delay` is `GovDelay` if set and `0` otherwise), and `GovDelay` is unchanged.	Before dispatching, the function rejects `new_delay < MIN_DELAY` (`DelayTooSmall`) and `new_delay > MAX_DELAY` (`DelayTooLarge`), so success implies `new_delay` is within bounds. It then dispatches on `read_et_next_gov_delay()`, which yields `EtNextGovDelay` if set and `0` otherwise. An unset or `0` slot gives `0`, taking the request branch, which is unconditional: it writes `new_delay` to `NextGovDelay` and `now + delay` to `EtNextGovDelay`, and does not touch `GovDelay`, before returning.
`change_committed`	If `EtNextGovDelay` is non-zero at entry, then on success `GovDelay == new_delay` (with `MIN_DELAY <= new_delay <= MAX_DELAY`), and `NextGovDelay` and `EtNextGovDelay` are unset.	Before dispatching, the function rejects `new_delay < MIN_DELAY` (`DelayTooSmall`) and `new_delay > MAX_DELAY` (`DelayTooLarge`), so success implies `new_delay` is within bounds. It then dispatches on `read_et_next_gov_delay()`, which yields `EtNextGovDelay` if set and `0` otherwise. A non-zero slot gives a non-zero value, taking the execute branch. By `next_gov_delay_consistent`, `NextGovDelay` is then also set, so reading the pending governance delay does not fault. The execute branch requires that pending `NextGovDelay == new_delay` (else `PendingRequestExists`) and the effective time to have elapsed (else `TooEarlyToExecute`); on success it writes `new_delay` to `GovDelay` and removes both `NextGovDelay` and `EtNextGovDelay` before returning.
`timelock_respected`	On success via the execute path (`EtNextGovDelay` non-zero at entry), the current ledger timestamp is strictly greater than the effective time that was stored in `EtNextGovDelay`.	The execute branch raises `TooEarlyToExecute` when `et >= now`, so success requires `et < now`: the current timestamp strictly exceeds the stored effective time.

`revoke_next_gov_delay`

Categories	`configuration`
Authorization	`Owner`
Parameters	`env: Env`
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	`NextGovDelay` — Unset. `EtNextGovDelay` — Unset.
Raises	—
Emits	`GovDelayRevoked` — On success.

Cancels a pending governance delay change, unconditionally clearing the pending change state. Does not require a change to actually be pending.

Postconditions

Name	Property	Proof
`next_gov_delay_cleared`	On success, `NextGovDelay` and `EtNextGovDelay` are unset.	Both are unset unconditionally.

`forced_transfer`

Categories	`core`
Authorization	`Owner`
Parameters	`env: Env` `from: Address` — The address whose tokens are seized (the balance source). `to: Address` — The recipient address. `amount: i128` — The number of tokens to transfer. Must be non-negative. `data: String` — Free-form data recorded as `data` in the emitted `ForcedTransfer` event; does not affect contract state. `extra_data: String` — Free-form data recorded as `extra_data` in the emitted `ForcedTransfer` event; does not affect contract state.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `Balance` — To read the current balances of `from` and `to`.
Mutates	`Balance` — The balance of `from` is decreased by `amount` and the balance of `to` is increased by `amount`; both `Balance` entries are written and their persistent TTLs extended to `BALANCE_BUMP_AMOUNT`.
Raises	`NegativeAmountNotAllowed` — If `amount` < 0. `InsufficientBalance` — If `amount` exceeds the balance of `from`.
Emits	`Transfer` — On success (with `to_muxed_id` set to `None`; muxed destinations are not supported here). `ForcedTransfer` — On success.

Forcibly transfers `amount` tokens from `from` to `to` by owner authority, bypassing the block list (neither `from` nor `to` is checked). Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`balance_transferred_distinct`	Let `from != to`. Then on success, the balance of `from` decreases by `amount` and that of `to` increases by `amount` (each relative to its current value, counting an absent entry as `0`). Moreover, both addresses have an entry in `Balance` afterward.	After the `NegativeAmountNotAllowed` guard, `update_balance(Some(from), Some(to), amount)` runs `spend_balance` then `receive_balance`, with `from` and `to` distinct in this case. Each reads the relevant balance via `read_balance`, which returns the stored value if a `Balance` entry is present and `0` if absent. `spend_balance` checks `amount <= b_from` (else `InsufficientBalance`, where `b_from` is the read balance of `from`) and writes `b_from - amount` to `from`; `receive_balance` writes `b_to + amount` to `to`, where `b_to` is its read balance. Each `write_balance` unconditionally sets the entry (creating it if absent), so afterward `from` holds `b_from - amount` and `to` holds `b_to + amount`, both with entries. Being distinct, the two writes target different entries and do not interfere.
`self_transfer_unchanged`	Let `from == to`. Then on success, the balance of the address is unchanged. Moreover, the address has an entry in `Balance` afterward.	After the `NegativeAmountNotAllowed` guard, `update_balance(Some(from), Some(to), amount)` runs `spend_balance` then `receive_balance` on the one shared entry (since `from` and `to` coincide). `spend_balance` reads the balance `b` via `read_balance` (the stored value, or `0` if absent), checks `amount <= b` (else `InsufficientBalance`), and writes `b - amount`. `receive_balance` then re-reads that just-written value and writes `(b - amount) + amount = b`, restoring the original. These writes set the entry (creating it if absent), so afterward the address has an entry holding its original balance `b`.
`total_supply_unchanged`	On success, `TotalSupply` is unchanged.	Both sides of `update_balance` are present (`Some`), so neither the mint branch (`from = None`) nor the burn branch (`to = None`) of `spend_balance` / `receive_balance` runs; only `Balance` entries are written, never `TotalSupply`.

