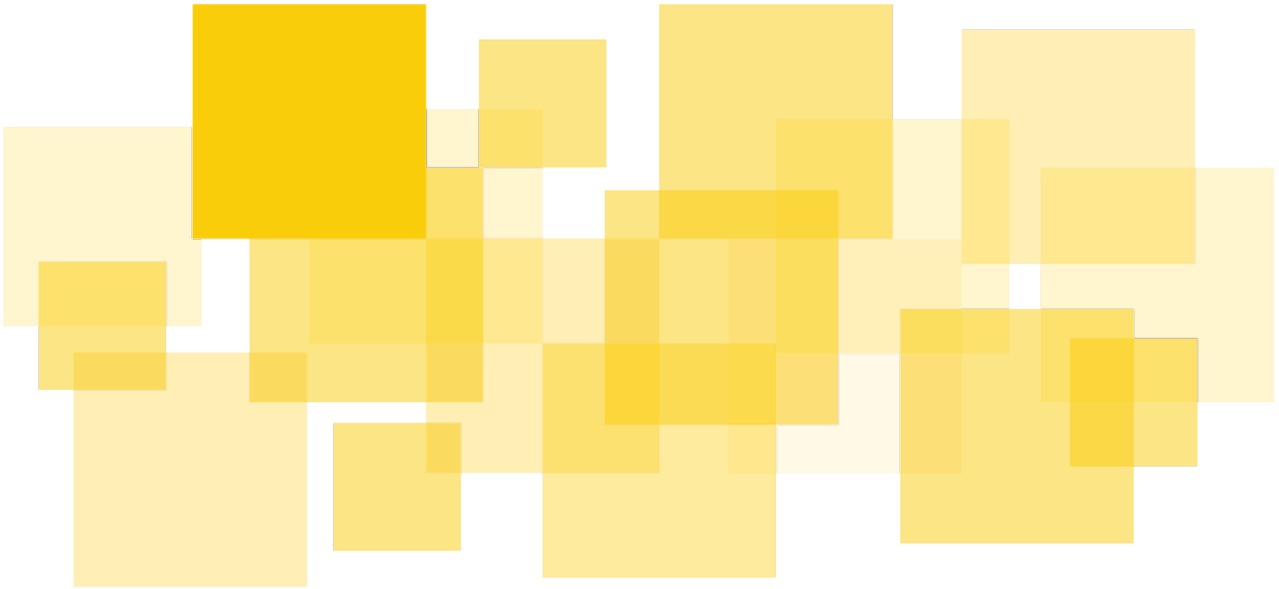
Security Audit Report

Moonlight Stellar

Delivered: July 24, 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 that 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 carry 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 that impact a large quantity of funds.

Executive Summary

Moonlight engaged Runtime Verification Inc. to conduct a 2-week review of the Moonlight core contracts, followed by 1 week of remediation support. The audit began on June 29th, 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 is the UTXO-based privacy-channel contract, which implements a shielded deposit / private-transfer / withdrawal lifecycle secured by P256 UTXO-owner signatures and Ed25519 provider co-signatures, with a secondary focus on the channel-auth (Quorum Auth) contract that gates this activity via a 1-of-N provider signature threshold and owner-governed provider registry. Each actor (Admin/Owner, Pending Admin, Privacy Provider, UTXO owner, Depositor, Withdrawal recipient, Transaction submitter) has a specific, bounded set of capabilities, and verifying these bounds, including that a provider cannot redirect, reduce, or drop a signer's intended outputs, is central to the protocol's security. The review will also cover TTL maintenance across both in-scope contracts.

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 Moonlight team to address identified issues and verify proposed fixes.

Scope

The scope of this audit was limited to the code contained in the public GitHub repository provided by the Moonlight team, located at soroban-core, with the fixed commit hash `d65780bbf0a6f8b2b601e4a536b38000775c8a15`. The elements within scope are divided into two core components, both implemented as Soroban smart contracts for the Stellar network:

- `./contracts/channel-auth` : Contains the implementation of the Quorum Auth contract, including its authority model (Admin/Owner and Pending Admin roles via stellar_access::ownable), the provider registry and its 1-of-N Privacy Provider signature threshold enforced through require_provider, the asset-channel enable/disable lifecycle, and contract upgrade logic.
- `./contracts/privacy-channel` : Contains the implementation of the Privacy Channel contract, including the UTXO-based deposit, private-transfer, and withdrawal flows; the P256 (secp256r1) UTXO-owner and Ed25519 depositor authorization and execution-binding of signed Create/ExtWithdraw conditions; conservation accounting across spend/create/withdraw operations; and persistent/instance TTL maintenance.

As findings are identified during the review, the client is expected to submit commits addressing reported issues. These remediation commits will also be reviewed to verify that the identified issues are appropriately resolved prior to the final report. The latest reviewed PRs are contained in commit ID `867c23da1769a2f56e42449ce5d5315ffb5f4cdc`.

Methodology

Runtime Verification followed a structured methodology to evaluate the correctness and security of the Moonlight core smart contracts within the agreed 2-week engagement, followed by 1-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 Moonlight protocol, focusing on the lifecycle of the channel-auth (Quorum Auth) contract and the privacy-channel contract, and on how the two components interact during deposit, private-transfer, and withdrawal operations.
The Moonlight contracts use a minimal authority model layered on explicit, signature-based authorization to correctly execute their functionality and implement their business logic. Each actor is enabled with specific capabilities in each contract, 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 actors available in each contract, including the Admin/Owner, Pending Admin, Privacy Provider, UTXO owner, Depositor, Withdrawal recipient, and Transaction submitter.

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 and each signed condition (Create, ExtWithdraw, deposit), 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 conservation guarantees, the implications of compromised or misused provider and owner keys, and dependencies on external inputs such as the provider registry and asset-channel lifecycle state.

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 channel-auth contract's privileged operations, including provider registry management, the asset-channel enable/disable lifecycle, and the two-step admin rotation mechanism;
- Correctness of the channel-auth contract's 1-of-N provider signature threshold enforced via require_provider, and its distinction from the owner-level governance quorum;
- Correctness of the privacy-channel contract's UTXO lifecycle, including deposit, private-transfer, and withdrawal flows, and the execution-binding of signed Create/ExtWithdraw conditions;
- Soroban-specific concerns across both contracts, including storage layout, persistent and instance TTL maintenance and refresh behavior, event emission, and the existing test suites;

- Conservation accounting across spend/create/withdraw operations, and the reentrancy guard protecting transact;
- 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, including archived (TTL-lapsed) UTXOs and the currently-unexecuted ExtIntegration condition variant.

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 Moonlight core contracts use a minimal authority model layered on top of explicit, signature-based authorization. Privilege is concentrated in a single `admin`, which is implemented as the contract **owner** via `stellar_access::ownable` (so "admin" and "owner" are the same role under two names), plus a registry of `provider` accounts. The owner is intended to be the Moonlight Security Council quorum (multi-sig) account, so owner authorization is itself the quorum gate. On-chain, however, the contract only ever sees a single owner `Address`, and nothing enforces that it is a multisig (a deployment choice). Note, too, that "quorum" carries two distinct meanings in this system: this **owner-level** quorum gates governance, whereas the **"Quorum Auth"** contract name refers to *provider* authorization, which is currently 1-of-N (a single required signature) rather than an M-of-N quorum. Every other participant is a transient actor that authorizes individual operations with its own key rather than holding a standing role. To validate the trust boundaries, we enumerate the actors of each contract.

`channel-auth` (Quorum Auth) contract roles

- **Admin / Owner**: the sole standing authority, set once at construction (`ownable::set_owner`) and intended to be the council quorum multi-sig. Governs the provider registry (`add_provider` / `remove_provider`), the asset-channel lifecycle (`enable_channel` / `disable_channel`), and contract upgrades (`upgrade`). All are gated by `ownable::enforce_owner_auth` . It also initiates transfer of its own role (`set_admin` -> `ownable::transfer_ownership`). Ownership transfer is **two-step** and revocable, and the contract does not expose `renounce_ownership` , so the role cannot be dropped to a null owner.
- **Pending Admin / Owner**: a transient actor, which is the address proposed by `set_admin` as the next owner. It holds no authority until it calls `accept_admin` (`ownable::accept_ownership`) with its own authorization, which finalizes the transfer. The current owner retains full control until then, and the proposal expires at a set ledger.
- **Privacy Provider**: a registered Ed25519 account (added by the admin) that must co-sign **every** governed bundle. `require_provider` enforces a hardcoded threshold of exactly one valid, unexpired provider signature, so any single registered provider can authorize a transaction. Despite the contract's *Quorum Auth* name, this is a 1-of-N check, not an M-of-N quorum. This is the off-chain, KYB-verified entity that bundles user activity. It is a gatekeeper, not a fund-holder.

`privacy-channel` (Privacy Channel) contract roles

- **Admin / Owner**: same authority/identity as above, set at construction. Upgrades the channel (`upgrade`) and transfers its own role (`set_admin` / `accept_admin`), both via `ownable` . The authorization-contract binding (`set_auth`) is written once in the constructor and is **not** a public runtime entrypoint, so the admin cannot (and need not) repoint it after deployment.

- **Pending Admin / Owner**: transient, identical two-step `accept_admin` semantics as the auth contract.
- **UTXO owner**: a transient actor identified by the 65-byte P256 (secp256r1) public key of a UTXO. Holding the matching secret key authorizes spending that UTXO via a signature over its conditions, verified through the auth contract. Its signed `Create` / `ExtWithdraw` conditions are now execution-bound: the bundle must execute each signed effect exactly, with the same destination and amount, so a provider cannot redirect, reduce, or drop a signer's intended outputs while keeping the bundle balanced.
- **Depositor**: a transient actor, which can be any address transferring the channel's asset into the privacy channel. It consents via `require_auth_for_args` over its deposit conditions (Ed25519), and those signed conditions are likewise execution-bound. Deposit and withdraw amounts must be strictly positive.
- **Withdrawal recipient**: a transient, passive actor that receives withdrawn assets. It grants no authorization and is named by the bundle.
- **Transaction submitter (bundler)**: unprivileged, meaning anyone may call `transact`, but execution depends entirely on the embedded P256 (UTXO) and provider (Ed25519) signatures and on `__check_auth`. In practice, this is a provider's operating-expense account that assembles the bundle and pays fees. It may also claim a fee as extra (unsigned) creates/withdraws, bounded by the residual value the signers left unallocated. `transact` is protected by a transient-storage reentrancy guard.

Trust/Authority Summary

Restricting attention to the standing authorities plus the key signing actors, the trust and authorization permissions per core function are summarized below:

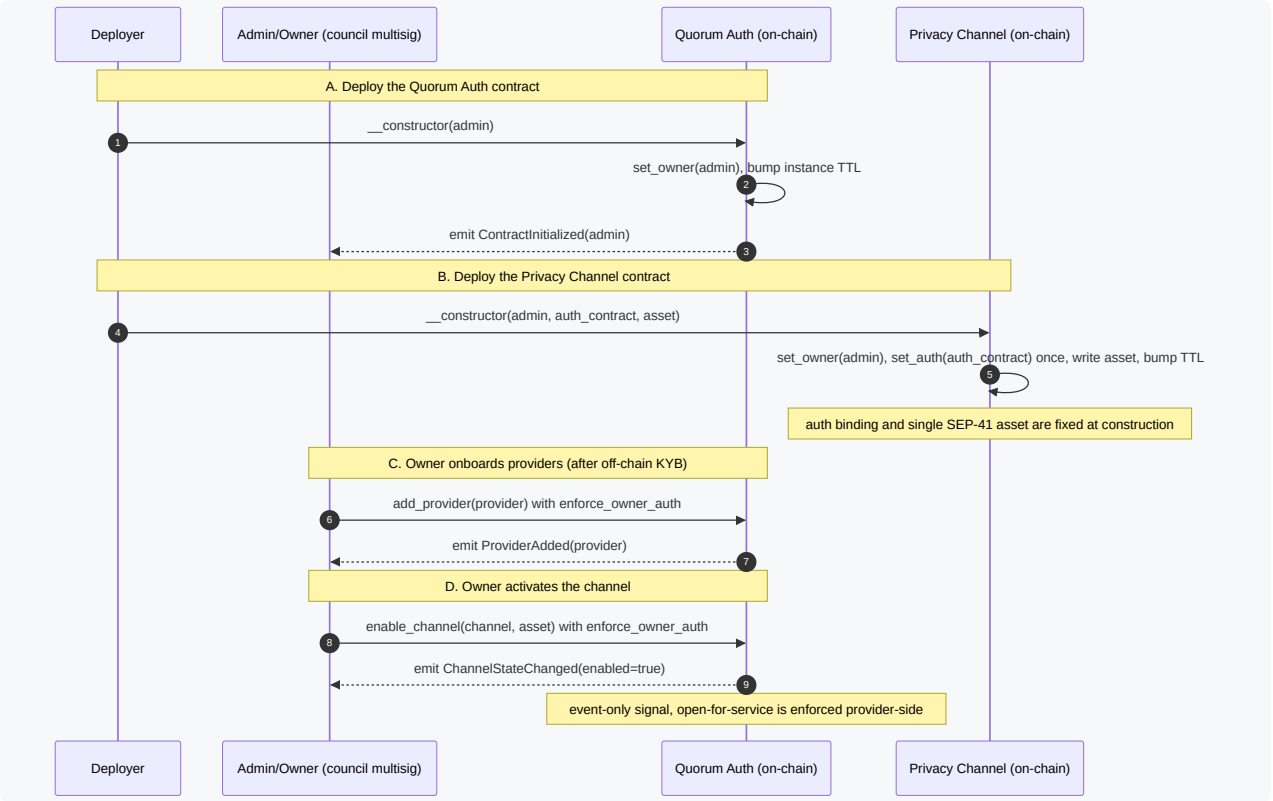
Capability	Admin / Owner	Privacy Provider	UTXO Owner	Depositor
Upgrade contract (either contract)	✓			
Propose / cancel admin transfer (`set_admin`)	✓			
Accept admin transfer (`accept_admin`)	(*)			
Add / remove provider	✓			
Enable / disable asset channel	✓			
Co-authorize any bundle (threshold = 1)		✓		
Spend a UTXO + bind its signed create/withdraw effects			✓	
Authorize a deposit + bind its signed effects				✓

(*) Exercised by the **pending** admin (the proposed next owner), not the current one. It requires the address's own authorization and finalizes the two-step transfer.

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 Quorum Auth contract with `__constructor(admin)`. This sets the council multisig as the contract **owner** (`ownable::set_owner`), bumps the instance TTL, and emits `ContractInitialized`.
B. Deploy the Privacy Channel with `__constructor(admin, auth_contract, asset)`. This sets the same owner, writes the **one-time** `auth` binding (the Quorum Auth address) and the single SEP-41 `asset` handled by that channel, and bumps the instance TTL. Neither binding is a runtime entrypoint, so they cannot drift after deployment.
C. After off-chain KYB, the owner registers each Privacy Provider via `add_provider(provider)` (owner-gated). This populates the 1-of-N provider set whose signatures gate every bundle.
D. The owner activates a channel via `enable_channel(channel, asset)` (owner-gated, **event-only**). This emits `ChannelStateChanged(enabled=true)`. The channel contract stores no such flag, so "open for service" is enforced provider-side.



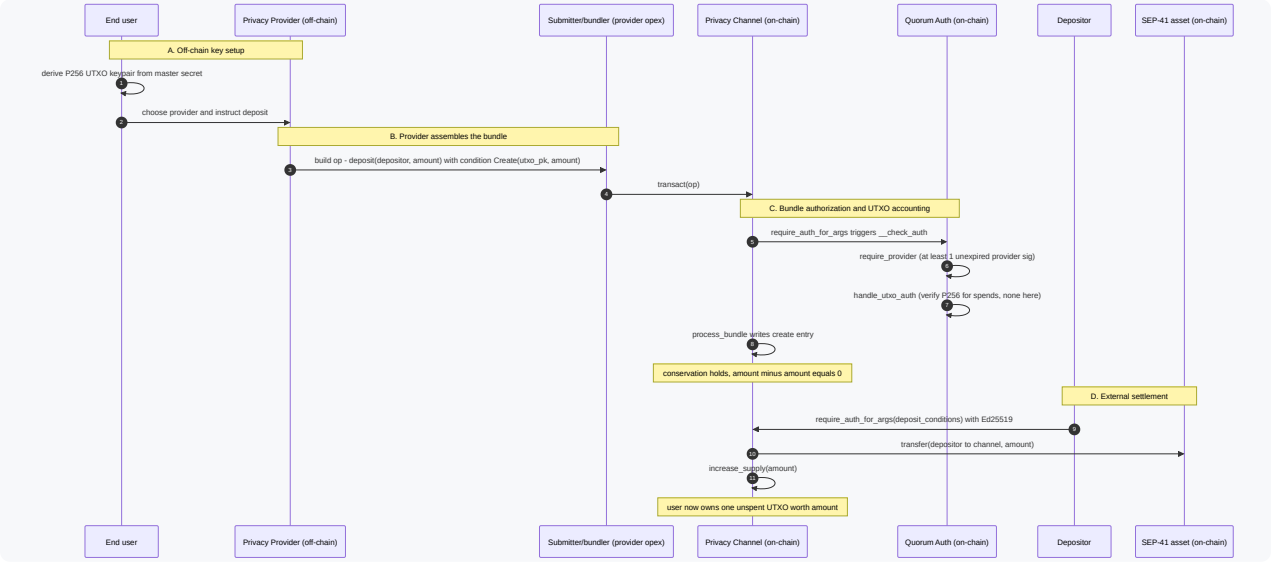
2. Deposit flow (user enters the channel)

A. Off-chain setup. The end user derives a P256 (secp256r1) UTXO keypair from their master secret and instructs a chosen provider to deposit on their behalf. Owning the UTXO means holding this P256 secret key.

B. Bundle assembly. The provider builds a `ChannelOperation` with a `deposit(depositor, amount)` carrying a `Create(utxo_pk, amount)` condition, plus a matching `create(utxo_pk, amount)`. The submitter (provider opex account) calls `transact(op)`.

C. Bundle authorization. `process_bundle` calls `require_auth_for_args` on the Quorum Auth contract, invoking its `__check_auth`: `require_provider` checks at least one unexpired provider Ed25519 signature. `handle_utxo_auth` verifies P256 signatures for any spends (none here). The create entry is written, and conservation holds (`amount − amount == 0`).

D. External settlement. The depositor authorizes via `require_auth_for_args(deposit_conditions)` (Ed25519). The asset is transferred into the channel, and `increase_supply` runs. The user now owns one unspent UTXO.



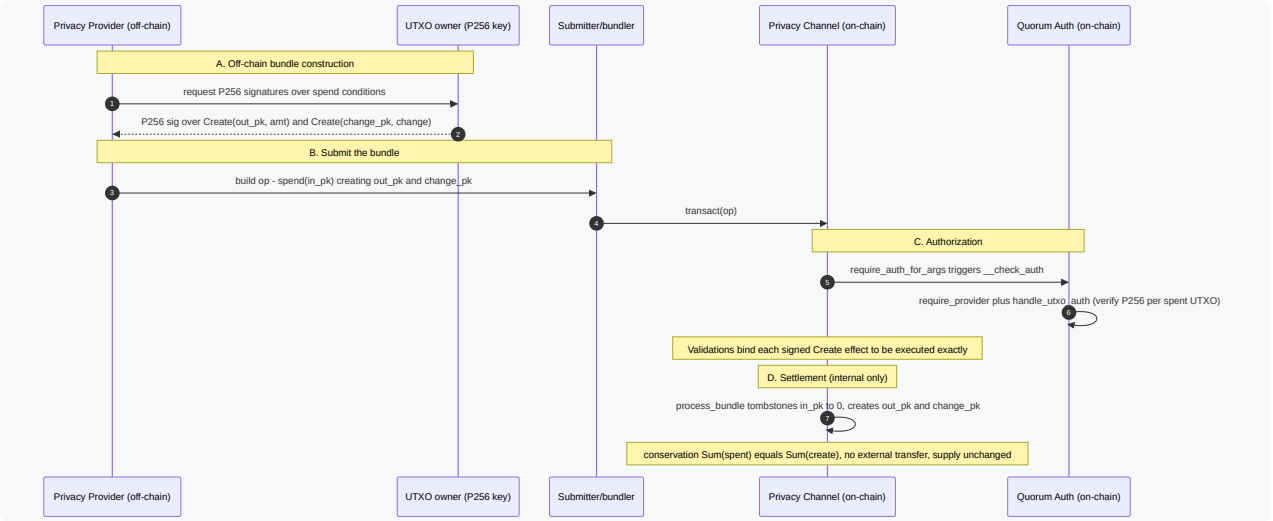
3. Private transfer flow (internal spend + create)

A. Off-chain construction. The provider builds a spend -> create bundle from the user's UTXOs. The UTXO owner P256-signs each spent UTXO's condition list, which names the intended output UTXOs (recipient + change).

B. Submission. The submitter calls `transact(op)` with the assembled spend/create sets.

C. Authorization. `__check_auth` verifies the provider signature plus a P256 signature per spent UTXO over its conditions. Validations bind each signed `Create` effect to be executed exactly, so the provider cannot redirect, reduce, or drop the owner's intended outputs.

D. Settlement (internal only). `process_bundle` tombstones each input UTXO and creates the outputs. Conservation requires `Σspent == Σcreate`. No asset leaves the contract, and total supply is unchanged. This is the unlinkable transfer.



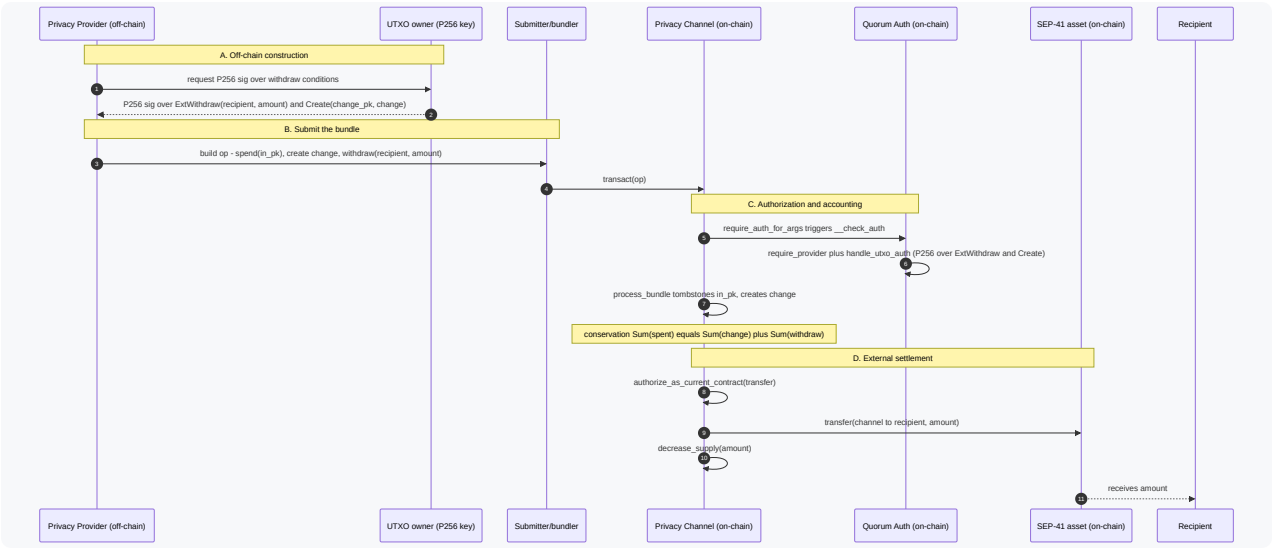
4. Withdrawal flow (user exits the channel)

A. Off-chain construction. The provider builds a bundle that spends the user's UTXO with an `ExtWithdraw(recipient, amount)` condition (plus optional `Create(change_pk, change)`). The UTXO owner P256-signs it.

B. Submission. The submitter calls `transact(op)` with the spend, any change create, and the `withdraw(recipient, amount)` entry.

C. Authorization & accounting. `__check_auth` verifies the provider signature and the P256 signatures over the `ExtWithdraw` / `Create` conditions. `process_bundle` tombstones the input, creates change, and enforces `Σspent == Σchange + Σwithdraw`.

D. External settlement. The channel authorizes its own outbound transfer (`authorize_as_current_contract`), transfers the asset to the (passive) recipient, and runs `decrease_supply`.



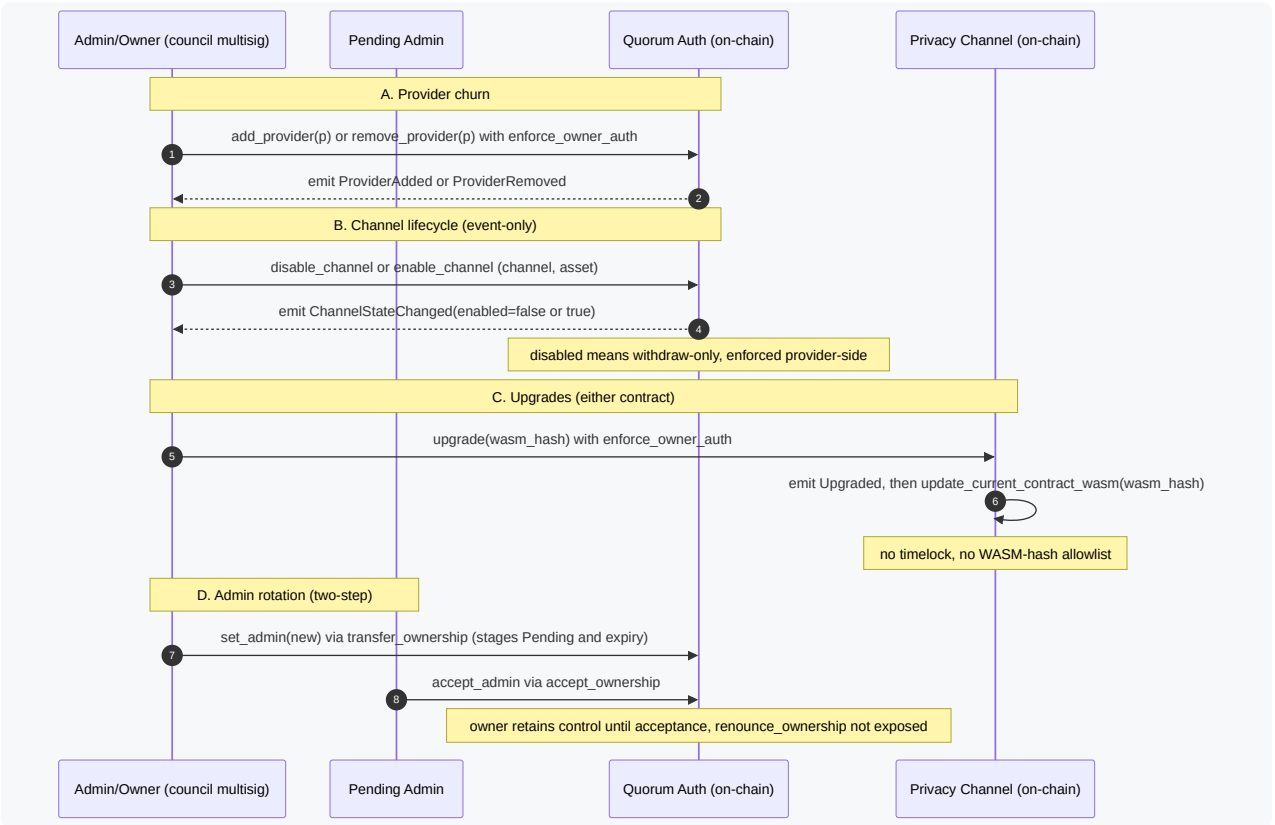
5. Governance flows (owner/council, over time)

A. Provider churn. As the KYB'd set changes, the owner calls `add_provider` / `remove_provider` (owner-gated), each emitting its event.

B. Channel lifecycle. The owner calls `disable_channel` / `enable_channel` (owner-gated, event-only). A disabled channel is treated as withdraw-only, but that restriction is enforced provider-side. The contract keeps accepting `transact`.

C. Upgrades. The owner calls `upgrade(wasm_hash)` on either contract (owner-gated). It emits `Upgraded` and then calls `update_current_contract_wasm`. There is no timelock and no WASM-hash allowlist.

D. Admin rotation (two-step). The current owner calls `set_admin(new)` (`transfer_ownership`), which stages a Pending Admin with an expiry ledger. The Pending Admin calls `accept_admin` (`accept_ownership`) to finalize. The current owner keeps full control until acceptance, and `renounce_ownership` is not exposed, so the role can never be dropped to null.



6. Ongoing keep-alive and boundary considerations: Every `create`, `spend`, and `balance` refreshes the touched UTXO's persistent TTL (~30 days), and every mutating entrypoint and every `__check_auth` refreshes the instance TTL, so active usage keeps all state alive, while a UTXO left idle past its TTL *archives* (not deleted) and needs a paid restore before it can be spent again.

Two boundaries fall outside this intended path and should be treated as such: the `ExtIntegration` condition variant exists in the types but is **never executed** by `transact`, so cross-protocol integration is not a live on-chain flow, and `enable_channel` / `disable_channel` are advisory events only. The "disabled = withdraw-only" behavior lives entirely in off-chain provider logic, not in the contract.

Invariants

The following invariants capture the properties that must hold for the Moonlight contracts (Privacy Channel and Channel Auth) 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 and serve as the reference against which the code is reviewed. Any execution path that can violate one of these properties is a finding.

A. Value conservation & supply

- **I1 — Bundle conservation.** For every `transact`: `Σspent + Σdeposited == Σcreated + Σwithdrawn`. Violation = minting or destroying channel value.
- **I2 — Supply/UTXO consistency (global, inductive).** At all times `supply == Σ amounts of all unspent UTXOs`.
- **I3 — Asset backing.** The channel's actual token balance ≥ `supply`. Holds only if the asset transfers exactly `amount` (see T1). Violation = withdrawals become insolvent.
- **I4 — Strict positivity.** Every UTXO amount > 0. Violation = negative-amount inversion of supply accounting.
- **I5 — Supply non-negativity (derived).** At all times `supply >= 0`.
- **I6 — Overflow safety.** Mathematical operations should never overflow.

B. UTXO state machine

- **I7 — One-way lifecycle.** `absent -> unspent(>0) -> spent(0)`, no backward transitions.
- **I8 — No double-spend.** A UTXO's amount is credited to a bundle at most once.
- **I9 — No intra-bundle duplicates.** A key appears at most once in the spend set and once in the create set.
- **I10 — Identifier permanence.** A spent key can never be re-created (tombstone kept forever, TTL-bumped on spend).
- **I11 — Identity injectivity.** UTXO identity = `SHA-256(pubkey65)`.

C. Authorization

- **I12 — Universal provider gate.** Every `__check_auth` requires ≥ 1 registered, unexpired provider Ed25519 signature over the tx payload.
- **I13 — Owner-bound spends.** Every spent UTXO requires a valid P256 signature under that UTXO's key, over its exact condition list + `live_until_ledger`, domain-separated by the calling contract's address. Violation = theft of user funds.

- **I14 — Effect binding (MOON-01).** Every signed `Create` / `ExtWithdraw` condition is executed exactly (same key/address, same amount): `authorized <contains> executed` by canonical XDR. Violation = provider redirects user value.
- **I15 — Depositor consent.** Every deposit executes `from.require_auth_for_args(conditions)` before pulling tokens. No one can be deposited-from involuntarily.
- **I16 — Signature expiry.** Any signature with `valid_until_ledger < current` is rejected (both P256 and provider paths). Sets a bounded replay window.
- **I17 — No effective replay.** A signed payload cannot be re-executed with effect: spends are blocked by the tombstone (I7/I8), re-creates by I7, and deposits by Soroban's own auth-nonce on the depositor's entry.
- **I18 — Non-empty conditions.** Every signer in the auth requirements carries ≥ 1 condition.
- **I19 — Single egress path.** Tokens leave the channel only through the withdraw branch of a conserved, authorized bundle.

D. Governance

- **I20 — Owner-only privileged surface.** `upgrade`, `add_provider` / `remove_provider`, `enable_channel` / `disable_channel`, `set_admin` are all gated by `ownable::enforce_owner_auth`. Violation = full protocol takeover.
- **I21 — Owner always exists.** Set once in each constructor, `renounce_ownership` not exposed, and two-step transfer can never null it, so the privileged surface can't be orphaned or bricked.
- **I22 — Two-step handover.** A pending admin has zero authority until `accept_admin` under its own key; the incumbent retains control until acceptance; proposals expire.
- **I23 — Immutable bindings.** The channel's `auth` contract and `asset` are written only in the constructor — no runtime entrypoint can repoint them. Violation (e.g., re-exposing `set_auth`) = total authorization bypass; this regressed once before (pre-MOON fix), so it deserves a regression test.

E. Execution discipline

- **I24 — Non-reentrancy.** `transact` cannot be re-entered, and neither is used in a two-part reentrancy design.
- **I25 — State-before-interaction.** All UTXO state transitions and the conservation check are complete before any external token call.
- **I26 — Liveness maintenance.** Every touch of a UTXO entry and every mutating entrypoint/auth check bumps the relevant TTL. Expiry can only ever produce archival (recoverable DoS), never silent data loss or state corruption, since nothing protocol-critical lives in temporary storage.

T. Trust assumptions (boundary conditions, not enforced)

- **T1 — Standard asset.** The SEP-41 asset transfers exactly `amount`, no fee-on-transfer/rebasing. Tokens with a custom design can be used with the consent of the privacy channel administrator.
- **T2 — Provider honesty for privacy & liveness only.** A malicious provider can censor or deanonymize (off-chain), but I12–I14 must prevent it from stealing or redirecting funds.
- **T3 — Council multisig.** The owner is a single `Address` on-chain. "Quorum" governance is a deployment assumption (T-of-N multisig), not contract-enforced.
- **T4 — Key hygiene.** P256 UTXO keys are generated fresh and never reused across channels.
- **T5 — SDK panic semantics.** `secp256r1_verify` / `ed25519_verify` trap on invalid signatures.

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] `hash_payload` encoding is not injective

Severity: Medium | Difficulty: High | Recommended Action: Fix Code | Addressed by client

Description

`hash_payload` hashes the requesting contract, the four condition buckets concatenated back to back, and the `live_until_ledger`, with no length prefixes, variant tags, or bucket separators:

Moonlight-Protocol/soroban-core/modules/primitives/src/lib.rs

Line 120 to 163 in d65780b

```rust
pub fn hash_payload(e: &Env, auth_payload: &AuthPayload, contract: &Bytes) -> Hash<32> {
    let mut b = Bytes::new(&e);
    b.append(&contract);

    let mut b_create = Bytes::new(&e);
    let mut b_deposit = Bytes::new(&e);
    let mut b_withdraw = Bytes::new(&e);
    let mut b_integrate = Bytes::new(&e);

    for cond in auth_payload.conditions.iter() {
        match cond {
            Condition::Create(utxo, amount) => {
                b_create.append(&Bytes::from_slice(&e, utxo.to_array().as_ref()));
                b_create.append(&Bytes::from_slice(&e, &amount.to_le_bytes()));
            }
            Condition::ExtDeposit(addr, amount) => {
                b_deposit.append(&addr.to_string().to_bytes());
                b_deposit.append(&Bytes::from_slice(&e, &amount.to_le_bytes()));
            }
            Condition::ExtWithdraw(addr, amount) => {
                b_withdraw.append(&addr.to_string().to_bytes());
                b_withdraw.append(&Bytes::from_slice(&e, &amount.to_le_bytes()));
            }
            Condition::ExtIntegration(adapter, utxos, amount) => {
                b_integrate.append(&adapter.to_string().to_bytes());
                for utxo in utxos.iter() {
                    b_integrate.append(&Bytes::from_slice(&e, utxo.to_array().as_ref()));
                }
                b_integrate.append(&Bytes::from_slice(&e, &amount.to_le_bytes()));
            }
        }
    }
    b.append(&b_create);
    b.append(&b_deposit);
    b.append(&b_withdraw);
    b.append(&b_integrate);

    b.append(&Bytes::from_slice(
        &e,
        &auth_payload.live_until_ledger.to_le_bytes(),
    ));

```

```
162     e.crypto().sha256(&b)
163 }
```

The number of conditions in a bucket is unmarked, the `ExtIntegration` UTXO vector has no length marker, and empty buckets contribute no bytes, so distinct condition lists can encode to the same byte stream. Record sizes are fixed (`i128` amount 16 bytes, UTXO id 65 bytes, address strkey 56 bytes), so a collision must preserve total length — easily arranged, since UTXO ids and amounts are free bytes. Three examples:

- **Redistribute UTXOs within the integration bucket** (same count, same total UTXOs): `Int(A, [u1, u2], m1), Int(B, [], m2)` and `Int(A, [u1], m1'), Int(B, [u2], m2')` are both 274 bytes and can be made byte-identical by choosing the free UTXO/amount bytes so the boundaries line up.
- **Cross-bucket through `Create`** : a `Create` record is 81 bytes (`65 + 16`) and an `ExtDeposit` record is 72 bytes (`56 + 16`); `81·a = 72·b` first solves at `a = 8, b = 9` , so an all- `Create` bucket of 8 conditions (648 bytes) can read as an all- `ExtDeposit` bucket of 9, provided the 56-byte address slots decode as valid strkeys.
- **Same-size cross-bucket, no crafting**: `ExtDeposit` and `ExtWithdraw` records encode identically — both `56 + 16` bytes, address strkey then little-endian `i128` , with no tag to separate them. Since the buckets are adjacent and empty ones contribute nothing, `[Dep(X, a)]` and `[Wd(X, a)]` both encode to `enc(X, a)` and are byte-identical, and the boundary slides freely (`Dep(X, a), Dep(Y, b)` vs `Dep(X, a), Wd(Y, b)`). `ExtIntegration(A, [], m)` is likewise 72 bytes and collides with `Dep(A, m)` / `Wd(A, m)` . Unlike the cases above, this needs no free-byte arrangement and no strkey carving — the address slots hold the signer's own valid strkeys — so the colliding pair exists with probability 1.