`revoke_next_delay`

Categories	`configuration`
Authorization	`Revoker`
Parameters	`env: Env`
Returns	`()`
Reads	`Revoker` — To authenticate the caller as the current revoker.
Mutates	`NextDelay` — Unset. `EtNextDelay` — Unset.
Raises	—
Emits	`DelayRevoked` — On success.

Cancels a pending delay change, unconditionally clearing the pending change state. Does not require a change to actually be pending.

Postconditions

Name	Property	Proof
`next_delay_cleared`	On success, `NextDelay` and `EtNextDelay` are unset.	Both are unset unconditionally.

`revoke_next_operator`

Categories	`configuration`
Authorization	`Revoker`
Parameters	`env: Env`
Returns	`()`
Reads	`Revoker` — To authenticate the caller as the current revoker.
Mutates	`NextOperator` — Unset. `EtNextOperator` — Unset.
Raises	—
Emits	`OperatorRevoked` — On success.

Cancels a pending operator change, unconditionally clearing the pending change state. Does not require a change to actually be pending.

Postconditions

Name	Property	Proof
`next_operator_cleared`	On success, `NextOperator` and `EtNextOperator` are unset.	Both are unset unconditionally.

`revoke_next_revoker`

Categories	`configuration`
Authorization	`Owner`
Parameters	`env: Env`
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	`NextRevoker` — Unset. `EtNextRevoker` — Unset.
Raises	—
Emits	`RevokerRevoked` — On success.

Cancels a pending revoker change, unconditionally clearing the pending change state. Does not require a change to actually be pending.

Postconditions

Name	Property	Proof
`next_revoker_cleared`	On success, `NextRevoker` and `EtNextRevoker` are unset.	Both are unset unconditionally.

`change_mint_budget`

Categories	`core`
Authorization	`Operator`
Parameters	`env: Env` `delta: i128` — Signed amount to add to the current mint budget; negative values decrease it.
Returns	`()`
Reads	`Operator` — To authenticate the caller as the current operator. `MintBudget` — Its current value is the base for the `delta` adjustment.
Mutates	`MintBudget` — Set to its prior value plus `delta`.
Raises	`MintBudgetNotEnough` — If the resulting budget (`MintBudget + delta`) would be negative.
Emits	`ChangeMintBudget` — On success.

Adjusts the mint budget by a signed `delta` (positive to increase, negative to decrease). Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`mint_budget_updated`	On success, `MintBudget` equals its prior value plus `delta`, and is `>= 0`.	The function computes `new_budget = MintBudget + delta` (the exact sum, by `assume_no_overflow`), rejects `new_budget < 0` with `MintBudgetNotEnough`, then writes `new_budget`. Non-negativity of the result follows from that guard, consistent with `mint_budget_nonneg`.

`mint_to`

Categories	`core`
Authorization	`Operator`
Parameters	`env: Env` `receiver: Address` — The address to receive the minted tokens. Part of the request id. `amount: i128` — The number of tokens to mint. Must be non-negative. Part of the request id. `nonce: u64` — A caller-chosen value distinguishing otherwise-identical mint requests (same receiver and amount). Part of the request id.
Returns	`bool` — `false` if the call registered a new request; `true` if it executed a pending one.
Reads	`Operator` — To authenticate the caller as the current operator. `MintRequest` — To determine whether a request for the computed id `r` is pending and, if so, its effective time. `Delay` — On the request path, to compute the effective time. `MintBudget` — On the execute path, to check it covers `amount` and to compute the new budget. `TotalSupply` — On the execute path, to compute the new total supply. `Balance` — On the execute path, to compute the new balance of `receiver`.
Mutates	`MintRequest` — On the request path, the entry for the request id `r` is set to `now + delay` (TTL extended to `MINT_REQUEST_TTL_LEDGERS`). On the execute path, the entry for `r` is removed. `MintBudget` — On the execute path, decreased by `amount`. `TotalSupply` — On the execute path, increased by `amount`. `Balance` — On the execute path, the balance of `receiver` is increased by `amount` (entry written, TTL extended to `BALANCE_BUMP_AMOUNT`).
Raises	`NegativeAmountNotAllowed` — If `amount` < 0. `TooEarlyToExecute` — On the execute path, if the pending request's effective time has not yet elapsed (`>= now`). `MintBudgetNotEnough` — On the execute path, if `amount` exceeds `MintBudget`.
Emits	`MintRequest` — On the request path. `MintWithAmountOnly` — On the execute path. `MintEffected` — On the execute path.

Mints `amount` tokens to `receiver` through a two-step, time-locked process keyed by a request id `r` derived from the arguments: `r` is the `sha256` of `receiver`'s XDR encoding concatenated with `amount` (16-byte big-endian) and `nonce` (8-byte big-endian). On the request path, i.e. on the first call for a given `r` (no `MintRequest` entry), it registers the request with effective time `now + delay`, emits `MintRequest`, and returns `false`. On the execute path, i.e. a later call with the same arguments whose effective time has passed, it checks and decrements `MintBudget`, mints to `receiver` (increasing `TotalSupply` and the receiver balance), clears the request, emits `MintWithAmountOnly` and `MintEffected`, and returns `true`. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`request_recorded`	Let `r` be the request id (the `sha256` of `receiver`'s XDR encoding, `amount` as 16-byte big-endian, and `nonce` as 8-byte big-endian). If `MintRequest` has no entry for `r` at entry, then the call returns `false`, sets the `MintRequest` entry for `r` to `now + delay` (where `delay` is `Delay` if set and `0` otherwise), and leaves `MintBudget`, `TotalSupply`, and all balances unchanged.	After the `NegativeAmountNotAllowed` guard, the function computes `r` and dispatches on `read_mint_request(r)`. A `None` result (no entry) takes the request branch, which is unconditional: it writes `now + delay` to the `MintRequest` entry for `r`, emits `MintRequest`, and returns `false`, without touching `MintBudget`, `TotalSupply`, or any `Balance` entry.
`mint_executed`	Let `r` be the request id (as above). If `MintRequest` maps `r` to some `et` at entry, then on success the call returns `true`, removes the `MintRequest` entry for `r`, decreases `MintBudget` by `amount`, increases `TotalSupply` by `amount`, and increases the balance of `receiver` by `amount` (relative to its current value, counting an absent entry as `0`).	A `Some(et)` result from `read_mint_request(r)` takes the execute branch: it requires `et < now` (else `TooEarlyToExecute`), removes the `MintRequest` entry for `r`, requires `amount <= MintBudget` (else `MintBudgetNotEnough`) and writes `MintBudget - amount`, then calls `update_balance(None, Some(receiver), amount)`. With `from = None` the mint branch of `spend_balance` writes `TotalSupply + amount`, and `receive_balance` writes the receiver balance `b + amount` (where `b` is read via `read_balance`, `0` if absent), setting the entry if absent. It then emits `MintWithAmountOnly` and `MintEffected` and returns `true`.
`timelock_respected`	On success via the execute path (`MintRequest` has an entry for the request id at entry), the current ledger timestamp is strictly greater than the `et` that was stored in that entry.	The execute branch raises `TooEarlyToExecute` when `et >= now`, so success requires `et < now`: the current timestamp strictly exceeds the stored effective time.

`revoke_mint_request`

Categories	`core`
Authorization	`Revoker`
Parameters	`env: Env` `req: BytesN<32>` — Identifier of the mint request: the SHA-256 hash of the encoded `(receiver, amount, nonce)`.
Returns	`()`
Reads	`Revoker` — To authenticate the caller as the current revoker.
Mutates	`MintRequest` — The entry for `req` is removed (the key `MintRequest(req)` is unset).
Raises	—
Emits	`MintRequestRevoked` — On success.

Cancels a pending mint request, unconditionally removing its entry. Does not require the request to actually exist.

Postconditions

Name	Property	Proof
`mint_request_cleared`	On success, `MintRequest` has no entry for `req`.	The key `MintRequest(req)` is removed unconditionally, before the function returns.

`add_to_blocked_list`

Categories	`blocklist`
Authorization	`Operator`
Parameters	`env: Env` `user: Address` — Address to add to the block list.
Returns	`()`
Reads	`Operator` — To authenticate the caller as the current operator.
Mutates	`Blocked` — `user` is added to the set (the key `Blocked(user)` is set to `true`), and its persistent TTL is extended to `ALLOW_BLOCK_EXTEND_AMOUNT` (if it would otherwise expire sooner than `ALLOW_BLOCK_TTL_THRESHOLD`).
Raises	—
Emits	`BlockPlaced` — On success.

Adds `user` to the block list. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`user_blocked`	On success, `user` is in `Blocked`.	The key `Blocked(user)` is set to `true` unconditionally, before the function returns.

`remove_from_blocked_list`

Categories	`blocklist`
Authorization	`Operator`
Parameters	`env: Env` `user: Address` — Address to remove from the block list.
Returns	`()`
Reads	`Operator` — To authenticate the caller as the current operator.
Mutates	`Blocked` — `user` is removed from the set (the key `Blocked(user)` is unset).
Raises	—
Emits	`BlockReleased` — On success.

Removes `user` from the block list. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`user_unblocked`	On success, `user` is not in `Blocked`.	The key `Blocked(user)` is removed unconditionally, before the function returns.

`owner`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Address` — The current contract owner.
Reads	`Owner` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the address of the current contract owner.

Postconditions

Name	Property	Proof
`owner_return_value`	The result is `Owner`.	Direct read. The slot is set per `owner_always_set`, so the read cannot fail.

`pending_owner`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<Address>` — The pending owner address, or `None` if no transfer is pending.
Reads	`PendingOwner` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the address of the requested new owner, or `None` if no ownership transfer is pending.

Postconditions

Name	Property	Proof
`pending_owner_return_value`	The result is `PendingOwner` if set, otherwise `None`.	Direct read.

`et_next_owner`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<u64>` — Effective time of the pending ownership transfer (Unix timestamp, seconds), or `None` if no transfer is pending.
Reads	`EtNextOwner` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the effective time after which the pending ownership transfer can be executed, or `None` if no transfer is pending.

Postconditions

Name	Property	Proof
`et_next_owner_return_value`	The result is `Some(EtNextOwner)` if the slot is set to a non-zero value, otherwise `None`.	An absent value or the `0` sentinel maps to `None`, any other value to `Some`.