The variable-length variant:

Moonlight-Protocol/soroban-core/modules/primitives/src/lib.rs

Line 11 in d65780b

```
11     ExtIntegration(Address, Vec<BytesN<65>>, i128), // contract id of the adapter, the keys to authorize the withdrawal, the amount to deposit
```

A signed hash therefore does not identify one condition list. A signer who reviews decoded conditions and signs may have authorized a second list with the same encoding, which `handle_utxo_auth` will accept:

Moonlight-Protocol/soroban-core/modules/auth/src/core.rs

Line 126 in d65780b

```
126                         verify_signature(&e, &signer, &sig_variant, &msg)?;
```

Severity splits in two. The crafted cases (`Create` -bucket reparse, integration redistribution) can't reuse an existing honest signature (that would be a SHA-256 second preimage), so they need the payload composed before signing. The deposit↔withdraw / empty-integration case has no such gate: a genuine signature over `[Dep(X, a)]` is byte-identical to one over `[Wd(X, a)]`, so an untrusted *submitter* can reinterpret an already-signed deposit as a withdraw at tx-build time. Either way it is an audit hazard: two bundles indistinguishable to a signer break any "this signature authorized exactly this bundle" reasoning.

Recommendation

Make the signed encoding self-delimiting. Minimally, length-prefix every variable-length field (the `ExtIntegration` UTXO vector especially, plus a count of conditions per bucket) and mark bucket boundaries. Better, hash the canonical XDR of the `Vec<Condition>` via `ToXdr`, injective by construction and already a dependency, since `equal_condition_sequence` already compares conditions on XDR bytes:

Moonlight-Protocol/soroban-core/modules/primitives/src/lib.rs

Line 200 to 214 in d65780b

```rust
pub fn equal_condition_sequence(e: &Env, a: &Vec<Condition>, b: &Vec<Condition>) -> bool {
    if a.len() != b.len() {
        return false;
    }
    let mut it_b = b.iter();
    for ca in a.iter() {
        match it_b.next() {
            Some(cb) => {
                // Compare on-wire representation to avoid needing Eq on Condition.
                if ca.to_xdr(&e) != cb.to_xdr(&e) {
                    return false;
                }
            }
            None => return false,
        }
```

The fix is small, and afterward a P256 signature covers exactly one condition list. Note that a whole- `Vec` XDR digest is order-sensitive, so it also binds the ordering the signer reviewed; if order-insensitivity is wanted, canonicalize (e.g. sort) the conditions before hashing.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/38 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/97922db) by hashing the canonical XDR of the whole `Vec<Condition>`.

Since the only production caller passes `contract` as a fixed-length 56-byte strkey, the payload is injective in practice.

The `hash_payload` function itself is still not injective, though: the payload is

```
contract ‖ xdr(conditions) ‖ le32(live_until)
```

and the contract field is appended as unframed Bytes, so injectivity of the full payload rests on the caller's convention rather than the encoding. To avoid a future regression, consider length-prefixing the contract field too, or XDR-encoding the whole `(contract, conditions, live_until)` tuple as one structure.

[A2] UTXO keys are never validated, value under a malformed key is locked permanently

Severity: Medium | Difficulty: High | Recommended Action: Fix Code | Addressed by client

Description

No UTXO write path validates that a created key is a well-formed P-256 point. `Store::create` accepts any 65-byte value:

Moonlight-Protocol/soroban-core/modules/storage/src/lib.rs

Line 80 to 93 in d65780b

```
80      pub fn create(&mut self, utxo65: &BytesN<65>, amount: i128) {
81          if amount <= 0 {
82              panic_with_error!(&self.env, Error::InvalidCreateAmount);
83          }
84
85          let k = self.utxo_key(utxo65);
86
87          if self.env.storage().persistent().get::<_, i128>(&k).is_some() {
88              panic_with_error!(&self.env, Error::UtxoAlreadyExists);
89          }
90
91          self.env.storage().persistent().set(&k, &amount);
92          self.bump_ttl(&k);
93      }
```

and `process_bundle` records it as-is, guarding only amount and prior existence:

Moonlight-Protocol/soroban-core/modules/utxo-core/src/core.rs

Line 116 to 124 in d65780b

```
116             for (create_utxo, amount) in bundle.create.iter() {
117                 if store.balance(&create_utxo) != -1 {
118                     panic_with_error!(e, MoonlightError::UtxoAlreadyExists);
119                 }
120
121                 assert_with_error!(&e, amount > 0, MoonlightError::InvalidCreateAmount);
122
123                 store.create(&create_utxo, amount);
124                 total_available_balance -= amount;
```

Under the deployed wiring the recorded bytes are also the sole spending credential: `__check_auth` releases a UTXO only on a P-256 signature verifying under its key, and `secp256r1_verify` traps on a malformed key, so any spend attempt on such a UTXO aborts the transaction. Value created under a malformed key, for example a mistyped key in a signed `Create` condition, is therefore locked permanently once the bundle settles.

The precondition is confined: the key is chosen by the party who signs the `Create` (or fixed as composer-controlled residual), so this is honest-mistake loss, not an attack path. No trust boundary is crossed. But the check is cheap, near-complete against the stated failure mode, and the failure is unrecoverable, which is why it is worth stating.

The check cannot, of course, prove spendability: a valid on-curve key whose private key nobody holds is equally locked, and no on-chain check can detect that. That residual risk is inherently a wallet-side concern, which is why the recommendation below carries a documentation component either way.

Recommendation

Reject a create whose key is not a valid P-256 point at the `Store::create` guard (a one-time cost paid only at creation, never on spend), with the `0x04` prefix check as a minimum if full on-curve validation is deemed too costly on-chain. In the latter case, document the burden explicitly so wallets validate keys before funding them; wallets should validate regardless, since key-possession errors are undetectable on-chain.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/44 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/49b0fdb) by checking the `0x04` marker in `Store::create`, and documenting the additional validation burden.

Note that the comment identifier PC-16 collides with PC-16 from [A3]. When merging the PRs, this needs to be resolved.

[A3] Byte-identical signed effects merge into one execution

Severity: Medium | Difficulty: High | Recommended Action: Fix Code | Addressed by client

Description

The signed-effects binding compares conditions as a set keyed by canonical XDR bytes, so byte-identical conditions collapse to one entry, discharged by a single executed occurrence:

Moonlight-Protocol/soroban-core/contracts/privacy-channel/src/transact.rs

Line 113 to 130 in d65780b

```rust
fn assert_signed_effects_are_executed(e: &Env, op: &ChannelOperation) {
    let mut authorized: Map<Bytes, ()> = Map::new(e);
    collect_authorized_effects(e, &mut authorized, &op.spend);
    collect_authorized_effects_from_external(e, &mut authorized, &op.deposit);

    let mut executed: Map<Bytes, ()> = Map::new(e);
    for (utxo, amount) in op.create.iter() {
        executed.set(Condition::Create(utxo, amount).to_xdr(e), ());
    }
    for (addr, amount, _conds) in op.withdraw.iter() {
        executed.set(Condition::ExtWithdraw(addr, amount).to_xdr(e), ());
    }

    // Subset: every signed effect must be executed exactly. Extra executed effects are allowed.
    for key in authorized.keys().iter() {
        assert_with_error!(e, executed.contains_key(key), Error::UnauthorizedOperation);
    }
}
```

The balance still counts every spent and deposited amount in full. When two independently signed spends each demand the identical `ExtWithdraw(to, amount)`, a composer holding both can deliberately batch them: the bundle pays `to` once, and the difference becomes unsigned residual the composer allocates to any recipient. Nor can such a bundle settle the effect once per signer: duplicate withdraw addresses are rejected, and a combined `(to, 2·amount)` tuple has different bytes and fails the subset check — merged execution is the only way the pair settles in one bundle.

The precondition is two independently signed identical conditions, each backed by its signer's value; a composer cannot manufacture the second from the first, since duplicating a condition inside a signed list breaks the signature, re-spending the same UTXO is rejected, and a condition on the composer's own

spend is backed by the composer's own funds. Byte identity requires equal address and equal amount (for `Create`, the same key and amount). That arises when several parties withdraw a common amount to a shared address, and — more plainly — when one party authorizes the same effect repeatedly: recurring payments of a fixed amount to a fixed recipient produce byte-identical conditions on distinct spends, and a composer batching two such spends pays the recipient once while the other payment becomes residual.

Recommendation

Distinguishing merged effects would require matching executed effects to signed conditions with multiplicity rather than as a set — a substantial change to the binding and its balance reasoning.

Given the precondition, treat this as a batching constraint and document it: byte-identical execution-bound conditions must never share a bundle, screening co-batched conditions is the submitter's responsibility, and repeated identical effects (e.g. recurring payments) settle safely only in separate bundles.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/45 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/90bf08c) by documenting the batching constraint.

Note that the comment identifier PC-16 collides with PC-16 from [A2]. When merging the PRs, this needs to be resolved.

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] `set_admin` grants the maximum acceptance window, a pending transfer never auto-expires

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

Description

Both contracts wrap `ownable::transfer_ownership` in `set_admin`, hardcoding `live_until_ledger` to `e.ledger().max_live_until_ledger()` without exposing it:

Moonlight-Protocol/soroban-core/contracts/channel-auth/src/contract.rs

Line 86 to 88 in f9cdd3b

```
86      pub fn set_admin(e: &Env, new_admin: Address) {
87          ownable::transfer_ownership(e, &new_admin, e.ledger().max_live_until_ledger());
88      }
```

Moonlight-Protocol/soroban-core/contracts/privacy-channel/src/contract.rs

Line 70 to 72 in f9cdd3b

```
70      pub fn set_admin(e: &Env, new_admin: Address) {
71          ownable::transfer_ownership(e, &new_admin, e.ledger().max_live_until_ledger());
72      }
```

Per its documentation, `transfer_ownership` treats `live_until_ledger` as the ledger "until which the new owner can accept." By always passing the maximum ledger, `set_admin` gives every pending transfer the longest possible window: a `PendingOwner` set today stays acceptable for the full maximum temporary TTL rather than lapsing on its own.

A pending transfer only matters while it lingers unaccepted, since a healthy one closes by acceptance. While it lingers, whoever holds `PendingOwner` can `accept_admin` at any time until expiry, so if that key is later compromised, the exposure runs for the full maximum TTL (months). A finite deadline caps this automatically, without anyone monitoring. The owner can always overwrite `PendingOwner` via `set_admin(SAFE)`, but that requires noticing and reacting, whereas expiry does not.

Additionally, because the deadline is fixed at the maximum, the documented `0`-cancels behavior is also unreachable — there is no way to clear a pending transfer other than redirecting it (e.g. `set_admin(self)` then `accept_admin`).

Recommendation

Thread the deadline through as `set_admin(e, new_admin, live_until_ledger)` and forward it to `transfer_ownership`. Callers can then bound the acceptance window to a sensible finite value (and, as a bonus, pass `0` to cancel), without changing behavior for callers who continue to pass the maximum.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/40 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/e769ee7) by threading `live_until_ledger` through `set_admin` in both contracts, and additionally enforcing an in-contract 7-day ceiling (`AcceptanceWindowTooLong`) with `live_until_ledger == 0` exempted to preserve the library's cancel path.

[B2] Bundle balance accumulator uses raw arithmetics rather than checked_add/checked_sub

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

Description

`process_bundle` maintains its running conservation total, `total_available_balance`, with **unchecked** `+=` / `-=` operators, rather than the `checked_add` / `checked_sub` pattern used elsewhere in the same flow. The invariant that guards value conservation (`total_available_balance == expected_outgoing`) is therefore only sound due to its reliance on the overflow checks on the environment.

The accumulator initializes at the incoming amount, then adds each spent UTXO's amount and subtracts each created amount with raw operators, before the equality check:

Moonlight-Protocol/soroban-core/modules/utxo-core/src/core.rs

Line 95 to 141 in d65780b

```
95          Store::apply(e, |store| {
96              for spend_utxo in bundle.spend.iter() {
97                  let amount = match store.balance(&spend_utxo) {
98                      a if a > 0 => a,
99                      0 => panic_with_error!(e, MoonlightError::UtxoAlreadySpent),
100                     _ => panic_with_error!(e, MoonlightError::UtxoDoesNotExist),
101                 };
102
103                 store.spend(&spend_utxo);
104                 total_available_balance += amount;
105
106                 #[cfg(not(feature = "no-utxo-events"))]
107                 UtxoEvent {
108                     name: symbol_short!("utxo"),
109                     utxo: spend_utxo.clone(),
110                     action: symbol_short!("spend"),
111                     amount,
112                 }
113                 .publish(&e);
114             }
115
116             for (create_utxo, amount) in bundle.create.iter() {
117                 if store.balance(&create_utxo) != -1 {
118                     panic_with_error!(e, MoonlightError::UtxoAlreadyExists);
119                 }
120
121                 assert_with_error!(&e, amount > 0, MoonlightError::InvalidCreateAmount);
122
123                 store.create(&create_utxo, amount);
124                 total_available_balance -= amount;
125
126                 #[cfg(not(feature = "no-utxo-events"))]
127                 UtxoEvent {
128                     name: symbol_short!("utxo"),
129                     utxo: create_utxo.clone(),
130                     action: symbol_short!("create"),
131                     amount,
132                 }
133                 .publish(&e);
134             }
135         });
136
```

```
137         assert_with_error!(
138             &e,
139             total_available_balance == expected_outgoing,
140             MoonlightError::UnbalancedBundle
141         );
```

Contrast this with `pre_process_channel_operation` in the privacy channel, which already sums external amounts defensively with `checked_add` and a dedicated `AmountOverflow` error:

Moonlight-Protocol/soroban-core/contracts/privacy-channel/src/transact.rs

Line 41 to 58 in d65780b

```
41      let mut total_deposit: i128 = 0;
42      for (_addr, amt, _conds) in op.deposit.iter() {
43          // MOON-05: reject non-positive amounts in-contract rather than relying on the asset SAC.
44          assert_with_error!(&e, amt > 0, Error::InvalidExternalAmount);
45          total_deposit = match total_deposit.checked_add(amt) {
46              Some(v) => v,
47              None => panic_with_error!(&e, Error::AmountOverflow),
48          };
49      }
50
51      let mut total_withdraw: i128 = 0;
52      for (_addr, amt, _conds) in op.withdraw.iter() {
53          assert_with_error!(&e, amt > 0, Error::InvalidExternalAmount);
54          total_withdraw = match total_withdraw.checked_add(amt) {
55              Some(v) => v,
56              None => panic_with_error!(&e, Error::AmountOverflow),
57          };
58      }
```

With the environment specification of overflow checks, an overflow/underflow panics and reverts the transaction, so conservation holds, and this is not exploitable as shipped. The concern is one of robustness and locality of reasoning, and it is sharper because `utxo-core` is a reusable library. `process_bundle` is exposed for arbitrary integrating contracts, and a consumer that compiles it under a profile under different circumstances would silently reintroduce wrap-around arithmetic on the protocol's central invariant. A core safety property of a shared module should not depend on the build configuration of the linker. Reaching the overflow boundary also requires amounts near `i128::MAX`, which is not approachable with a legitimately funded channel, unless we consider tokens with custom, abnormally large decimals.

Recommendation

Replace the raw `+=` / `-=` on `total_available_balance` with `checked_add` / `checked_sub`, mapping failure to the existing `AmountOverflow` / `AmountUnderflow` errors, mirroring the deposit/withdraw

summation in `pre_process_channel_operation`. This makes conservation correct-by-construction regardless of the compiling profile and removes the implicit dependency on `overflow-checks` for a security-critical invariant. It also standardizes the approaches to mathematical operations in the contract.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/39 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/12a5280) by replacing the accumulator's raw `+=` / `-=` with checked_add/checked_sub, mapping failure to the existing `AmountOverflow` / `AmountUnderflow` errors as recommended.

[B3] `hash_payload` Doc Specifies the Wrong Integer Width (8 vs 16 bytes)

Severity: Informative | Recommended Action: Document Prominently | Addressed by client

Description

The `hash_payload` docstring states that "all integer amounts are encoded as little-endian 8-byte sequences," but amounts are `i128`, and the implementation appends `amount.to_le_bytes()`, which is **16 bytes**.

Docstring:

Moonlight-Protocol/soroban-core/modules/primitives/src/lib.rs

Line 115 in d65780b

```
115  /// For consistency, all integer amounts are encoded as little-endian 8-byte sequences.
```

Encoding that contradicts it using i128::to_le_bytes:

Moonlight-Protocol/soroban-core/modules/primitives/src/lib.rs

Line 129 to 151 in d65780b

```
129     for cond in auth_payload.conditions.iter() {
130         match cond {
131             Condition::Create(utxo, amount) => {
132                 b_create.append(&Bytes::from_slice(&e, utxo.to_array().as_ref()));
133                 b_create.append(&Bytes::from_slice(&e, &amount.to_le_bytes()));
134             }
135             Condition::ExtDeposit(addr, amount) => {
136                 b_deposit.append(&addr.to_string().to_bytes());
137                 b_deposit.append(&Bytes::from_slice(&e, &amount.to_le_bytes()));
138             }
139             Condition::ExtWithdraw(addr, amount) => {
140                 b_withdraw.append(&addr.to_string().to_bytes());
141                 b_withdraw.append(&Bytes::from_slice(&e, &amount.to_le_bytes()));
142             }
143             Condition::ExtIntegration(adapter, utxos, amount) => {
144                 b_integrate.append(&adapter.to_string().to_bytes());
145                 for utxo in utxos.iter() {
146                     b_integrate.append(&Bytes::from_slice(&e, utxo.to_array().as_ref()));
147                 }
148                 b_integrate.append(&Bytes::from_slice(&e, &amount.to_le_bytes()));
149             }
150         }
151     }
```

This is a documentation-only defect with no on-chain security impact, as the contract hashes 16-byte amounts and verifies against 16-byte amounts, so it is internally consistent. The failure mode is a fail-safe. An off-chain signer built to the documented 8-byte format computes a non-matching hash, and its signatures are simply rejected by the verifier (no unauthorized action, no stuck state). The real cost is developer/integration confusion for anyone treating the prose as the canonical serialization spec.

Two sibling errors live in the same docstring and should be corrected together:

1. The contract address is described as "(32 bytes)" when it is the ~56-byte strkey string, and;
2. The note suggesting conditions be kept "in the same ordering as the original bundle" is misleading, since the implementation buckets conditions by type.

Recommendation

Correct the amount width (8 to 16 bytes), and fix the sibling contract-address and ordering statements in the same pass. Preferably, replace hand-counted byte widths with a precise field-by-field serialization

description and commit signed test vectors as the authoritative reference, so off-chain signers verify against fixtures rather than prose.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/38 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/97922db), where all three docstring errors were corrected. The rewritten docstring accurately describes the new XDR-based encoding.

[B4] Redundant Reentrancy Guard

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

Description

`transact` wraps its body in a temporary-storage reentrancy guard, where `enter_reentrancy_guard` sets a flag on entry and panics with `ReentrantCall` if it is already set, and `exit_reentrancy_guard` clears it on exit.

Moonlight-Protocol/soroban-core/contracts/privacy-channel/src/contract.rs

Line 39 to 54 in d65780b

```
39  // MOON-06: reentrancy guard. `transact` makes external SAC calls (deposit pull / withdraw push);
40  // a non-standard or malicious asset could call back into `transact`. The guard is a transient flag
41  // that is set on entry and cleared on exit; a re-entrant call observes it and reverts. (On revert
42  // the transient write is rolled back, so the flag never persists across transactions.)
43  const REENTRANCY_GUARD: Symbol = symbol_short!("RGUARD");
44
45  fn enter_reentrancy_guard(e: &Env) {
46      if e.storage().temporary().has(&REENTRANCY_GUARD) {
47          panic_with_error!(e, Error::ReentrantCall);
48      }
49      e.storage().temporary().set(&REENTRANCY_GUARD, &true);
50  }
51
52  fn exit_reentrancy_guard(e: &Env) {
53      e.storage().temporary().remove(&REENTRANCY_GUARD);
54  }
```

Moonlight-Protocol/soroban-core/contracts/privacy-channel/src/contract.rs

Line 107 to 119 in d65780b

```rust
    pub fn transact(e: Env, op: ChannelOperation) {
        bump_instance_ttl(&e);
        enter_reentrancy_guard(&e);

        let (utxo_op, total_deposit, total_withdraw) =
            pre_process_channel_operation(&e, op.clone());

        Self::process_bundle(&e, utxo_op.clone(), total_deposit, total_withdraw);

        execute_external_operations(&e, op.deposit, op.withdraw);

        exit_reentrancy_guard(&e);
    }
```

Soroban prohibits reentrancy at the host level, meaning that a contract currently in the call stack cannot be re-entered. The exact scenario the guard targets (a non-standard or malicious asset calling back into `transact` during the external `deposit` / `withdraw` transfers) is therefore already rejected by the runtime. The guard is redundant with that platform guarantee, as the host rule already covers every entrypoint (not just `transact`). Additionally, the flag takes no effect for cross-contract cases (a call into a different channel is not reentrancy of this one and would not observe this contract's guard). This is not a vulnerability, and the cost is minor: two temporary-storage writes per `transact`, plus the extra code path and `ReentrantCall` error.

Recommendation

Optionally, remove the guard and the `ReentrantCall` error, replacing it with a comment noting that reentrancy is prevented by Soroban's host-level non-reentrancy. If the team prefers to keep it as explicit defense-in-depth, document that the primary protection is the platform guarantee and the guard is secondary, and do not treat it as a load-bearing control. Either way, since the redundancy depends on the host guarantee, re-validate that non-reentrancy semantics hold for the target protocol version on any SDK/protocol upgrade.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/41 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/7d24257) by removing the redundant reentrancy guard and its `ReentrantCall` error, as recommended.

[B5] Fragmented and Partially Duplicated Validation Across the Transact Flow

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

Description

Validation of a `transact` bundle is distributed across four crates with no single structural-validation layer:

- **`primitives`** - reusable predicates: condition checks (`conflicts_with` , `equal_condition_sequence`) and operation-identifier deduplication (`no_duplicate_keys` over UTXO keys, `no_duplicate_addresses` over addresses).
- **`privacy-channel`** **(** **`transact.rs`** **)** - bundle orchestration and higher-level checks: `op_has_no_conflicting_conditions` , `verify_external_operations` , the amount-positivity loops, and `assert_signed_effects_are_executed` .
- **`utxo-core`** **(** **`process_bundle`** **)** - per-UTXO deduplication and state pre-checks.
- **`storage`** **(** **`Store`** **)** - per-UTXO existence, spent-state, and positivity enforcement.

Beyond being spread out, per-UTXO validation is duplicated between the last two layers, with redundant storage reads. For `spend` , `process_bundle` calls `store.balance()` to verify the UTXO is unspent and obtain its amount, then `store.spend()` re-reads and re-checks the same state, and its returned amount (the same value) is discarded. For `create` , `process_bundle` checks `balance() != -1` and `amount > 0` , then `store.create()` re-checks existence and positivity.

Moonlight-Protocol/soroban-core/modules/utxo-core/src/core.rs

Line 95 to 141 in d65780b

```
95          Store::apply(e, |store| {
96              for spend_utxo in bundle.spend.iter() {
97                  let amount = match store.balance(&spend_utxo) {
98                      a if a > 0 => a,
99                      0 => panic_with_error!(e, MoonlightError::UtxoAlreadySpent),
100                     _ => panic_with_error!(e, MoonlightError::UtxoDoesNotExist),
101                 };
102
103                 store.spend(&spend_utxo);
104                 total_available_balance += amount;
105
106                 #[cfg(not(feature = "no-utxo-events"))]
107                 UtxoEvent {
108                     name: symbol_short!("utxo"),
109                     utxo: spend_utxo.clone(),
110                     action: symbol_short!("spend"),
111                     amount,
112                 }
113                 .publish(&e);
114             }
115
116             for (create_utxo, amount) in bundle.create.iter() {
117                 if store.balance(&create_utxo) != -1 {
118                     panic_with_error!(e, MoonlightError::UtxoAlreadyExists);
119                 }
120
121                 assert_with_error!(&e, amount > 0, MoonlightError::InvalidCreateAmount);
122
123                 store.create(&create_utxo, amount);
124                 total_available_balance -= amount;
125
126                 #[cfg(not(feature = "no-utxo-events"))]
127                 UtxoEvent {
128                     name: symbol_short!("utxo"),
129                     utxo: create_utxo.clone(),
130                     action: symbol_short!("create"),
131                     amount,
132                 }
133                 .publish(&e);
134             }
135         });
136
```

```
137         assert_with_error!(
138             &e,
139             total_available_balance == expected_outgoing,
140             MoonlightError::UnbalancedBundle
141         );
```

Moonlight-Protocol/soroban-core/modules/storage/src/lib.rs

Line 74 to 116 in d65780b

```rust
    /// Creates a new unspent UTXO with the provided amount.
    ///
    /// # Panics
    ///
    /// Panics if the amount is not positive or if a record already exists for
    /// the UTXO key (including a spent record, which can never be recreated).
    pub fn create(&mut self, utxo65: &BytesN<65>, amount: i128) {
        if amount <= 0 {
            panic_with_error!(&self.env, Error::InvalidCreateAmount);
        }

        let k = self.utxo_key(utxo65);

        if self.env.storage().persistent().get::<_, i128>(&k).is_some() {
            panic_with_error!(&self.env, Error::UtxoAlreadyExists);
        }

        self.env.storage().persistent().set(&k, &amount);
        self.bump_ttl(&k);
    }

    /// Spends an existing unspent UTXO and returns its amount.
    ///
    /// The entry is tombstoned in place (amount set to `0`) rather than removed,
    /// so the spent record keeps blocking re-spend and re-creation. Its TTL is
    /// refreshed so the tombstone survives long idle periods (MOON-02).
    ///
    /// # Panics
    ///
    /// Panics if the UTXO does not exist or was already spent.
    pub fn spend(&mut self, utxo65: &BytesN<65>) -> i128 {
        let k = self.utxo_key(utxo65);
        match self.env.storage().persistent().get::<_, i128>(&k) {
            Some(amount) if amount > 0 => {
                self.env.storage().persistent().set(&k, &0i128);
                // Keep the spent record alive so the UTXO cannot be recreated after archival.
                self.bump_ttl(&k);
                amount
            }
            Some(_) => panic_with_error!(&self.env, Error::UtxoAlreadySpent),
            None => panic_with_error!(&self.env, Error::UtxoDoesNotExist),
```

```
115         }
116     }
```

Amount positivity alone is enforced in three separate places (create in `process_bundle`, create in `Store::create`, and deposit/withdraw in `pre_process_channel_operation`). There is no security impact (behavior is correct), but the same invariants must be tracked across multiple layers, which raises the maintenance and implementation cost, and the repeated reads add minor per-UTXO gas.

Recommendation

Establish a single source of truth for per-UTXO validation, and either let `Store::create/spend` be the sole enforcers and drop the pre-checks in `process_bundle` (which already delegates to the store), or make the store methods unchecked and validate only in `process_bundle`. Either choice removes the double read and double check. Additionally, group the bundle's structural checks (conflict, deduplication, sequence-equality, positivity) behind one clearly-named validation step so the invariants are enforced and discoverable in one place.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/43 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/7374602) by consolidating the bundle-level validation into a single validator.

[B6] Provider signature deadline is not provider-signed and enforced only against cooperative submitters

Severity: Informative | Recommended Action: Document Prominently | Addressed by client

Description

The provider check rejects a `Provider` entry whose `valid_until_ledger` is below the current ledger, but that field is not covered by any provider signature:

Moonlight-Protocol/soroban-core/modules/auth/src/core.rs

Line 215 to 222 in d65780b

```
215                 let (sig_variant, valid_until_ledger) =
216                     sig_map.get(signer.clone()).ok_or(Error::MissingSignature)?;
217
218                 if valid_until_ledger < e.ledger().sequence() {
219                     return Err(Error::SignatureExpired);
220                 }
221
222                 verify_signature(&e, &signer, &sig_variant, &payload)?;
```

The provider signs only `payload` (the host authorization payload). The `valid_until_ledger` paired with it in the `Signatures` map is attached by the transaction submitter, who can set any value. So this check constrains only a submitter acting against its own interest: it cannot enforce a deadline the provider chose. (A `P256` entry differs, as there the deadline is bound into the signed message.)

Neither direction has security impact. Expiry and replay protection for provider signatures are enforced separately, at the host layer: the entry's `signature_expiration_ledger` is part of the signed authorization preimage,

stellar/rs-soroban-env/soroban-env-host/src/auth.rs

Line 2030 to 2048 in cf58d53

```rust
    fn get_signature_payload(&self, host: &Host) -> Result<[u8; 32], HostError> {
        let (nonce, live_until_ledger) = self.nonce.ok_or_else(|| {
            host.err(
                ScErrorType::Auth,
                ScErrorCode::InternalError,
                "unexpected missing nonce",
                &[],
            )
        })?;
        let payload_preimage =
            HashIdPreimage::SorobanAuthorization(HashIdPreimageSorobanAuthorization {
                network_id: Hash(host.with_ledger_info(|li| li.network_id.metered_clone(host))?),
                nonce,
                signature_expiration_ledger: live_until_ledger,
                invocation: self.root_invocation_to_xdr(host)?,
            });

        host.metered_hash_xdr(&payload_preimage)
    }
```

and the host checks it and consumes the entry's nonce before the contract runs:

stellar/rs-soroban-env/soroban-env-host/src/auth.rs

Line 1985 to 2015 in cf58d53

```rust
    fn verify_and_consume_nonce(&mut self, host: &Host) -> Result<(), HostError> {
        if self.is_transaction_source_account {
            return Ok(());
        }
        if let Some((nonce, live_until_ledger)) = &self.nonce {
            let ledger_seq = host.with_ledger_info(|li| Ok(li.sequence_number))?;
            if ledger_seq > *live_until_ledger {
                return Err(host.err(
                    ScErrorType::Auth,
                    ScErrorCode::InvalidInput,
                    "signature has expired",
                    &[
                        self.address.into(),
                        ledger_seq.try_into_val(host)?,
                        live_until_ledger.try_into_val(host)?,
                    ],
                ));
            }
            let max_live_until_ledger = host.max_live_until_ledger()?;
            if *live_until_ledger > max_live_until_ledger {
                return Err(host.err(
                    ScErrorType::Auth,
                    ScErrorCode::InvalidInput,
                    "signature expiration is too late",
                    &[
                        self.address.into(),
                        max_live_until_ledger.try_into_val(host)?,
                        live_until_ledger.try_into_val(host)?,
                    ],
                ));
            }
```

So a submitter-inflated `valid_until_ledger` cannot extend a provider signature's life past the host-enforced, provider-signed expiry, and a deflated one only dooms the submitter's own transaction.

The contract-level field is therefore a second deadline that no provider signature covers, distinct from the host-enforced one. The residual concern is misreading: an integrator could take the in-contract check for the operative expiry control and reason about provider-signature lifetime from the wrong field.

Recommendation

Document at the check that this field is submitter-supplied and that the enforced provider deadline lives at the host layer over the signed authorization preimage, so no reader mistakes it for a binding control. If a provider-chosen contract-level deadline is in fact wanted, bind `valid_until_ledger` into the provider's signed message (as the `P256` path already does for its own deadline); the check then enforces a deadline the provider actually chose rather than one the submitter supplies.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/47 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/cc765e7) by documenting that `valid_until_ledger` is submitter-supplied and non-binding.

[B7] Provider registration accepts addresses the provider check can never match

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

Description

`ChannelAuthContract.add_provider` stores any `Address` in the provider registry without validating its kind:

Moonlight-Protocol/soroban-core/contracts/channel-auth/src/contract.rs

Line 112 to 117 in d65780b

```
112     pub fn add_provider(e: &Env, provider: Address) {
113         ownable::enforce_owner_auth(e);
114         let addr = provider.clone();
115         Self::register_provider(e, provider);
116         ProviderAdded { provider: addr }.publish(e);
117     }
```

The provider check of `__check_auth`, however, never looks up the registered address as submitted. For each `Provider`-keyed signature it derives a Stellar account address from the signer key's Ed25519 public key and matches the derived address against the registry:

Moonlight-Protocol/soroban-core/modules/auth/src/core.rs

Line 206 to 214 in d65780b

```
206         for signer in sig_map.keys().iter() {
207             if let SignerKey::Provider(pk32) = signer.clone() {
208                 let provider_addr: Address = address_from_ed25519_pk_bytes(&e, &pk32);
209 
210                 assert_with_error!(
211                     e,
212                     Self::is_provider(&e, provider_addr.clone()),
213                     Error::ProviderNotRegistered
214                 );
```

Moonlight-Protocol/soroban-core/modules/helpers/src/parser.rs

Line 4 to 9 in d65780b

```rust
pub fn address_from_ed25519_pk_bytes(e: &Env, provider_pk32: &BytesN<32>) -> Address {
    Address::from_payload(
        e,
        AddressPayload::AccountIdPublicKeyEd25519(provider_pk32.clone()),
    )
}
```

Since `SignerKey::Provider` carries a raw Ed25519 public key, only Ed25519 account addresses can ever match. Any other address — a contract address, for instance — is accepted by `add_provider` and reported as registered by `is_provider`, but the entry grants nothing, and both calls succeed without indication. Provider-gated calls then fail with `ProviderNotRegistered` until the correct account address is registered.

Recommendation

Validate the address kind at registration, so `add_provider` rejects any address the provider check cannot match. The `address_to_ed25519_pk_bytes` helper already performs exactly this discrimination, panicking with `NotEd25519AccountAddress` on a contract address:

Moonlight-Protocol/soroban-core/modules/helpers/src/parser.rs

Line 11 to 19 in d65780b

```rust
pub fn address_to_ed25519_pk_bytes(e: &Env, addr: &Address) -> BytesN<32> {
    match addr.to_payload() {
        Some(AddressPayload::AccountIdPublicKeyEd25519(provider_pk32)) => provider_pk32,
        Some(AddressPayload::ContractIdHash(_)) => {
            panic_with_error!(e, MoonlightError::NotEd25519AccountAddress)
        }
        None => panic_with_error!(e, MoonlightError::UnsupportedAddressPayload),
    }
}
```

Alternatively, since registration is owner-only and recoverable through `remove_provider`, this can be documented as an operational constraint: `add_provider` accepts only the Ed25519 account address of a provider signing key, and an entry reported by `is_provider` does not imply it can authorize.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/47 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/cc765e7) by validating the address kind in `add_provider`.

[B8] Argument-less authorization contexts are gated by the provider check alone

Severity: Informative | Recommended Action: Document Prominently | Addressed by client

Description

The UTXO check of `ChannelAuthContract.__check_auth` skips every authorization context that carries no arguments, inspecting neither the invoked contract nor the invoked function:

Moonlight-Protocol/soroban-core/modules/auth/src/core.rs

Line 80 to 90 in d65780b

```
80          for c in contexts.iter() {
81              if let Context::Contract(cc) = c {
82                  let sig_map = signatures.0.clone();
83
84                  if cc.args.len() < 1 {
85                      // MOON-03: a context with no auth-requirements arg carries no UTXO
86                      // requirements, but it must only skip THIS context — never short-circuit the
87                      // whole check. A `return Ok(())` here would let an empty-args context that
88                      // precedes a spend-bearing context bypass the latter's P256 verification.
89                      continue;
90                  }
```

A skipped context reaches no operation-binding check, so the provider check — one registered provider signature — is its only gate.