`operator`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Address` — The current operator.
Reads	`Operator` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the address of the current operator.

Postconditions

Name	Property	Proof
`operator_return_value`	The result is `Operator`.	Direct read. The slot is set per `operator_always_set`, so the read cannot fail.

`next_operator`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<Address>` — The pending operator address, or `None` if no operator change is pending.
Reads	`NextOperator` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the address of the requested new operator, or `None` if no operator change is pending.

Postconditions

Name	Property	Proof
`next_operator_return_value`	The result is `NextOperator` if set, otherwise `None`.	Direct read.

`et_next_operator`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<u64>` — Effective time of the pending operator change (Unix timestamp, seconds), or `None` if no operator change is pending.
Reads	`EtNextOperator` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the effective time after which the pending operator change can be executed, or `None` if no operator change is pending.

Postconditions

Name	Property	Proof
`et_next_operator_return_value`	The result is `Some(EtNextOperator)` if the slot is set to a non-zero value, otherwise `None`.	An absent value or the `0` sentinel maps to `None`, any other value to `Some`.

`revoker`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Address` — The current revoker.
Reads	`Revoker` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the address of the current revoker.

Postconditions

Name	Property	Proof
`revoker_return_value`	The result is `Revoker`.	Direct read. The slot is set per `revoker_always_set`, so the read cannot fail.

`next_revoker`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<Address>` — The pending revoker address, or `None` if no revoker change is pending.
Reads	`NextRevoker` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the address of the requested new revoker, or `None` if no revoker change is pending.

Postconditions

Name	Property	Proof
`next_revoker_return_value`	The result is `NextRevoker` if set, otherwise `None`.	Direct read.

`et_next_revoker`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<u64>` — Effective time of the pending revoker change (Unix timestamp, seconds), or `None` if no revoker change is pending.
Reads	`EtNextRevoker` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the effective time after which the pending revoker change can be executed, or `None` if no revoker change is pending.

Postconditions

Name	Property	Proof
`et_next_revoker_return_value`	The result is `Some(EtNextRevoker)` if the slot is set to a non-zero value, otherwise `None`.	An absent value or the `0` sentinel maps to `None`, any other value to `Some`.

`delay`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`u64` — The current delay in seconds, or `0` if no delay has been set yet.
Reads	`Delay` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the current delay (in seconds).

Postconditions

Name	Property	Proof
`delay_return_value`	The result is `Delay` if the slot is set, otherwise `0`.	Direct read of the instance slot via `unwrap_or(0)`: a set slot yields its stored value, an absent slot yields `0`. Note that unlike the `et_next_*` getters, `0` here is a genuine return value (an unset delay), not a sentinel mapped to `None`.
`delay_zero_or_in_bounds`	The result is either `0` or in `[MIN_DELAY, MAX_DELAY]`.	If `Delay` is unset the result is `0`. If set, `delay_within_bounds` gives `MIN_DELAY <= Delay <= MAX_DELAY`. Since `MIN_DELAY` > 0, a returned `0` unambiguously signals an unset delay.

`next_delay`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<u64>` — The requested new delay in seconds, or `None` if no delay change is pending.
Reads	`NextDelay` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the requested new delay (in seconds), or `None` if no delay change is pending.

Postconditions

Name	Property	Proof
`next_delay_return_value`	The result is `NextDelay` if set, otherwise `None`.	Direct read.
`next_delay_never_zero`	The result is never `Some(0)`.	A corollary of `next_delay_within_bounds`.

`et_next_delay`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<u64>` — Effective time of the pending delay change (Unix timestamp, seconds), or `None` if no delay change is pending.
Reads	`EtNextDelay` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the effective time after which the pending delay change can be executed, or `None` if no delay change is pending.

Postconditions

Name	Property	Proof
`et_next_delay_return_value`	The result is `Some(EtNextDelay)` if the slot is set to a non-zero value, otherwise `None`.	An absent value or the `0` sentinel maps to `None`, any other value to `Some`.

`gov_delay`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`u64` — The current governance delay in seconds, or `0` if no governance delay has been set yet.
Reads	`GovDelay` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the current governance delay (in seconds).

Postconditions

Name	Property	Proof
`gov_delay_return_value`	The result is `GovDelay` if the slot is set, otherwise `0`.	Direct read of the instance slot via `unwrap_or(0)`: a set slot yields its stored value, an absent slot yields `0`. Note that unlike the `et_next_*` getters, `0` here is a genuine return value (an unset governance delay), not a sentinel mapped to `None`.
`gov_delay_zero_or_in_bounds`	The result is either `0` or in `[MIN_DELAY, MAX_DELAY]`.	If `GovDelay` is unset the result is `0`. If set, `gov_delay_within_bounds` gives `MIN_DELAY <= GovDelay <= MAX_DELAY`. Since `MIN_DELAY` > 0, a returned `0` unambiguously signals an unset governance delay.

`next_gov_delay`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<u64>` — The requested new governance delay in seconds, or `None` if no governance delay change is pending.
Reads	`NextGovDelay` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the requested new governance delay (in seconds), or `None` if no governance delay change is pending.

Postconditions

Name	Property	Proof
`next_gov_delay_return_value`	The result is `NextGovDelay` if set, otherwise `None`.	Direct read.
`next_gov_delay_never_zero`	The result is never `Some(0)`.	A corollary of `next_gov_delay_within_bounds`.

`et_next_gov_delay`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<u64>` — Effective time of the pending governance delay change (Unix timestamp, seconds), or `None` if no governance delay change is pending.
Reads	`EtNextGovDelay` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the effective time after which the pending governance delay change can be executed, or `None` if no governance delay change is pending.

Postconditions

Name	Property	Proof
`et_next_gov_delay_return_value`	The result is `Some(EtNextGovDelay)` if the slot is set to a non-zero value, otherwise `None`.	An absent value or the `0` sentinel maps to `None`, any other value to `Some`.

`next_upgrade_wasm_hash`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<BytesN<32>>` — The Wasm hash of the pending upgrade, or `None` if no upgrade is pending.
Reads	`NewWasmHash` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the Wasm hash of the pending upgrade, or `None` if no upgrade is pending.

Postconditions

Name	Property	Proof
`next_upgrade_wasm_hash_return_value`	The result is `NewWasmHash` if set, otherwise `None`.	Direct read.

`et_next_upgrade`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`Option<u64>` — Effective time of the pending upgrade (Unix timestamp, seconds), or `None` if no upgrade is pending.
Reads	`EtNextUpgrade` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the effective time after which the pending upgrade can be executed, or `None` if no upgrade is pending.

Postconditions

Name	Property	Proof
`et_next_upgrade_return_value`	The result is `Some(EtNextUpgrade)` if the slot is set to a non-zero value, otherwise `None` .	An absent value or the `0` sentinel maps to `None` , any other value to `Some` .

`mint_budget`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`i128` — The current mint budget, or `0` when the slot is unset.
Reads	`MintBudget` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the current mint budget.

Postconditions

Name	Property	Proof
`mint_budget_return_value`	The result is `MintBudget` if the slot is set, otherwise `0` .	Direct read of the instance slot via `unwrap_or(0)` : a set slot yields its stored value, an absent slot yields `0` .
`mint_budget_never_negative`	The result is `>= 0` .	If `MintBudget` is unset the result is `0` . If set, `mint_budget_nonneg` gives `MintBudget >= 0` .

`mint_request_et`

Categories	`view`
Authorization	—
Parameters	`env: Env` `req: BytesN<32>` — Identifier of the mint request: the SHA-256 hash of the encoded `(receiver, amount, nonce)` .
Returns	`Option<u64>` — Effective time of the pending mint request (Unix timestamp, seconds), or `None` if no such request is pending.
Reads	`MintRequest` — To return the entry for `req` .
Mutates	—
Raises	—
Emits	—

Returns the effective time after which the pending mint request identified by `req` can be executed, or `None` if no such request is pending.

Postconditions

Name	Property	Proof
`mint_request_et_return_value`	The result is the `MintRequest` entry for `req` if set, otherwise `None`.	Direct read of the storage at key `MintRequest(req)`.

`is_blocked`

Categories	`blocklist`
Authorization	—
Parameters	`env: Env` `user: Address` — Address to query.
Returns	`bool` — Whether `user` is on the block list.
Reads	`Blocked` — To determine whether `user` is blocked (whether the key `Blocked(user)` exists).
Mutates	`Blocked` — If `user` is blocked, the persistent TTL of the key `Blocked(user)` is extended to `ALLOW_BLOCK_EXTEND_AMOUNT` (if it would otherwise expire sooner than `ALLOW_BLOCK_TTL_THRESHOLD`). Otherwise the slot is not modified.
Raises	—
Emits	—

Returns whether `user` is on the block list. If the user is blocked, this also extends the persistent TTL of the corresponding `Blocked` entry to `ALLOW_BLOCK_EXTEND_AMOUNT` (if it would otherwise expire sooner than `ALLOW_BLOCK_TTL_THRESHOLD`). Otherwise it is a pure read.

Postconditions

Name	Property	Proof
`is_blocked_return_value`	The result is `true` iff `user` is in `Blocked`.	Returns whether the key `Blocked(user)` exists. The conditional TTL extension does not change membership, so it does not affect the result.

`total_supply`

Categories	`view`
Authorization	—
Parameters	`env: Env`
Returns	`i128` — The current total token supply, or `0` when the slot is unset.
Reads	`TotalSupply` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the total token supply.

Postconditions

Name	Property	Proof
`total_supply_return_value`	The result is `TotalSupply` if the slot is set, otherwise `0`.	Direct read of the instance slot via `unwrap_or(0)`: a set slot yields its stored value, an absent slot yields `0`.
`total_supply_never_negative`	The result is `>= 0`.	If `TotalSupply` is unset the result is `0`. If set, `total_supply_nonneg` gives `TotalSupply >= 0`.