The skip is deliberate. As the `MOON-03` comment records, it must `continue` rather than `return Ok(())`, or an argument-less context preceding a spend-bearing one would bypass the latter's P256 verification. And the shape legitimately occurs: on spend-free bundles the channel requests authorization from this account with an empty argument vector:

Moonlight-Protocol/soroban-core/modules/utxo-core/src/core.rs

Line 87 to 93 in d65780b

```
87          let auth_args = if bundle.req.0.is_empty() {
88              vec![&e]
89          } else {
90              vec![&e, bundle.req.clone().into_val(e)]
91          };
92
93          Self::auth(&e).require_auth_for_args(auth_args);
```

For that intended case the single-provider gate is the intended authorization: such a bundle carries no P256 spend requirements, and the balance check, depositor authorizations, and signed-effects binding constrain the rest.

It becomes an issue only if this account authorizes anything else: a zero-argument entrypoint of another contract gated on it is approved by one provider signature, with the owner and every P256 holder uninvolved. Each approval still requires a fresh provider signature over the host payload, which commits to the specific invocation, a nonce, and an expiration — so the exposure is a lowered signing threshold, not an open gate.

Recommendation

Document the constraint: this account should be named as authorizer only where a single-provider gate on an argument-less invocation is acceptable, the channel's spend-free path being the intended case.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/47 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/cc765e7) by documenting the constraint.

[B9] P256 signatures verified by the UTXO check carry no replay protection

Severity: Informative | Recommended Action: Document Prominently | Addressed by client

Description

The UTXO check of `ChannelAuthContract.__check_auth` verifies a `P256` signature against a message binding the condition list, the entry's `valid_until_ledger`, and the requesting contract, and consumes nothing:

Moonlight-Protocol/soroban-core/modules/auth/src/core.rs

Line 107 to 127 in d65780b

```
107                     match signer.clone() {
108                         SignerKey::P256(_signer_pk) => {
109                             // Lookup signature by key.
110
111                             let (sig_variant, valid_until_ledger) =
112                                 sig_map.get(signer.clone()).ok_or(Error::MissingSignature)?;
113
114                             if valid_until_ledger < e.ledger().sequence() {
115                                 return Err(Error::SignatureExpired);
116                             }
117
118                             let auth_payload = AuthPayload {
119                                 conditions: conds,
120                                 live_until_ledger: valid_until_ledger,
121                             };
122
123                             let msg =
124                                 hash_payload(&e, &auth_payload, &caller_contract_bytes.clone());
125
126                             verify_signature(&e, &signer, &sig_variant, &msg)?;
127                         }
```

An unexpired `P256` entry of `signatures` therefore re-verifies identically in every authorization until its deadline passes. The host nonce does not close this: it is consumed per authorization entry, and a submitter can wrap the same P256 signature in a fresh entry with a fresh nonce and provider countersignature. Reuse is confined to the requesting contract and condition list the signed message binds, and to the signer-chosen deadline window.

Under the documented wiring this is not exploitable, because the channel supplies single-use itself. The signer key is the UTXO key; a settled spend tombstones its entry in place (amount zeroed, TTL refreshed), and a later bundle spending that key fails:

Moonlight-Protocol/soroban-core/modules/storage/src/lib.rs

Line 104 to 116 in d65780b

```rust
    pub fn spend(&mut self, utxo65: &BytesN<65>) -> i128 {
        let k = self.utxo_key(utxo65);
        match self.env.storage().persistent().get::<_, i128>(&k) {
            Some(amount) if amount > 0 => {
                self.env.storage().persistent().set(&k, &0i128);
                // Keep the spent record alive so the UTXO cannot be recreated after archival.
                self.bump_ttl(&k);
                amount
            }
            Some(_) => panic_with_error!(&self.env, Error::UtxoAlreadySpent),
            None => panic_with_error!(&self.env, Error::UtxoDoesNotExist),
        }
    }
```

So a P256 signature that has settled once can never authorize a second spend of the same UTXO, and the reusability of the check never surfaces.

The single-use guarantee, however, lives entirely in the requesting contract, not in `__check_auth`. For any other contract naming this account as authorizer, an unexpired P256 signature authorizes unboundedly many invocations within its deadline window — each still requiring a provider countersignature — unless that contract enforces its own single-use.

Recommendation

No change is required for the paired channel: the tombstone is a complete single-use mechanism, and it is the property the system's safety rests on here. Document that `__check_auth` provides no replay protection of its own for `P256` entries, so any contract naming this account as authorizer must supply its own single-use (as the channel does through the UTXO tombstone) or bound the reuse another way.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/47 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/cc765e7) by documenting the constraint.

[B10] Deposit amounts are not authorized at the channel, only by the asset transfer

Severity: Informative | Recommended Action: Document Prominently | Addressed by client

Description

A deposit entry `(from, amount, conditions)` splits the depositor's consent across two authorizations. The channel requires `from` to authorize only `[conditions]`; the amount appears solely in the immediately following `transfer(from, contract, amount)` and is covered only by the asset's transfer authorization:

Moonlight-Protocol/soroban-core/contracts/privacy-channel/src/transact.rs

Line 203 to 207 in d65780b

```
203     for (from, amount, deposit_conditions) in deposit.iter() {
204         from.require_auth_for_args(vec![&e, deposit_conditions.into_val(e)]);
205         asset_client.transfer(&from, &e.current_contract_address(), &amount);
206         increase_supply(&e, amount);
207     }
```

Under the documented wiring this is sound. The trust model pins the asset to a Stellar Asset Contract and trusts it to enforce its transfer semantics, and an SAC's `transfer` requires `from` to authorize the amount:

Failed to load

It becomes an issue only under an asset that moves value without requiring the authorization of `from`. Nothing at the channel then binds the amount to anything the depositor signed: a composer holding a signed `conditions` context can pair it with any amount, the channel check passes, the signed conditions execute, and the value above them becomes unsigned residual the composer allocates. (An `ExtDeposit` condition can carry an amount, but nothing compares it to the tuple's `amount`.)

Recommendation

Bind the amount into the depositor's channel authorization. It is in scope at the `require_auth_for_args` call site; pass it alongside `conditions` (or as a struct carrying both). This is a small, local change that makes the depositor's amount consent rest on the channel's own guarantee regardless of the asset's authorization behavior. It is defense-in-depth, not a correctness fix: under a compliant asset the amount is simply authorized twice, and value conservation still rests on the asset trust assumption, so asset vetting remains necessary for other reasons.

Alternatively, leave the amount consent with the asset and document the constraint: an asset that transfers without requiring `from`'s authorization must not be used as a channel asset.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/46 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/0d59e24) by documenting the constraint.

[B11] Withdrawals to the channel's own address burn claims without moving tokens

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

Description

No path of `transact` compares a withdrawal recipient against the channel's own address. The withdraw loop transfers to `to` and decrements `Supply`, whatever `to` is:

Moonlight-Protocol/soroban-core/contracts/privacy-channel/src/transact.rs

Line 209 to 230 in d65780b

```
209     for (to, amount, _) in withdraw.iter() {
210         let args_val: Vec<Val> = vec![
211             e,
212             (&e.current_contract_address()).into_val(e),
213             (&to).into_val(e),
214             (&amount).into_val(e),
215         ];
216
217         e.authorize_as_current_contract(vec![
218             &e,
219             InvokerContractAuthEntry::Contract(SubContractInvocation {
220                 context: ContractContext {
221                     contract: asset.clone(),
222                     fn_name: Symbol::new(e, "transfer"),
223                     args: args_val.clone(),
224                 },
225                 sub_invocations: vec![e],
226             }),
227         ]);
228         asset_client.transfer(&e.current_contract_address(), &to, &amount);
229         decrease_supply(&e, amount);
230     }
```

A `withdraw` naming the channel itself thus executes as `transfer(contract, contract, amount)` — net-zero under a compliant asset — while `Supply` still decreases by `amount`, which the bundle balance still requires be funded by spends or deposits. Claims burn without tokens moving; afterwards the channel's token balance exceeds `Supply` by `amount`.

The surplus is unreachable in the contract as deployed: the withdraw loop is the only outbound-transfer site, no allowance is ever granted, and every payout is coupled to an equal reduction of claims. Tokens sent to

the channel directly through the asset, outside `transact`, produce the same unreachable surplus. In either case the surplus is recoverable only by an admin-driven contract upgrade, which inherits the balance and could sweep or re-account it.

Recommendation

Reject a withdrawal naming the channel's own address — check `to != e.current_contract_address()` in the withdraw loop or in `verify_external_operations` — failing the bundle. This closes the in-band path at one site.

The direct-transfer variant cannot be prevented by the contract. The surplus it creates is inert — never payable out in deployed code — so it is at most a donation the admin can eventually recover, needing no handling beyond awareness that the token balance may exceed `Supply`.

Status

Fixed in https://github.com/Moonlight-Protocol/soroban-core/pull/46 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/0d59e24) by rejecting any withdrawal naming the channel's own address in `verify_external_operations` (`WithdrawToChannelAddress`).

[B12] `enable_channel` / `disable_channel` are event-only

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

Description

`disable_channel` (and `enable_channel`) only emit `ChannelStateChanged`; they write no state:

Moonlight-Protocol/soroban-core/contracts/channel-auth/src/contract.rs

Line 135 to 155 in d65780b

```
135     pub fn enable_channel(e: &Env, channel: Address, asset: Address) {
136         ownable::enforce_owner_auth(e);
137         ChannelStateChanged {
138             channel,
139             asset,
140             enabled: true,
141         }
142         .publish(e);
143     }
144
145     /// Disable an asset `channel`. The channel becomes withdraw-only (new deposits/sends rejected);
146     /// that enforcement lives provider-side. Emits `ChannelStateChanged { enabled: false }`.
147     pub fn disable_channel(e: &Env, channel: Address, asset: Address) {
148         ownable::enforce_owner_auth(e);
149         ChannelStateChanged {
150             channel,
151             asset,
152             enabled: false,
153         }
154         .publish(e);
155     }
```

`transact` never reads any enabled/disabled flag, so a "disabled" channel keeps accepting deposits, transfers, and withdrawals on-chain:

Moonlight-Protocol/soroban-core/contracts/privacy-channel/src/contract.rs

Line 107 to 119 in d65780b

```rust
    pub fn transact(e: Env, op: ChannelOperation) {
        bump_instance_ttl(&e);
        enter_reentrancy_guard(&e);

        let (utxo_op, total_deposit, total_withdraw) =
            pre_process_channel_operation(&e, op.clone());

        Self::process_bundle(&e, utxo_op.clone(), total_deposit, total_withdraw);

        execute_external_operations(&e, op.deposit, op.withdraw);

        exit_reentrancy_guard(&e);
    }
```

The disable is advisory, enforced off-chain by providers. Consequently, there is no on-chain way to halt a channel in an incident (compromised asset, discovered bug, misbehaving provider). The only on-chain lever is a full `upgrade`, which is itself the heaviest trust operation.

Recommendation

If an on-chain halt is desired, add an owner-settable `enabled` flag in the channel and check it in `transact` (at minimum gating deposits and transfers, leaving withdrawals open for exit). Otherwise, document explicitly that channel disabling is advisory/off-chain only and that `upgrade` is the sole on-chain stop.

Status

The client has acknowledged this finding. Still, in https://github.com/Moonlight-Protocol/soroban-core/pull/48 (head at https://github.com/Moonlight-Protocol/soroban-core/commit/32d929e), the project now documents its design decision to keep disable_channel and enable_channel event-only operations.

Appendix

This appendix gathers supplementary reference material that supported the review but does not form part of the findings. It is included for context only and introduces no new issues, severities, or recommendations. The Protocol Summary that follows provides a consolidated overview of the Moonlight core contracts (their roles, on-chain state, and the deposit, private-transfer, and withdrawal lifecycle) as understood during the engagement, and may serve as a reference material for readers new to the protocol before returning to the findings above.

Protocol Summary

API documentation of the `Moonlight Protocol`, a pair of Soroban contracts. `PrivacyChannelContract` is the vault: it custodies the pooled tokens of one asset, tracks per-UTXO spend state, and executes all value movement through its single transactional entrypoint `transact`. `ChannelAuthContract` is a custom account serving as the channel's authorizer: it holds the provider registry and approves or rejects authorization requests through `__check_auth`. The two are wired at deployment: the channel delegates spend and create authorization to the account stored in `UtxoAuth`, which the Moonlight deployment fills with a `ChannelAuthContract` instance.

Deployment. The two contracts are deployed as a pair: a `ChannelAuthContract` instance first, its address then passed to `__constructor` as the authorizer of the channel, where it is held in `UtxoAuth` for the lifetime of the channel (`utxo_auth_always_set`). The pairing itself is a deployment choice, enforced by neither source, so the guarantees of this preamble are conditional on it: they hold when the slot holds a `ChannelAuthContract` as documented here. Under that wiring the provider registry becomes the channel's chokepoint: `transact` requires the authorization of the auth contract on every invocation, spend-free bundles included (`spend_authorization_delegated`), and the auth contract approves no request without at least `PROVIDER_THRESHOLD` registered provider signatures (fixed at `1`) (`provider_quorum`). Providers are registered by the owner of the auth contract (`add_provider` , `AuthorizedProvider`), so who may drive a channel is an administrative decision recorded on-chain.

UTXO lifecycle. `PrivacyChannelContract` tracks every UTXO as one `UTXO` entry, keyed by the hash of its 65-byte P-256 public key: a positive amount while unspent, a permanent `0` tombstone once spent. A UTXO comes into existence as an effect of a bundle, in one of three ways. A spender demands it, as a `Create` condition of its spend, enforced by the signed-effects binding (`signed_effects_executed`). A depositor demands it, as a `Create` condition covered by its own authorization (`depositors_authorized`) and enforced the same way. Or the bundle composer adds it without any signature, confined by the balance to the value the signed conditions leave unallocated (`bundle_balanced`). The created key takes no part in its own creation: receiving requires no signature and no on-chain act, and owning the new UTXO is nothing but knowing the matching secret key. No creation path validates the key: any 65 bytes are recorded, and under this wiring value created under bytes that do not encode a well-formed P-256 public key is locked permanently, since no signature can verify under a malformed key (`utxo_keys_not_validated`).

To spend a UTXO, its owner signs once: over the condition list of the spend, a deadline, and the channel address. The signature commits to the effects and to nothing else, not the enclosing bundle, not the submitter, not the amount spent (cf. the `hash_payload` discussion at `__check_auth`). Under the same wiring, the signature then travels the following chain from consent to settlement:

1. `transact` forwards the condition list of every spend as the auth requirement of the spent key (`spend_authorization_delegated`).

2. The host resolves that requirement by invoking `__check_auth`.
3. The auth contract reconstructs the signed message from the forwarded conditions and verifies an unexpired P-256 signature of the spent key for it (`utxo_requirements_checked`). Reconstruction succeeding proves that the conditions being executed are the conditions the owner signed, exactly up to variant-stable reordering and the collisions of the non-injective `hash_payload` encoding (cf. `hash_payload_not_injective`).
4. The signed `Create` and `ExtWithdraw` conditions are forced into execution (`signed_effects_executed`), so a composer can batch independently signed intents but never redirect their outcomes, and the balance confines everything unsigned to the value the signed conditions leave unallocated (`bundle_balanced`).
5. The spend is unrepeatable: the UTXO entry is tombstoned permanently (`utxo_spend_permanent`), and since the key is the signer identity, the signature can never authorize again (cf. `p256_signatures_reusable_in_check`). Provider replay protection comes from the host instead: provider signatures cover a nonce-carrying payload committed to the specific invocation (`assume_check_auth_payload`).

Composed, this yields the guarantee carried by every settled bundle. For every successful `transact` invocation: the spent amounts plus the deposits equal the created amounts plus the withdrawals (`bundle_balanced`), every spend is backed by an unexpired P-256 signature of its UTXO key over the conditions its owner consented to (a binding exact only up to variant-stable reordering and the collisions of the non-injective `hash_payload` encoding, cf. `hash_payload_not_injective`), which the bundle must execute (`utxo_requirements_checked`, `signed_effects_executed`), and a registered provider has signed the invocation itself (`provider_quorum`). The first clause holds regardless of the deployed wiring. The other two hold under the wiring above and rest on the named assumptions of the auth contract (`assume_crypto_verify_traps`, `assume_check_auth_payload`).

Custody backs these guarantees. Outside a running invocation, `Supply` equals the sum of all unspent UTXO amounts (`supply_equals_unspent_total`), and under `assume_asset_token_compliant` the token balance of the channel never falls below it: the gap between balance and `Supply` starts non-negative at deployment and never decreases, since every balance-changing path the assumption admits either moves balance and counter equally (a deposit, a withdrawal to a third party) or widens the gap (a withdrawal naming the channel itself, a token transfer to the channel address outside `transact`, cf. `withdraw_to_channel_locks_value`). Every unspent UTXO is therefore at all times fully backed by tokens the contract holds: the failure mode of the surplus paths is over-collateralization, never a shortfall. Like the balance clause, this custody guarantee is independent of the deployed wiring.

On-chain trackability. A bundle emits no events (`BundleEvent` and `UtxoEvent` are compiled out of the deployed build), so the on-chain record of a bundle is the transaction itself, and that record is complete: the full `op` travels as a public invocation argument, with every spent key, every created key and amount, every

deposit and withdrawal address and amount, every condition list, and every signature of the authorization entries readable by any observer, as are the `UTXO` entries and `Supply` in ledger state. Within a bundle, everything is therefore linked by construction. Across bundles, a UTXO key links the bundle that created it to the bundle that spent it, so the hop graph of the pool is public. Signed conditions sharpen that graph: a `Create` or `ExtWithdraw` condition inside a spend publicly attributes that output to that spend (`signed_effects_executed`), so every consent-enforced flow is a public edge, and only outputs left unsigned (confined by the balance) are unattributed within their bundle. What the chain does not record is the binding of keys to parties: unlinkability rests on fresh keys, on bundles batching several parties, and on amount patterns not disambiguating flows, the standard limits of pooled mixing. The countersigning provider of every bundle is itself identified on chain by its signature.

How to read this document. This document is a side product of a structured audit of the contract sources: it records, entry by entry, what the review established. It is a best-effort account, not a warranty, and errors in the analysis itself are possible. The structure exists to make such errors findable: every behavioral claim is backed, invariants and postconditions carry proofs against the source, and properties that cannot be proven from the contract sources alone are isolated as named assumptions with their trust boundary stated (host behavior, build configuration, the deployed asset contract). Source links are pinned to commit `d65780b` of `Moonlight-Protocol/soroban-core`, so every claim can be re-checked against the exact code it describes. Every property holds only for those executables: either contract can be upgraded by its owner (cf. the upgrade notes of each contract). Contract bodies are deployment-agnostic: each contract is documented against arbitrary counterparties, and all pairing knowledge is confined to this preamble and the findings.

Findings

Name	Status	Description
`hash_payload_not_injective`	acknowledged	The `hash_payload` encoding, over which `__check_auth` verifies every P-256 signature, is not injective: distinct condition lists can produce the same signed message. The encoding writes the strkey of the requesting contract, then the four condition buckets (`Create` , then `ExtDeposit` , `ExtWithdraw` , `ExtIntegration`) back to back, then the `valid_until_ledger` , with no length prefixes, condition tags, or bucket separators in between. Every boundary is therefore positional: the byte stream does not record where one bucket, or one condition, ends and the next begins. Two condition lists yield one message in two cases. Lists that are variant-stable reorderings of each other (identical per-variant subsequences) encode identically, since the bucketing preserves order only within each bucket. Beyond reordering, distinct lists collide whenever their four buckets concatenate to the same bytes, for example across buckets (bytes contributed under one variant serving unchanged as bytes of another) or within the length-prefix-free `ExtIntegration` bucket (shifted condition boundaries). A P-256 signature verified by `__check_auth` therefore binds the signed condition list only up to variant-stable reordering and these collisions.
`pending_transfer_max_window`	acknowledged	Both contracts hardcode the acceptance deadline of a pending ownership transfer to the maximum: `set_admin` and `set_admin` pass the maximum live-until ledger, the farthest a storage entry can live, as the deadline of the underlying `transfer_role` helper, exposing no parameter for it. Every pending transfer therefore stays acceptable for the longest window the network allows (the maximum entry TTL, months of ledgers) instead of lapsing on a deadline the owner chose, and acceptance is possible at any point within it, as the deadline is an expiry, not a time-lock (cf. `accept_admin`). The window is pure exposure while a transfer lingers unaccepted, since a healthy transfer closes by acceptance: whoever holds the address recorded in `PendingOwner` or `PendingOwner` can take ownership at any time before expiry, so a later compromise of the pending key stays exploitable for the full maximum TTL. A finite deadline would cap that exposure on its own, whereas the only remedy here is an overwrite via `set_admin` or `set_admin` , which requires the owner to notice and react. The maximum deadline also makes the cancel path of the helper (a deadline of `0`) unreachable, so neither contract exposes a way to clear a pending transfer, only to redirect it. Both bounding the window and cancelling would require the deadline to be threaded through as a parameter of `set_admin` .
`utxo_keys_not_validated`	submitted	No path of `PrivacyChannelContract` validates a created UTXO key: `transact` records any 65-byte value as a UTXO public key (`UTXO`), whether or not it encodes a well-formed P-256 point. Under the documented wiring the recorded bytes are also the sole spending credential: `__check_auth` approves a spend only on a P-256 signature verifying under those bytes, and verification under a malformed key cannot succeed. Value created under a malformed key, for example a mistyped key in a signed `Create` condition, is therefore locked permanently once the bundle settles.
`withdraw_to_channel_locks_value`	submitted	No check of `transact` compares a withdrawal recipient against the channel address: a `withdraw` entry naming the channel itself executes as a transfer from the contract to the contract, a net-zero token movement under a compliant asset. The bundle balance still requires the amount to be funded by spends or deposits not allocated to creates, and `Supply` still decreases by it (`supply_updated`), so the bundle burns claims without moving tokens: afterwards the token balance of the channel exceeds `Supply` by the amount. The surplus is unreachable: the withdrawal loop of `transact` is the only outbound transfer site, no allowance is ever granted, and every payout is coupled by the balance to an equal reduction of claims (`bundle_balanced` , `supply_equals_unspent_total`). Tokens sent to the channel address directly through the asset contract, outside `transact` , produce the same unreachable surplus.
`argless_contexts_provider_gated`	submitted	The UTXO check of `__check_auth` skips every authorization context that carries no arguments: it inspects neither the invoked contract nor the invoked function. For such contexts the provider check is the only gate, so the account approves any argument-less invocation of any contract in its name on the signature of a single registered provider (`PROVIDER_THRESHOLD`), with no channel state to narrow the approval. The skip is load-bearing under the documented wiring: a spend-free bundle makes `transact` request authorization with an empty argument vector, so its context carries no arguments, and rejecting the shape would reject such bundles. A context with arguments is constrained by comparison, as its first argument must decode as an `AuthRequirements` map (`BadArg`). The approval surface matters wherever the account address is granted authority in further contracts: a zero-argument entrypoint gated on this account is authorized by one provider signature alone, with the owner and every other check uninvolved. Each such approval still requires a fresh provider signature over the host payload, which commits to the specific invocation, a nonce, and an expiration (`assume_check_auth_payload`).

`identical_signed_effects_merge`	submitted	The signed-effects binding of `transact` compares conditions as a set keyed by canonical XDR bytes: byte-identical `Create` or `ExtWithdraw` conditions carried by distinct `spend` or `deposit` entries collapse to one element, discharged by a single executed occurrence (`signed_effects_executed`). For `ExtWithdraw` the merge is also the only way byte-identical conditions can settle in one bundle: two `withdraw` entries to one address are rejected (`RepeatedAccountForWithdraw`), and a single entry over the summed amount encodes to different bytes, matching neither signed condition (`UnauthorizedOperation`). The balance still counts every spent and deposited amount in full (`bundle_balanced`), so when two signers each release value demanding the identical `ExtWithdraw(to, amount)` , the bundle pays `to` once and the difference becomes unsigned residual, allocatable by the bundle composer to any recipient. Which signer the executed payout belongs to is not represented on chain. A composer holding a signed spend can also produce the merge: adding its own spend carrying the identical condition bytes satisfies the binding with the one payout and frees the matching value as residual. Byte identity requires equal address and equal amount, a combination that arises when several parties withdraw a common amount to a shared address. `Create` conditions merge under the same rule, and identical `Create` bytes require the same 65-byte key. The contract distinguishes none of this: uniqueness of the signed condition bytes within a bundle is the only discriminator, and any screening of co-batched conditions happens outside the contract.
`deposit_amount_not_channel_authorized`	submitted	A deposit entry `(from, amount, conditions)` of `transact` splits the consent of `from` across two authorizations. The channel itself requires authorization over the argument vector `[conditions]` and over nothing else of the bundle: the amount is not part of that context (`depositors_authorized`). The amount is covered only by the transfer authorization of the asset: a Stellar Asset Contract and any SEP-41 compliant token require the authorization of `from` for `transfer(from, contract, amount)` , and that requirement is the only place the amount appears. `assume_asset_token_compliant` states value conservation and does not include such gating, so the documented properties alone bound the executed amount by nothing the depositor signed: under an asset that moves value without requiring the authorization of `from` , a composer holding a signed conditions context can pair it with any amount, the channel check passes, the signed conditions are executed (`signed_effects_executed`), and the value above them is unsigned residual. The amount consent of depositors is therefore part of what vetting an asset for a channel covers: it rests on the transfer authorization of the deployed asset.
`provider_expiry_submitter_supplied`	submitted	The provider check of `__check_auth` rejects a `Provider` -keyed entry of `signatures` whose `valid_until_ledger` is below the current ledger sequence (`SignatureExpired`), but the provider signature covers only the host authorization payload: the `valid_until_ledger` of the entry is not part of any signed message. The map is assembled by the submitter, who can attach any deadline to a provider signature, so the check constrains only submitters acting against themselves: it cannot enforce a deadline chosen by the provider. A `P256` -keyed entry differs, as there the deadline is bound into the signed message. The effective expiry and replay protection of provider signatures sit at the host layer: the signed payload commits to a nonce and to the `signature_expiration_ledger` of the authorization entry, both checked and the nonce consumed by the host after a successful call (`assume_check_auth_payload`). A provider deadline is therefore chosen and enforced at the host layer: the contract-level field is a second deadline that no provider signature covers.
`p256_signatures_reusable_in_check`	submitted	The UTXO check of `__check_auth` verifies a `P256` signature against a message binding the condition list, the `valid_until_ledger` of the entry, and the requesting contract, and consumes nothing: an unexpired entry of `signatures` verifies identically in every authorization until its deadline passes. The host nonce does not close this: it is consumed per authorization entry (`assume_check_auth_payload`), and a submitter can wrap the same `P256` signature in a fresh entry with a fresh nonce and a fresh provider countersignature. Reuse is confined to the requesting contract, which the signed message binds, and to the window the signer chose, as the deadline is covered by the signature. Single-use semantics come only from the requesting contract itself. Under the documented wiring the channel provides them: the signer key is the UTXO key, a settled spend tombstones its entry permanently (`utxo_spend_permanent`), and a bundle spending a tombstoned key fails (`UtxoAlreadySpent`), so a settled signature never authorizes again. For any other contract naming this account as authorizer, an unexpired `P256` signature authorizes unboundedly many invocations within its window unless that contract enforces its own single-use.
`provider_registration_ed25519_only`	submitted	Provider registration and provider matching disagree on the accepted key space. `add_provider` registers any `Address` in `AuthorizedProvider` unvalidated, but the provider check of `__check_auth` matches a `Provider` signer key against the registry by deriving the Stellar account address from the Ed25519 public key of that key. Registration is therefore effective only for the account address of an Ed25519 provider signing key: every other registered `Address` , a contract address for instance, can never match, and the mismatch is silent, as `add_provider` succeeds and `is_provider` reports the entry as registered. An ineffective entry grants nothing, and a registry holding only ineffective entries disables approval entirely: at least `PROVIDER_THRESHOLD` passing `Provider` -keyed entries are required and none can pass membership, so `__check_auth` rejects every authorization (`ProviderNotRegistered` against any submitted provider signature, `ProviderThresholdNotMet` without one). Under the documented wiring that blocks every bundle of the paired channel, spend-free ones included, until the registry holds an effective provider again. The state is recoverable, not permanent: the registry stays mutable through `add_provider` and `remove_provider` , but value in the channel is unreachable while it lasts.

`PrivacyChannelContract`

A Soroban privacy channel: a single-asset pool in which value circulates as UTXOs. The contract custodies the pooled tokens of the channel asset (`Asset`) and tracks per-UTXO spend state, with all value movement running through the single transactional entrypoint `transact` . All administration is gated on a single owner role.

Transactions. The core feature of the contract. `transact` executes a bundle: spend UTXOs, create UTXOs, pull deposits in, push withdrawals out, in one atomic, balance-checked invocation (`bundle_balanced` , structure at `op`). Authorization and enforcement rest on three mechanisms.

1. `UtxoAuth` must authorize every bundle, over auth requirements derived from the spends (`spend_authorization_delegated`).
2. Each depositor must authorize its own entry, bound to the conditions of that entry, with the amount covered only by the transfer authorization of the asset (`depositors_authorized` , `deposit_amount_not_channel_authorized`).
3. The signed-effects binding ties the signed `Create` and `ExtWithdraw` conditions to execution (`signed_effects_executed`).

The stages, the condition roles, and the raised errors are specified at `transact` .

UTXO ledger. Each UTXO is a 65-byte P-256 public key backed by one entry `UTXO(sha256(utxo)) -> state` : a positive amount while unspent, a permanent `0` tombstone once spent (`utxo_spend_permanent`). The key doubles as the signer identity under which `transact` requests spend authorization, so a fresh key per UTXO makes the spending credential single-use: once tombstoned, a key can neither be re-spent nor re-created (`UtxoAlreadySpent` , `UtxoAlreadyExists`). `utxo_balance` and `utxo_balances` surface the spend state (and extend the TTL of queried entries, cf. `UTXO`). `Supply` counts the net deposited amount, surfaced by `supply` .

Channel configuration. Two immutable references, both set at deployment by `__constructor` : the token contract `Asset` (`asset_always_set` , readable via `asset` , trusted only per `assume_asset_token_compliant`) and the authorizer account `UtxoAuth` (`utxo_auth_always_set` , readable via `auth`). The contract exposes no way to change either (short of `upgrade`).

Ownership. A single administrative role, `Owner` (surfaced as the admin in the interface), set at deployment by `__constructor` (which the owner must authorize) and readable via `admin` . Transfers are two-step: `set_admin` records a pending transfer in `PendingOwner` , `accept_admin` completes it. The role is never vacant (`owner_always_set`). Beyond deployment, its only privileges are `upgrade` and initiating the transfer.

Upgrade. `upgrade` replaces the contract executable with owner authorization, preserving the contract address and all storage. Every property in this document describes the current executable: invariants and postconditions hold only up to an upgrade, since a replacement executable is free to change them.

Assumptions

Name	Property	Description
`assume_max_live_until_ledger`	Every read of `e.ledger().max_live_until_ledger()` returns a value that is strictly positive, at least the current ledger sequence (`e.ledger().sequence()`), and exceeds the current sequence by at least the initial TTL granted to a newly created temporary entry (the minimum temporary-entry TTL of the network).	The maximum live-until ledger is the highest ledger sequence to which the TTL of a storage entry may be extended, computed by the host as the current ledger sequence plus the maximum entry TTL of the network (`max_entry_ttl` , a network configuration parameter) minus one. On any live Stellar network the maximum entry TTL is strictly positive and the ledger sequence is strictly positive, so the returned value is at least the current sequence and never `0` . Moreover, the maximum entry TTL (millions of ledgers) exceeds the minimum temporary-entry TTL (tens of ledgers) by orders of magnitude, giving the final clause. That clause is what lets `set_admin` end the TTL of a freshly written `PendingOwner` entry exactly at the returned deadline: the extension applies only because the deadline lies beyond the initial TTL of the fresh entry (cf. `TransferExpired`). A violation could arise only in an artificial environment such as a misconfigured test ledger. This is a property of the runtime environment, not of the contract source.
`assume_asset_token_compliant`	The token contract stored in `Asset` implements standard token semantics: a successful `transfer(from, to, amount)` moves exactly `amount` of the asset from the balance of `from` to the balance of `to` , and the balance of this contract changes through no path other than such transfers.	The contract accepts any address as `asset` at deployment and never validates it: `transact` simply invokes `transfer` on it, once per entry of `op.deposit` and once per entry of `op.withdraw` . All the contract source proves is that those invocations return successfully. Whether value actually moved, and how much, is behavior of the asset contract. The property holds for a Stellar Asset Contract without the clawback flag and for any compliant SEP-41 token. (A clawback-enabled Stellar Asset Contract violates the second clause: its issuer can reduce the balance of this contract through `clawback` , without any transfer and without the authorization of this contract, so such an asset satisfies the property only with the issuer inside the trust boundary.) A non-compliant or malicious asset can violate it freely, and the source treats the asset as untrusted: `transact` runs under the reentrancy guard (`ReentrancyGuard`), and a callback of the asset is additionally rejected by the host (cf. the note at `ReentrancyGuard`). Every claim about token movement in this document (cf. `tokens_transferred`) is conditional on this assumption, so a channel must be vetted asset-by-asset before deployment. The transfer authorization of the asset also carries the deposit amounts: no channel-side authorization covers them (`deposit_amount_not_channel_authorized`). This is a property of the deployed asset contract, not of the source of this contract.
`assume_overflow_checks`	The deployed Wasm is compiled with `overflow-checks = true` : every arithmetic overflow in contract code traps the transaction instead of wrapping silently.	The bundle balance accumulator of `process_bundle` in `moonlight-utxo-core` sums spent amounts and subtracts created amounts with plain (unchecked) `i128` arithmetic. With wrapping arithmetic, a wrapped accumulator could pass the `UnbalancedBundle` equality check even though the true sums differ, invalidating `bundle_balanced` . The workspace build configuration sets `overflow-checks = true` in the release profile, so any overflow on that path traps and fails the transaction. (The deposit and withdrawal totals are independent of this assumption: they are computed with `checked_add` behind `AmountOverflow` .) The assumption must be re-validated whenever the build profile changes or the contract is built outside the workspace profile. This is a property of the build configuration, not of the contract source.

Invariants

Name	Property	Proof
`owner_always_set`	`Owner` is always set.	Established by `__constructor`, which always sets `Owner`. `accept_admin` overwrites it but never unsets it (cf. `ownership_transferred`). No other function writes the slot, and there is no remover for it: `renounce_ownership`, the only remover in the `stellar-access` library, is not exposed by this contract.
`utxo_auth_always_set`	`UtxoAuth` is always set, and always holds the value set at deployment.	Established by `__constructor`, which always sets `UtxoAuth` (cf. `auth_established`). No other function writes the slot: the `set_auth` setter of `UtxoHandlerTrait` is not exposed as an entrypoint and is called by no entrypoint after construction, and there is no remover.
`asset_always_set`	`Asset` is always set, and always holds the value set at deployment.	Established by `__constructor`, which always sets `Asset` (cf. `asset_established`). No other function writes the slot: `write_asset_unchecked` has no caller besides the constructor, no entrypoint writes the key, and there is no remover.
`utxo_spend_permanent`	Every `UTXO` entry transitions only along `absent -> a -> 0` with `a > 0`: created with a positive amount on a previously unrecorded key (create), later overwritten with the `0` tombstone (spend). In particular a spent entry never changes again, and no entry is ever removed.	Within this contract every write of a `UTXO` entry goes through `Store::create` or `Store::spend` of `moonlight-storage`, both reachable only from `transact`. (The `create`, `spend`, `unchecked_create`, and `unchecked_spend` defaults of `UtxoHandlerTrait` write `UTXO` entries through the same helpers, but none is exposed as an entrypoint or called by one.) `Store::create` writes a positive amount (`InvalidCreateAmount` guard) and refuses any key with an existing record, spent or unspent (`UtxoAlreadyExists` guard). `Store::spend` overwrites an entry with `0` only if its current value is positive (`UtxoAlreadySpent` and `UtxoDoesNotExist` guards). No code path calls `remove` on a `UTXO` entry: the module contains no remover. (Expiry does not delete an entry either: a persistent entry whose TTL ends is archived by the network, and its key cannot be written again without restoring the entry. This is a property of the Stellar state-archival mechanism, not of the contract source.)
`supply_equals_unspent_total`	Outside a running `transact` invocation, `Supply` (reading unset as `0`) equals the sum of the amounts of all unspent `UTXO` entries. In particular `Supply` is never negative.	At deployment both sides are `0`: `__constructor` leaves `Supply` unset (`supply_untouched`) and writes no `UTXO` entry. Both sides change only in `transact` (`utxo_spend_permanent` for the entries, the `Supply` description for the counter), and a failed invocation rolls back all writes with the transaction, so it suffices that a successful bundle changes both sides equally. `Supply` changes by the total of `op.deposit` minus the total of `op.withdraw` (`supply_updated`), both totals computed with `checked_add` (`AmountOverflow`). The unspent total changes by the total of `op.create` minus the sum of the spent amounts: each create records a distinct new positive entry (`utxos_created`, `RepeatedCreateUtxo`, `UtxoAlreadyExists`), and each spend zeroes a distinct positive entry (`utxos_spent`, `RepeatedSpendUtxo`), distinct even under a `sha256` collision of two spent keys, which would read the tombstone of the first (`UtxoAlreadySpent`). The two deltas are equal by the balance check: the spent amounts plus the deposit total equal the create total plus the withdrawal total (`bundle_balanced`, exact under `assume_overflow_checks`). Non-negativity follows, as the unspent total is a sum of positive amounts. (An entry whose TTL lapses is archived, not deleted, cf. the proof of `utxo_spend_permanent`, so the sum ranges over all recorded entries.)