`allowance`

Categories	`token`
Authorization	—
Parameters	`env: Env` `from: Address` — The address that owns the tokens (the allowance grantor). `spender: Address` — The address allowed to spend on behalf of `from` (the allowance grantee).
Returns	`i128` — The amount `spender` may spend from `from`, or `0` if no allowance entry exists or it has expired.
Reads	`Allowance` — To read the allowance amount for the pair (`from` , `spender`) (the return value).
Mutates	—
Raises	—
Emits	—

Returns the amount that `spender` is allowed to spend on behalf of `from` . Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`allowance_active`	If `Allowance` maps the key (`from` , `spender`) to an entry `AllowanceValue { amount, expiration_ledger }` such that `expiration_ledger >= s` with `s` the current ledger sequence, then the result is `amount` .	The getter bumps the instance TTL and returns the `amount` field of `read_allowance(from, spender)` . That helper looks up the temporary `Allowance` entry for the key, and since `expiration_ledger >= s` (not expired), it returns the entry unchanged. The getter therefore returns `amount` . The instance TTL bump does not affect the result.
`allowance_expired`	If `Allowance` maps the key (`from` , `spender`) to an entry `AllowanceValue { _, expiration_ledger }` such that `expiration_ledger < s` with `s` the current ledger sequence, then the result is `0` .	The getter bumps the instance TTL and returns the `amount` field of `read_allowance(from, spender)` . That helper looks up the temporary `Allowance` entry for the key, and since `expiration_ledger < s` (expired), it returns a fresh `AllowanceValue` with `amount` set to `0` (preserving the stored `expiration_ledger`). The getter therefore returns `0` . The instance TTL bump does not affect the result.
`allowance_absent`	If `Allowance` has no entry for the key (`from` , `spender`), then the result is `0` .	The getter bumps the instance TTL and returns the `amount` field of `read_allowance(from, spender)` . That helper looks up the temporary `Allowance` entry for the key, and since there is no entry, it returns a fresh `AllowanceValue` with `amount 0` (and `expiration_ledger 0`). The getter therefore returns `0` . The instance TTL bump does not affect the result.
`allowance_never_negative`	The result is `>= 0` .	By `allowance_active` the result is a stored amount `amount` , which is `>= 0` by `allowance_nonneg` . By `allowance_expired` and `allowance_absent` it is `0` .

`approve`

Categories	`token`
Authorization	`from`
Parameters	`env: Env` `from: Address` — The address that owns the tokens and is granting the allowance; authorizes the call. `spender: Address` — The address being granted permission to spend on behalf of `from` . `amount: i128` — The allowance amount to set. Must be non-negative. `expiration_ledger: u32` — The ledger sequence through which the allowance remains valid. If `amount` is positive, must be at least the current ledger sequence.
Returns	`()`
Reads	—
Mutates	`Allowance` — The entry for the key (`from` , `spender`) is set to `AllowanceValue { amount: amount, expiration_ledger: expiration_ledger }` . If `amount` > 0, the entry TTL is extended so it lives until `expiration_ledger` (`expiration_ledger - s` ledgers, where `s` is the current ledger sequence).
Raises	`NegativeAmountNotAllowed` — If `amount` < 0. `InvalidExpirationLedger` — If `amount` > 0 and `expiration_ledger` is below the current ledger sequence.
Emits	`Approve` — On success.

Sets `spender`'s allowance over `from`'s tokens to `amount`, valid through ledger `expiration_ledger`. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`allowance_set`	On success, `Allowance` maps the key (`from`, `spender`) to `AllowanceValue { amount: amount, expiration_ledger: expiration_ledger }`.	After the `NegativeAmountNotAllowed` and `InvalidExpirationLedger` guards pass, `write_allowance` sets the entry for the key to exactly `AllowanceValue { amount: amount, expiration_ledger: expiration_ledger }` before the function returns.

`balance`

Categories	`token`
Authorization	—
Parameters	`env: Env` `id: Address` — Address whose balance to query.
Returns	`i128` — The balance of `id`, or `0` if it has no balance entry.
Reads	`Balance` — To read the balance of `id` (the return value).
Mutates	`Balance` — If `id` has a balance entry, its persistent TTL is extended to `BALANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `BALANCE_LIFETIME_THRESHOLD`). Otherwise the slot is not modified.
Raises	—
Emits	—

Returns the balance of `id`. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`balance_return_value`	The result is the balance stored for `id` in `Balance`, or `0` if no entry exists.	Direct read of `Balance(id)`: the stored value if present, otherwise `0`. The conditional TTL extension does not change the stored value, so it does not affect the result.
`balance_never_negative`	The result is `>= 0`.	If `id` has no balance entry the result is `0`. Otherwise `balance_nonneg` gives that the stored balance is `>= 0`.

`transfer`

Categories	`token`
Authorization	`from`
Parameters	`env: Env` `from: Address` — The address sending tokens; authorizes the call. `to_muxed: MuxedAddress` — The recipient, as a muxed address. Only its underlying address (`to_muxed.address()`) determines the balance update; its multiplexing id is recorded as `to_muxed_id` in the emitted `Transfer` event. `amount: i128` — The number of tokens to transfer. Must be non-negative.
Returns	`()`
Reads	`Blocked` — To check that `from` is not blocked. `Balance` — To read the current balances of `from` and `to_muxed.address()` .
Mutates	`Balance` — The balance of `from` is decreased by `amount` and the balance of `to_muxed.address()` is increased by `amount` ; both `Balance` entries are written and their persistent TTLs extended to `BALANCE_BUMP_AMOUNT` .
Raises	`UserBlocked` — If `from` is blocked. `NegativeAmountNotAllowed` — If `amount` < 0. `InsufficientBalance` — If `amount` exceeds the balance of `from` .
Emits	`Transfer` — On success.