Constants

Name	Type	Value	Description
`DAY_IN_LEDGERS`	`u32`	`17_280`	Number of ledgers in approximately one day, at the nominal five-second ledger close time.
`INSTANCE_BUMP_AMOUNT`	`u32`	`7 * DAY_IN_LEDGERS`	TTL, in ledgers, to which the contract instance is extended by `__constructor` and `transact`. Approximately seven days.
`INSTANCE_LIFETIME_THRESHOLD`	`u32`	`INSTANCE_BUMP_AMOUNT - DAY_IN_LEDGERS`	Remaining-TTL threshold, in ledgers, below which the instance TTL is extended to `INSTANCE_BUMP_AMOUNT`. Approximately six days.
`Store_DAY_IN_LEDGERS`	`u32`	`17_280`	Number of ledgers in approximately one day, at the nominal five-second ledger close time. (Source name: `Store::DAY_IN_LEDGERS`.)
`Store_PERSISTENT_BUMP_AMOUNT`	`u32`	`30 * Store_DAY_IN_LEDGERS`	TTL, in ledgers, to which a `UTXO` entry is extended on every read or write of an existing (or just-created) entry, in `transact`, `utxo_balance`, and `utxo_balances`. Approximately thirty days. (Source name: `Store::PERSISTENT_BUMP_AMOUNT`.)
`Store_PERSISTENT_LIFETIME_THRESHOLD`	`u32`	`Store_PERSISTENT_BUMP_AMOUNT - Store_DAY_IN_LEDGERS`	Remaining-TTL threshold, in ledgers, below which a `UTXO` entry TTL is extended to `Store_PERSISTENT_BUMP_AMOUNT`. Approximately twenty-nine days. (Source name: `Store::PERSISTENT_LIFETIME_THRESHOLD`.)

Storage

Name	Type	Lifetime	Description
`Asset`	`Address`	`instance`	Address of the token contract of the channel: the asset deposited into and withdrawn from the channel by `transact`. Set once by `__constructor` and never changed thereafter (`asset_always_set`): the contract exposes no way to migrate the channel to a different asset. Read by `asset` and by every external leg of `transact`, which transfers this token (cf. `tokens_transferred`).
`Supply`	`i128`	`instance`	Supply counter of the channel, written only by `transact` through the checked `increase_supply` and `decrease_supply` helpers of the treasury module (cf. `supply_updated`). Left unset by `__constructor` (cf. `supply_untouched`): the absent entry reads as `0`, which `supply` surfaces. Outside a running invocation the value equals the sum of the amounts of all unspent `UTXO` entries, and is in particular never negative (`supply_equals_unspent_total`). The token balance of the channel can exceed the value: withdrawals naming the channel itself and tokens sent to its address outside `transact` create a surplus that no invocation can pay out (`withdraw_to_channel_locks_value`).
`ReentrancyGuard`	`bool`	`temporary`	Reentrancy guard of `transact`, a temporary entry under the symbol key `RGUARD`. (The name used in this document is coined: the source defines the key as the `REENTRANCY_GUARD` constant, without a data-key enum.) Set to `true` when `transact` is entered and removed when it exits, so a nested invocation would observe the entry and fail with `ReentrantCall`. The flag is set before any external interaction, so it covers the whole body: against a callback of the `Asset` contract invoked in stage 4 as much as against one of a `__check_auth` invoked while the host resolves an authorization (stages 2 and 4). Note that the host itself also rejects contract re-entry, before any code of the re-entered contract runs, so on current hosts a re-entering call fails in the host and `ReentrantCall` is never raised. The guard does not rest on that behavior: it makes `transact` non-reentrant from the contract source alone. Never observable outside a running invocation: a successful call removes it (`guard_cleared`), and on failure the temporary write rolls back with the transaction.
`UtxoAuth`	`Address`	`instance`	Address of the authorizer of UTXO spends and creates: the account to which `transact` delegates spend and create authorization via `require_auth_for_args`. An instance entry under the symbol key `UTXO_AUTH`. (The name used in this document is coined: the source defines the key as the `STORAGE_KEY_UTXO_AUTH` constant in `moonlight-utxo-core`, without a data-key enum.) Set once by `__constructor` and never changed thereafter (`utxo_auth_always_set`). Read by `auth` and on every invocation of `transact`.
`UTXO`	`Map<BytesN<32>, i128>`	`persistent`	Per-UTXO spend state, encoded as one persistent entry `UTXO(sha256(utxo)) -> state` per recorded UTXO: the key is the SHA-256 of the 65-byte UTXO public key, the value its `i128` state. A positive value is the amount of an unspent UTXO, `0` is a spent tombstone, and an absent entry means no record exists. A spent entry is tombstoned in place, never removed (`utxo_spend_permanent`), so a spent UTXO permanently blocks re-spending and re-creation of its key. No write validates the key bytes: any 65-byte value is accepted and recorded (`utxo_keys_not_validated`). Whether an entry recorded under bytes that encode no P-256 point can ever be spent is behavior of the account stored in `UtxoAuth`. Written only by `transact`: a create writes the amount to a previously absent key (`utxos_created`), a spend overwrites a positive value with `0` (`utxos_spent`). Read by `transact`, `utxo_balance`, and `utxo_balances`. Every read or write of an existing (or just-created) entry extends its TTL to `Store_PERSISTENT_BUMP_AMOUNT` (if it would otherwise expire sooner than `Store_PERSISTENT_LIFETIME_THRESHOLD`), so a holder keeps their UTXO alive simply by observing it, independently of every other UTXO. If no such access reaches an entry before its TTL ends, the entry is archived, not deleted: it leaves the live ledger with its value intact, and any transaction touching the key, a spend or a balance query alike, fails until the entry is restored. Restoration (a host operation, open to anyone willing to pay the fee) returns the entry unchanged, so a lapsed unspent UTXO is temporarily unspendable, never lost: spending it means restore first, then spend. Tombstones archive under the same rule, and an archived entry still blocks re-creation of its key (`utxo_spend_permanent`). Archival and restoration are behavior of the Stellar state-archival mechanism, not of the contract source.
`Owner`	`Address`	`instance`	Address of the current contract owner, surfaced as the admin in the contract interface. The sole administrative role of the contract: it performs contract upgrades (`upgrade`), initiates ownership transfers (`set_admin`), and authorizes the deployment itself (`__constructor`). Set at construction by `__constructor` and thereafter only by `accept_admin` when a transfer completes.
`PendingOwner`	`PendingTransfer`	`temporary`	The pending ownership transfer, held as a `PendingTransfer { address, live_until_ledger }` pair: the proposed new owner and the ledger sequence through which the transfer may be accepted. Set (or overwritten) by `set_admin` and cleared when the transfer completes (`accept_admin`). Its sole privilege: `PendingOwner.address` may complete the transfer by calling `accept_admin`, becoming the new owner. A temporary entry whose TTL normally ends at its `live_until_ledger` deadline, so an expired pending transfer disappears on its own.

Events

Event	Fields	Description
`Upgraded`	`wasm_hash: BytesN<32>` *(topic)* — Hash of the installed Wasm blob the contract was upgraded to.	Emitted by `upgrade` to record the hash of the Wasm the contract was upgraded to.
`BundleEvent`	`name: Symbol` *(topic)* — The constant symbol `"bundle"`, identifying the event kind. `spend: Vec<BytesN<65>>` — The public keys of the UTXOs spent by the bundle. `create: Vec<(BytesN<65>, i128)>` — The UTXOs created by the bundle, as `(utxo, amount)` pairs. `deposited: i128` — The total amount deposited by the bundle. `withdrawn: i128` — The total amount withdrawn by the bundle.	Defined in `moonlight-utxo-core` to be emitted once per `transact` bundle, recording the spent and created UTXOs and the external totals, but only when compiled without the `no-bundle-events` feature. This contract enables that feature, so the event is compiled out of the deployed build and never emitted. It is documented here because a rebuild without the feature would emit it unchanged.
`UtxoEvent`	`name: Symbol` *(topic)* — The constant symbol `"utxo"`, identifying the event kind. `utxo: BytesN<65>` — The 65-byte public key of the affected UTXO. `action: Symbol` — `"spend"` or `"create"`. `amount: i128` — The amount of the affected UTXO: the spent amount for a spend, the created amount for a create.	Defined in `moonlight-utxo-core` to be emitted once per UTXO spend and once per UTXO create in `transact`, but only when compiled without the `no-utxo-events` feature. This contract enables that feature, so the event is compiled out of the deployed build and never emitted. It is documented here because a rebuild without the feature would emit it unchanged.
`OwnershipTransfer`	`old_owner: Address` — The current owner, who initiated the transfer and authorized the call. `new_owner: Address` — The proposed new owner. `live_until_ledger: u32` — The ledger sequence through which the transfer may be accepted.	Emitted by `set_admin` when an ownership transfer is initiated (or a pending one overwritten).
`OwnershipTransferCompleted`	`new_owner: Address` — The new owner, who authorized the acceptance.	Emitted by `accept_admin` when a pending ownership transfer is accepted.

Errors

Error	Code	Description
`UtxoAlreadyExists`	`2000`	Raised by `transact` when a `utxo` named in `op.create` already has a `UTXO` record, spent or unspent, including a record spent by the same bundle (spends precede creates). Two raise sites, the pre-check in `process_bundle` and the write guard in `Store::create`, of which the pre-check fires first.
`UtxoDoesNotExist`	`2001`	Raised by `transact` when a `utxo` named in `op.spend` has no `UTXO` record. Two raise sites, the balance match in `process_bundle` and its re-check in `Store::spend`, of which the former fires first.
`UtxoAlreadySpent`	`2002`	Raised by `transact` when a `utxo` named in `op.spend` has a spent (`0`) `UTXO` record. Two raise sites, the balance match in `process_bundle` and its re-check in `Store::spend`, of which the former fires first.
`UnbalancedBundle`	`2003`	Raised by `transact` when the bundle does not balance: the sum of the spent amounts plus the total deposit does not equal the total created amount plus the total withdrawal (cf. `bundle_balanced`).
`InvalidCreateAmount`	`2004`	Raised by `transact` when an `amount` in `op.create` is not positive. Two raise sites, the check in `process_bundle` and its re-check in `Store::create`, of which the former fires first.
`RepeatedCreateUtxo`	`2005`	Raised by `transact` when the same `utxo` appears more than once in `op.create`.
`RepeatedSpendUtxo`	`2006`	Raised by `transact` when the same `utxo` appears more than once in `op.spend`.
`AuthContractNotSet`	`2008`	Declared by `transact` and `auth` for the case that `UtxoAuth` is not set. Never triggered: `UtxoAuth` is always set per `utxo_auth_always_set`.
`RepeatedAccountForDeposit`	`3000`	Raised by `transact` when the same `from` appears more than once in `op.deposit`.
`RepeatedAccountForWithdraw`	`3001`	Raised by `transact` when the same `to` appears more than once in `op.withdraw`.
`ConflictingConditionsForAccount`	`3002`	Raised by `transact` when an address appears in both `op.deposit` and `op.withdraw` with condition lists that are not identical as sequences (order and content, compared by canonical XDR).
`AmountOverflow`	`3003`	Raised by `transact` when summing the deposit amounts of `op`, summing its withdrawal amounts, or increasing `Supply` by a deposit exceeds the `i128` range.
`BundleHasConflictingConditions`	`3004`	Raised by `transact` when two conditions anywhere in `op` conflict: `Create` conditions naming one `utxo` with different amounts, `ExtDeposit` or `ExtWithdraw` conditions naming one address with different amounts, or `ExtIntegration` conditions whose `utxos` overlap across adapters or whose amounts or `utxos` differ under one adapter.
`AmountUnderflow`	`3005`	Declared by `transact` for the case that decreasing `Supply` by a withdrawal underflows the `i128` range. Never triggered: the running value of `Supply` in stage 4 never drops below `0`, so no subtraction can underflow. It enters the stage equal to the non-negative unspent UTXO total (`supply_equals_unspent_total`), the deposit loop only increases it, and every partial withdrawal total is at most the full withdrawal total, so the running value stays at or above the final value, itself the non-negative unspent total after the bundle.
`UnauthorizedOperation`	`3006`	Raised by `transact` when a `Create` or `ExtWithdraw` condition carried by an entry of `op.spend` or `op.deposit` is not executed by the bundle: no entry with identical key and amount in `op.create` or `op.withdraw` (cf. `signed_effects_executed`).
`InvalidExternalAmount`	`3007`	Raised by `transact` when an `amount` in `op.deposit` or `op.withdraw` is not positive.
`ReentrantCall`	`3008`	Declared by `transact` for the case that `ReentrancyGuard` is set on entry: an invocation re-entering one in progress. Never triggered on current hosts, which reject contract re-entry before the guard is reached (cf. the note at `ReentrancyGuard`). The guard makes `transact` non-reentrant independently of that host behavior.
`OwnerNotSet`	`2100`	Declared by every owner-gated function (`__constructor`, `set_admin`, `upgrade`) for the case that `Owner` is not set. Never triggered: in `__constructor` the owner is set immediately before being enforced, and elsewhere `Owner` is always set per `owner_always_set`.
`OwnerAlreadySet`	`2102`	Declared by `__constructor` for the case that `Owner` is already set. Never triggered: the constructor runs exactly once, at deployment, against fresh instance storage.
`NoPendingTransfer`	`2200`	Raised by `accept_admin` when no ownership transfer is pending: `PendingOwner` is not set, or the pending entry has expired. (Also declared on a cancel path in `set_admin` that is never taken.)
`InvalidLiveUntilLedger`	`2201`	Declared by `set_admin` for the case that the requested acceptance deadline is out of range. Never triggered: the deadline passed is always in range (cf. `assume_max_live_until_ledger`).
`InvalidPendingAccount`	`2202`	Declared on a cancel path in `set_admin` that is never taken. Never triggered.
`TransferExpired`	`2203`	Raised by `accept_admin` when the current ledger sequence exceeds the `live_until_ledger` of the pending transfer. (Reachable only if the entry TTL was extended past the deadline: an expired transfer normally reads as absent.)

Functions

`__constructor`

Categories	`constructor`
Authorization	`Owner` (the just-set `admin`)
Parameters	`e: Env` `admin: Address` — Address of the initial contract owner, which must authorize the deployment. `auth_contract: Address` — Address of the account to authorize UTXO spends and creates. `asset: Address` — Address of the token contract of the channel.
Returns	`()`
Reads	`Owner` — To check that it is not already set, and, once set, to enforce its authorization.
Mutates	`Owner` — Set to `admin`. `UtxoAuth` — Set to `auth_contract`. `Asset` — Set to `asset`.
Raises	`OwnerAlreadySet` — If `Owner` is already set. Never triggered: the constructor runs exactly once, at deployment, against fresh instance storage in which `Owner` is unset. `OwnerNotSet` — If `Owner` is not set when its authorization is enforced. Never triggered: `set_owner` runs immediately before `enforce_owner_auth` within the same call.
Emits	—

Initializes the contract: sets `admin` as the contract owner (`Owner`), `auth_contract` as the authorizer of UTXO spends and creates (`UtxoAuth`), and `asset` as the token of the channel (`Asset`). The owner role is enforced immediately after being set, so `admin` must authorize the deployment. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

`Supply` is left unset: the absent entry reads as `0` (cf. `supply`).

Postconditions

Name	Property	Proof
`owner_established`	On success, `Owner` is set to `admin`.	`ownable::set_owner` checks that `Owner` is unset, raising `OwnerAlreadySet` otherwise, then writes `admin` to the slot unconditionally. Nothing later in the function writes it: `enforce_owner_auth` only reads it.
`auth_established`	On success, `UtxoAuth` is set to `auth_contract`.	`set_auth` writes `auth_contract` to the slot unconditionally, and nothing else in the function touches it.
`asset_established`	On success, `Asset` is set to `asset`.	`write_asset_unchecked` writes `asset` to the slot unconditionally, and nothing else in the function touches it.
`supply_untouched`	The function does not write `Supply`: on a fresh deployment the entry remains unset.	No statement in the body touches the slot. The only writers in the contract are the treasury helpers invoked by `transact`.

`admin`

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

Returns the address of the current contract owner (admin).

Postconditions

Name	Property	Proof
`admin_return_value`	The result is `Owner` .	Direct read via `get_owner(e).unwrap()` , which cannot fail: the slot is set per `owner_always_set` . A violation would surface as a codeless `unwrap` trap, not as `OwnerNotSet` , which this function does not declare.

`set_admin`

Categories	`ownership`
Authorization	`Owner`
Parameters	`e: Env` `new_admin: Address` — Address of the proposed new owner.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner, and to record it in the emitted `OwnershipTransfer` .
Mutates	`PendingOwner` — Set to `PendingTransfer { address: new_admin, live_until_ledger: m }` , where `m` is `e.ledger().max_live_until_ledger()` at the time of the call, overwriting any previous value. The temporary entry TTL is extended to end at ledger `m` (if it would otherwise expire sooner).
Raises	`OwnerNotSet` — If `Owner` is not set. Never triggered: `Owner` is always set per `owner_always_set` . `NoPendingTransfer` — Only raised on the cancel path of `transfer_role` (deadline `0`), which is never taken (cf. the description). Never triggered. `InvalidPendingAccount` — Only raised on the cancel path of `transfer_role` (deadline `0`), which is never taken (cf. the description). Never triggered. `InvalidLiveUntilLedger` — If the requested deadline exceeds the maximum live-until ledger or is below the current ledger sequence. Never triggered: the deadline passed is exactly the maximum live-until ledger, which is at least the current sequence by `assume_max_live_until_ledger` .
Emits	`OwnershipTransfer` — On success: `OwnershipTransfer { old_owner: Owner, new_owner: new_admin, live_until_ledger: m }` , where `m` is `e.ledger().max_live_until_ledger()` at the time of the call.