Transfers `amount` tokens from `from` to the recipient. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`balance_transferred_distinct`	Let `from != to_muxed.address()` . Then on success, the balance of `from` decreases by `amount` and that of `to_muxed.address()` increases by `amount` (each relative to its current value, counting an absent entry as `0`). Moreover, both addresses have an entry in `Balance` afterward.	After the `UserBlocked` and `NegativeAmountNotAllowed` guards, `update_balance(Some(from), Some(to), amount)` runs `spend_balance` then `receive_balance` , where `to` is `to_muxed.address()` , distinct from `from` in this case. Each reads the relevant balance via `read_balance` , which returns the stored value if a `Balance` entry is present and `0` if absent. `spend_balance` checks `amount <= b_from` (else `InsufficientBalance` , where `b_from` is the read balance of `from`) and writes `b_from - amount` to `from` ; `receive_balance` writes `b_to + amount` to the recipient, where `b_to` is its read balance. Each `write_balance` unconditionally sets the entry (creating it if absent), so afterward `from` holds `b_from - amount` and the recipient holds `b_to + amount` , both with entries. Being distinct, the two writes target different entries and do not interfere.
`self_transfer_unchanged`	Let `from == to_muxed.address()` . Then on success, the balance of the address is unchanged. Moreover, the address has an entry in `Balance` afterward.	After the `UserBlocked` and `NegativeAmountNotAllowed` guards, `update_balance(Some(from), Some(to), amount)` runs `spend_balance` then `receive_balance` , where `to` is `to_muxed.address()` , equal to `from` in this case, so both act on the one shared entry. `spend_balance` reads the balance `b` via `read_balance` (the stored value, or `0` if absent), checks `amount <= b` (else `InsufficientBalance`), and writes `b - amount` . `receive_balance` then re-reads that just-written value and writes `(b - amount) + amount = b` , restoring the original. These writes set the entry (creating it if absent), so afterward the address has an entry holding its original balance `b` .
`total_supply_unchanged`	On success, `TotalSupply` is unchanged.	Both sides of `update_balance` are present (`Some`), so neither the mint branch (`from = None`) nor the burn branch (`to = None`) of `spend_balance` / `receive_balance` runs; only `Balance` entries are written, never `TotalSupply` .

`transfer_from`

Categories	`token`
Authorization	`spender`
Parameters	`env: Env` `spender: Address` — The address spending on behalf of `from`; authorizes the call and must hold sufficient allowance. `from: Address` — The address whose tokens are spent (the allowance grantor and balance source). `to: Address` — The recipient address. `amount: i128` — The number of tokens to transfer. Must be non-negative.
Returns	`()`
Reads	`Blocked` — To check that `spender` and `from` are not blocked. `Allowance` — To read the allowance of `spender` over `from`, to verify and decrement it. `Balance` — To read the current balances of `from` and `to`.
Mutates	`Allowance` — If `amount` > 0, the entry for the key `(from, spender)` has its `amount` decreased by `amount` (its `expiration_ledger` preserved). If `amount` == 0, `Allowance` is not modified. `Balance` — The balance of `from` is decreased by `amount` and the balance of `to` is increased by `amount`; both `Balance` entries are written and their persistent TTLs extended to `BALANCE_BUMP_AMOUNT`.
Raises	`UserBlocked` — If `spender` or `from` is blocked. `NegativeAmountNotAllowed` — If `amount` < 0. `InsufficientAllowance` — If `amount` > 0 and the live allowance of `spender` over `from` is less than `amount`. `InsufficientBalance` — If `amount` exceeds the balance of `from`.
Emits	`Transfer` — On success (with `to_muxed_id` set to `None`; muxed destinations are not supported here).

Transfers `amount` tokens from `from` to `to`, spending `spender`'s allowance over `from`. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`allowance_decremented`	On success, if `amount` > 0, the `amount` of the `Allowance` entry for `(from, spender)` decreases by `amount`, with its `expiration_ledger` preserved; if `amount` == 0, `Allowance` is unchanged.	For `amount` > 0, `spend_allowance` reads the live allowance `a` via `read_allowance` (which yields `0` if the entry is absent or expired), requires `a >= amount` (else `InsufficientAllowance`), and writes `a - amount` for the key with the original `expiration_ledger`; success therefore implies the allowance was live with `a >= amount`. For `amount` == 0, `spend_allowance` performs no write.
`balance_transferred_distinct`	Let `from != to`. Then on success, the balance of `from` decreases by `amount` and that of `to` increases by `amount` (each relative to its current value, counting an absent entry as `0`). Moreover, both addresses have an entry in `Balance` afterward.	After the `UserBlocked`, `NegativeAmountNotAllowed`, and `InsufficientAllowance` guards, `update_balance(Some(from), Some(to), amount)` runs `spend_balance` then `receive_balance`, with `from` and `to` distinct in this case. Each reads the relevant balance via `read_balance`, which returns the stored value if a `Balance` entry is present and `0` if absent. `spend_balance` checks `amount <= b_from` (else `InsufficientBalance`, where `b_from` is the read balance of `from`) and writes `b_from - amount` to `from`; `receive_balance` writes `b_to + amount` to `to`, where `b_to` is its read balance. Each `write_balance` unconditionally sets the entry (creating it if absent), so afterward `from` holds `b_from - amount` and `to` holds `b_to + amount`, both with entries. Being distinct, the two writes target different entries and do not interfere.
`self_transfer_unchanged`	Let `from == to`. Then on success, the balance of the address is unchanged. Moreover, the address has an entry in `Balance` afterward. (The allowance is still decremented per `allowance_decremented`.)	After the `UserBlocked`, `NegativeAmountNotAllowed`, and `InsufficientAllowance` guards, `update_balance(Some(from), Some(to), amount)` runs `spend_balance` then `receive_balance` on the one shared entry (since `from` and `to` coincide). `spend_balance` reads the balance `b` via `read_balance` (the stored value, or `0` if absent), checks `amount <= b` (else `InsufficientBalance`), and writes `b - amount`. `receive_balance` then re-reads that just-written value and writes `(b - amount) + amount = b`, restoring the original. These writes set the entry (creating it if absent), so afterward the address has an entry holding its original balance `b`.
`total_supply_unchanged`	On success, `TotalSupply` is unchanged.	Both sides of `update_balance` are present (`Some`), so neither the mint branch (`from = None`) nor the burn branch (`to = None`) of `spend_balance` / `receive_balance` runs; only `Balance` entries are written, never `TotalSupply`. The allowance spend does not touch `TotalSupply` either.

`burn`

Categories	`token`
Authorization	`Operator`
Parameters	`env: Env` `from: Address` — The customer on whose behalf the redemption is recorded, emitted as `customer` in the `Redeem` event. The tokens are burned from the operator's balance, not from `from`. `amount: i128` — The number of tokens to burn. Must be non-negative.
Returns	`()`
Reads	`Operator` — To authenticate the caller as the current operator, whose balance is the burn source. `Balance` — To read the operator balance, to compute the new balance. `TotalSupply` — To compute the new total supply. `MintBudget` — To compute the new mint budget.
Mutates	`Balance` — The operator balance is decreased by `amount` (entry written, TTL extended to `BALANCE_BUMP_AMOUNT`). `TotalSupply` — Decreased by `amount`. `MintBudget` — Increased by `amount`.
Raises	`NegativeAmountNotAllowed` — If `amount < 0`. `InsufficientBalance` — If `amount` exceeds the operator balance. `TotalSupplyUnderflow` — If `amount` exceeds `TotalSupply`. Defensive: given `total_supply_eq_sum_of_balances` the operator balance is at most `TotalSupply`, so passing the `InsufficientBalance` check already implies `amount <= TotalSupply`; this guard is therefore not expected to trigger.
Emits	`Burn` — On success (with `from` set to the `Operator`). `Redeem` — On success (with `customer` set to `from`).

Burns `amount` tokens from the operator's own balance, decreasing `TotalSupply`, and credits `amount` back to `MintBudget`. Records the redemption against `from` (the customer) via the emitted `Redeem`. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`operator_balance_burned`	On success, the operator balance decreases by `amount` (relative to its current value, counting an absent entry as `0`), and the operator has an entry in `Balance` afterward.	After the `NegativeAmountNotAllowed` guard, `update_balance(Some(operator), None, amount)` runs `spend_balance` on the operator: it reads the operator balance `b` via `read_balance` (stored value, or `0` if absent), requires `amount <= b` (else `InsufficientBalance`), and writes `b - amount` via `write_balance`, which sets the entry (creating it if absent). The operator is `Operator`, read for authentication.
`total_supply_decreased`	On success, `TotalSupply` decreases by `amount`.	The `to = None` side of `update_balance` takes the burn branch of `receive_balance`: it requires `amount <= TotalSupply` (else `TotalSupplyUnderflow`) and writes `TotalSupply - amount`.
`mint_budget_increased`	On success, `MintBudget` increases by `amount`.	After `update_balance`, the function reads `MintBudget` and writes it back increased by `amount` (`budget + amount`), the exact sum by `assume_no_overflow`.

`burn_from`

Categories	`token`
Authorization	—

Parameters	`env: Env`
	`_spender: Address`
	`_from: Address`
	`_amount: i128`

Returns	`()`
Reads	—
Mutates	—
Raises	`NotSupported` — Always: the function reverts unconditionally on every call.
Emits	—

Stub for the standard `TokenInterface::burn_from`. This token does not support it: every call reverts unconditionally with `NotSupported`.

`decimals`

Categories	`token`, `view`
Authorization	—
Parameters	`env: Env`
Returns	`u32` — The number of decimals.
Reads	`Metadata` — To return its `decimal` field.
Mutates	—
Raises	—
Emits	—

Returns the number of decimals used to represent token amounts.

Postconditions

Name	Property	Proof
`decimals_return_value`	The result is the `decimal` field of `Metadata`.	Direct read. The slot is set per `metadata_immutable`, so the read cannot fail.
`decimals_at_most_18`	The result is `<= 18`.	Corollary of `decimal_at_most_18`.

`name`

Categories	`token`, `view`
Authorization	—
Parameters	`env: Env`
Returns	`String` — The token name.
Reads	`Metadata` — To return its `name` field.
Mutates	—
Raises	—
Emits	—

Returns the token name.

Postconditions

Name	Property	Proof
`name_return_value`	The result is the `name` field of `Metadata`.	Direct read. The slot is set per `metadata_immutable`, so the read cannot fail.

`symbol`

Categories	`token`, `view`
Authorization	—
Parameters	`env: Env`
Returns	`String` — The token symbol.
Reads	`Metadata` — To return its `symbol` field.
Mutates	—
Raises	—
Emits	—

Returns the token symbol.

Postconditions

Name	Property	Proof
`symbol_return_value`	The result is the `symbol` field of `Metadata`.	Direct read. The slot is set per `metadata_immutable`, so the read cannot fail.