Initiates a two-step ownership transfer: records `new_admin` as the pending owner, to take over once it accepts via `accept_admin`. Until then the current owner keeps its privileges.

The acceptance deadline is set to the maximum live-until ledger, the farthest a storage entry can live, so a pending transfer stays acceptable for the longest window the network allows (`pending_transfer_max_window`). Calling again overwrites any previous pending transfer. The cancel path of the underlying `transfer_role` helper (a deadline of `0`) is never taken, since the passed deadline is positive by `assume_max_live_until_ledger`. The contract therefore exposes no way to cancel a pending transfer, only to overwrite it.

Postconditions

Name	Property	Proof
`pending_transfer_established`	On success, `PendingOwner` holds `PendingTransfer { address: new_admin, live_until_ledger: m }`, where `m` is `e.ledger().max_live_until_ledger()` at the time of the call.	After `enforce_owner_auth` (a read-only authorization check), `transfer_role` runs with deadline `m`. By `assume_max_live_until_ledger` `m` is non-zero, so the cancel branch is skipped, and `m` passes the range check (`m` is trivially at most the maximum live-until ledger, and at least the current sequence by the same assumption). The helper then writes the `PendingTransfer` value unconditionally, overwriting any previous one.
`owner_unchanged`	On success, `Owner` is unchanged.	On this path `transfer_role` writes only `PendingOwner`, and `enforce_owner_auth` only reads `Owner`. Nothing else in the function touches storage.

`accept_admin`

Categories	`ownership`
Authorization	`PendingOwner`
Parameters	`e: Env`
Returns	`()`
Reads	`PendingOwner` — To verify a transfer is pending, check its deadline, authenticate the caller as the pending owner, set `Owner`, and record the new owner in the emitted `OwnershipTransferCompleted`.
Mutates	`Owner` — Set to `PendingOwner.address` at entry (the accepted new owner). `PendingOwner` — Unset on success.
Raises	`NoPendingTransfer` — If `PendingOwner` is not set: no transfer is pending, or the pending entry has expired (its TTL ends at its `live_until_ledger` deadline). `TransferExpired` — If the current ledger sequence exceeds `PendingOwner.live_until_ledger`. Since `set_admin` sets the entry TTL to end exactly at that deadline (the extension applies, rather than leaving the initial TTL of the fresh entry in place, because the deadline clears that initial TTL by `assume_max_live_until_ledger`), an expired transfer normally reads as absent (raising `NoPendingTransfer` instead). This error thus fires only if the entry TTL was extended past the deadline (which is possible because TTL extension is permissionless on Stellar).
Emits	`OwnershipTransferCompleted` — On success: `OwnershipTransferCompleted { new_owner: a }`, where `a = PendingOwner.address` at entry (equivalently, the value of `Owner` on return, per `ownership_transferred`).

Completes a pending two-step ownership transfer: the pending owner recorded by `set_admin` accepts, becoming the new owner (admin). There is no waiting period: the transfer may be accepted immediately after being requested (i.e. the recorded deadline is an expiry, not a time-lock). Clears the pending transfer state on success.

Postconditions

Name	Property	Proof
`ownership_transferred`	On success, `Owner` is set to `PendingOwner.address` at entry. In particular, `Owner` is written, never unset.	The `NoPendingTransfer` guard requires `PendingOwner` to be set. After the `TransferExpired` guard and the authorization of the pending `address`, `accept_transfer` removes `PendingOwner` and writes its `address` field to `Owner` (a `set`, never a removal), and nothing later in the function touches the slot.
`pending_transfer_cleared`	On success, `PendingOwner` is unset.	Removed unconditionally by `accept_transfer` before it writes `Owner` and returns.

`upgrade`

Categories	`upgrade`
Authorization	`Owner`
Parameters	`e: Env` `wasm_hash: BytesN<32>` — Hash identifying the installed Wasm blob to upgrade to.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	—
Raises	`OwnerNotSet` — If `Owner` is not set. Never triggered: `Owner` is always set per `owner_always_set`.
Emits	`Upgraded` — On success: `Upgraded { wasm_hash: wasm_hash }`.

Replaces the contract executable with the Wasm identified by `wasm_hash`.

The replacement is performed via the `stellar-contract-utils` helper `upgradeable::upgrade`, a thin wrapper around the host code-replacement call: it takes effect only after the current invocation completes, and the referenced Wasm must already be installed on the ledger, otherwise the host traps.

Postconditions

Name	Property	Proof
`wasm_replaced`	On success, the contract executable is replaced by the Wasm identified by `wasm_hash`, taking effect once the current invocation completes.	After the authorization check, `upgradeable::upgrade` unconditionally calls `update_current_contract_wasm` with `wasm_hash`. The deferred activation and the requirement that the Wasm blob be installed are host behavior, per the `soroban-sdk` documentation of `update_current_contract_wasm`.
`storage_unchanged`	The function writes no storage entry (the code replacement modifies the executable reference of the contract instance, not its storage).	The body is `enforce_owner_auth` (a read-only authorization check), an event publication, and the host code-replacement call. None writes a storage entry, and no TTL is extended.

`auth`

Categories	`config` , `view`
Authorization	—
Parameters	`e: Env`
Returns	`Address` — The authorizer account address.
Reads	`UtxoAuth` — To return its value.
Mutates	—
Raises	`AuthContractNotSet` — If `UtxoAuth` is not set. Never triggered: `UtxoAuth` is always set per `utxo_auth_always_set` .
Emits	—

Returns the address of the authorizer of UTXO spends and creates (`UtxoAuth`), to which `transact` delegates authorization.

Postconditions

Name	Property	Proof
`auth_return_value`	The result is `UtxoAuth` .	Direct read via the `auth` accessor of `UtxoHandlerTrait` , whose absent-entry branch (raising `AuthContractNotSet`) is unreachable per `utxo_auth_always_set` .

`utxo_balance`

Categories	`utxo`
Authorization	—
Parameters	`e: Env` `utxo: BytesN<65>` — The 65-byte SEC1-uncompressed public key of the UTXO to query.
Returns	`i128` — The amount of the UTXO if unspent (positive), `0` if spent, `-1` if no record exists.
Reads	`UTXO` — To return the value of `UTXO(sha256(utxo))` .
Mutates	`UTXO` — No entry value is written. When `UTXO(sha256(utxo))` exists, its TTL is extended to `Store_PERSISTENT_BUMP_AMOUNT` (if it would otherwise expire sooner than `Store_PERSISTENT_LIFETIME_THRESHOLD`).
Raises	—
Emits	—

Returns the spend state of `utxo` : a positive value is the amount of an unspent UTXO, `0` means the UTXO exists but has been spent, and `-1` means no record exists for the key.

Reading an existing record extends the TTL of its `UTXO` entry to `Store_PERSISTENT_BUMP_AMOUNT` (if it would otherwise expire sooner than `Store_PERSISTENT_LIFETIME_THRESHOLD`), so a holder keeps their UTXO alive simply by observing it. The instance TTL is not extended.

Postconditions

Name	Property	Proof
`utxo_balance_return_value`	The result is the value of `UTXO(sha256(utxo))` if that entry exists, and `-1` otherwise.	`Store::balance` reads the entry, returning its value when present and `-1` when absent.
`no_value_written`	The function writes no storage entry value: its only storage effect is the conditional TTL extension of the read entry.	The body is a single `Store::balance` call, whose only effects are the read and the conditional TTL extension.

`utxo_balances`

Categories	`utxo`
Authorization	—
Parameters	`e: Env` `utxos: Vec<BytesN<65>>` — The 65-byte SEC1-uncompressed public keys of the UTXOs to query.
Returns	`Vec<i128>` — The states of the queried UTXOs, in input order (cf. `utxo_balance`).
Reads	`UTXO` — To return the value of `UTXO(sha256(utxo))` for each `utxo` in `utxos`.
Mutates	`UTXO` — No entry value is written. The TTL of every existing entry read is extended to `Store_PERSISTENT_BUMP_AMOUNT` (if it would otherwise expire sooner than `Store_PERSISTENT_LIFETIME_THRESHOLD`).
Raises	—
Emits	—

Returns the spend states of `utxos` element-wise: for each `utxo` in `utxos`, in input order, the result holds `utxo_balance(utxo)`, with the same semantics (positive amount, `0` spent, `-1` no record) and the same side effect (the TTL of every existing `UTXO` entry read is extended).

Postconditions

Name	Property	Proof
`utxo_balances_return_value`	`result` has the length of `utxos`, and `result[i] = utxo_balance(utxos[i])` for every index `i`.	The body iterates `utxos` in order, appending the result of one `utxo_balance` call per element.
`no_value_written`	The function writes no storage entry value: its only storage effect is the conditional TTL extension of every read entry that exists.	Each iteration is a `Store::balance` call, cf. `no_value_written`.

`asset`

Categories	`config` , `view`
Authorization	—
Parameters	`e: Env`
Returns	`Address` — The token contract of the channel.
Reads	`Asset` — To return its value.
Mutates	—
Raises	—
Emits	—

Returns the address of the token contract of the channel (`Asset`).

Postconditions

Name	Property	Proof
`asset_return_value`	The result is `Asset` .	Direct read via `read_asset` , whose `unwrap` cannot fail: the slot is set per `asset_always_set` . A violation would surface as a codeless `unwrap` trap. The function declares no error.

`supply`

Categories	`utxo` , `view`
Authorization	—
Parameters	`e: Env`
Returns	`i128` — The value of `Supply` , `0` when the entry is unset.
Reads	`Supply` — To return its value, defaulting to `0` when unset.
Mutates	—
Raises	—
Emits	—

Returns the supply counter of the channel (`Supply`), or `0` while the entry is unset (cf. `supply_untouched`).

Postconditions

Name	Property	Proof
`supply_return_value`	The result is `Supply` if set, and `0` otherwise.	Direct read via `read_supply` , an `unwrap_or(0)` on the entry.

`transact`

<table>
<tr><td>Categories</td><td><code>utxo</code></td></tr>
<tr><td>Authorization</td><td>
<table>
<tr><th>Item</th><th>Authorizer</th><th>Description</th></tr>
<tr><td><code>∀(utxo, conditions) ∈ op.spend</code></td><td><code>UtxoAuth</code></td><td>The requirement <code>P256(utxo) -> conditions</code> enters the <code>AuthRequirements</code> over which the authorizer must approve the invocation via <code>require_auth_for_args</code> (or empty args when <code>op.spend</code> is empty), delegating to its <code>__check_auth</code> (<code>spend_authorization_delegated</code>).</td></tr>
<tr><td><code>∀(utxo, amount) ∈ op.create</code></td><td><code>UtxoAuth</code></td><td>Same authorization as <code>op.spend</code> : the one requirement gates the whole invocation (<code>spend_authorization_delegated</code>). Consent to the create travels as a <code>Create</code> condition through the signed-effects binding (<code>signed_effects_executed</code>).</td></tr>
<tr><td><code>∀(from, _, conditions) ∈ op.deposit</code></td><td><code>from</code></td><td>Must authorize via <code>require_auth_for_args</code> , bound to <code>conditions</code> and to nothing else of the bundle (<code>depositors_authorized</code>). The amount is covered only by the transfer authorization of the asset (<code>deposit_amount_not_channel_authorized</code>).</td></tr>
<tr><td><code>∀(to, amount, _) ∈ op.withdraw</code></td><td>None</td><td>No authorization is required of <code>to</code> by this contract. The outgoing transfer is pre-authorized by the contract itself via <code>authorize_as_current_contract</code> (<code>withdraw_conditions_inert</code>).</td></tr>
</table>
</td></tr>
<tr><td>Parameters</td><td>
<code>e: Env</code>

<code>op: ChannelOperation</code> —

The bundle to execute, a <code>ChannelOperation</code> of four lists:

• <code>spend: Vec<(BytesN<65>, Vec<Condition>)></code> : the UTXOs to spend, each <code>(utxo, conditions)</code> with <code>utxo</code> the 65-byte public key and <code>conditions</code> the attached condition list.
• <code>create: Vec<(BytesN<65>, i128)></code> : the UTXOs to create, each <code>(utxo, amount)</code> .
• <code>deposit: Vec<(Address, i128, Vec<Condition>)></code> : the deposits to pull in, each <code>(from, amount, conditions)</code> .
• <code>withdraw: Vec<(Address, i128, Vec<Condition>)></code> : the withdrawals to push out, each <code>(to, amount, conditions)</code> .

A <code>Condition</code> declares an effect of the bundle:

• <code>Create(utxo, amount)</code> : the creation of a UTXO.
• <code>ExtDeposit(address, amount)</code> : a deposit from an account.
• <code>ExtWithdraw(address, amount)</code> : a withdrawal to an account.
• <code>ExtIntegration(adapter, utxos, amount)</code> : a deposit of <code>amount</code> through the adapter contract at <code>adapter</code> , with <code>utxos</code> the keys that authorize its withdrawal.

The roles of the condition lists (authorization, signed-effects binding, conflict checks) are laid out in the description of <code>transact</code> .
</td></tr>
<tr><td>Returns</td><td><code>()</code> — Returns nothing on success. Two failure modes lie outside the raised errors listed here: a failed authorization (of <code>UtxoAuth</code> or of a depositor) surfaces as a host authorization error, and a failed token <code>transfer</code> propagates whatever the <code>Asset</code> contract raises.</td></tr>
<tr><td>Reads</td><td>
<code>ReentrancyGuard</code> — To reject a re-entering invocation.

<code>UtxoAuth</code> — To name the authorizer of the bundle in <code>require_auth_for_args</code> (stage 2).

<code>UTXO</code> — To check the spend state of every <code>utxo</code> named in <code>op.spend</code> and <code>op.create</code> (stage 3).

<code>Asset</code> — To identify the token contract for the transfers of stage 4.

<code>Supply</code> — To update it: each per-entry update of stage 4 is a checked read-modify-write.
</td></tr>
<tr><td>Mutates</td><td>
<code>ReentrancyGuard</code> — Set to <code>true</code> on entry and removed on exit: unset again on success (<code>guard_cleared</code>).

<code>UTXO</code> — The entry of every <code>(utxo, _)</code> in <code>op.spend</code> is overwritten with the <code>0</code> tombstone (<code>utxos_spent</code>), and one entry <code>UTXO(sha256(utxo)) -> amount</code> is added per <code>(utxo, amount)</code> in <code>op.create</code> (<code>utxos_created</code>). The TTL of every touched entry is extended to <code>Store_PERSISTENT_BUMP_AMOUNT</code> (if it would otherwise expire sooner than <code>Store_PERSISTENT_LIFETIME_THRESHOLD</code>).

<code>Supply</code> — Increased by <code>amount</code> per <code>(_, amount, _)</code> in <code>op.deposit</code> and decreased by <code>amount</code> per <code>(_, amount, _)</code> in <code>op.withdraw</code> , interleaved with the transfers of stage 4 (<code>supply_updated</code>).
</td></tr>
</table>

Raises	`ReentrantCall` — On entry: if `ReentrancyGuard` is set, the invocation re-entering an in-progress one. Never triggered on current hosts, which reject contract re-entry on their own (cf. the note at `ReentrancyGuard`). `BundleHasConflictingConditions` — Stage 1: if two conditions anywhere in `op` conflict. `InvalidExternalAmount` — Stage 1: if an `amount` in `op.deposit` or `op.withdraw` is not positive. `AmountOverflow` — Stage 1: if the total deposit or total withdrawal exceeds the `i128` range. Stage 4: if increasing `Supply` by a deposit does. `RepeatedAccountForDeposit` — Stage 1: if the same `from` appears more than once in `op.deposit`. `RepeatedAccountForWithdraw` — Stage 1: if the same `to` appears more than once in `op.withdraw`. `ConflictingConditionsForAccount` — Stage 1: if an address in both `op.deposit` and `op.withdraw` carries diverging condition lists. `UnauthorizedOperation` — Stage 1: if a signed `Create` or `ExtWithdraw` condition is not executed by the bundle (`signed_effects_executed`). `RepeatedSpendUtxo` — Stage 2: if the same `utxo` appears more than once in `op.spend`. `RepeatedCreateUtxo` — Stage 2: if the same `utxo` appears more than once in `op.create`. `AuthContractNotSet` — Stage 2: if `UtxoAuth` is not set. Never triggered: `UtxoAuth` is always set per `utxo_auth_always_set`. `UtxoDoesNotExist` — Stage 3: if a `utxo` in `op.spend` has no `UTXO` record. `UtxoAlreadySpent` — Stage 3: if a `utxo` in `op.spend` has a spent (`0`) record. `UtxoAlreadyExists` — Stage 3: if a `utxo` in `op.create` already has a record, spent or unspent. `InvalidCreateAmount` — Stage 3: if an `amount` in `op.create` is not positive. `UnbalancedBundle` — Stage 3: if, after all spends and creates, the bundle does not balance (`bundle_balanced`). `AmountUnderflow` — Stage 4: if decreasing `Supply` by a withdrawal underflows the `i128` range. Never triggered: the running value never drops below `0` (`supply_equals_unspent_total`, cf. `AmountUnderflow`).
Emits	—

Executes `op` as one atomic bundle that spends UTXOs, creates UTXOs, pulls deposits in, and pushes withdrawals out. Any failure aborts the whole invocation: a bundle executes in full or not at all. The bundle must balance: the spent amounts plus the total deposit equal the created amounts plus the total withdrawal (`bundle_balanced`).

The invocation proceeds in four stages, the first error aborting it.

Stage 1: Validation (`pre_process_channel_operation`). Rejects conflicting conditions anywhere in `op` (`BundleHasConflictingConditions`), non-positive external amounts (`InvalidExternalAmount`), overflowing deposit or withdrawal totals (`AmountOverflow`), repeated addresses within `op.deposit` or within `op.withdraw` (`RepeatedAccountForDeposit`, `RepeatedAccountForWithdraw`), diverging condition lists of an address listed in both (`ConflictingConditionsForAccount`), and violations of the signed-effects binding (defined below, `UnauthorizedOperation`).

Stage 2: Authorization (first part of `process_bundle`). Rejects duplicate keys within `op.spend` and within `op.create` (`RepeatedSpendUtxo`, `RepeatedCreateUtxo`), then requires the authorization of `UtxoAuth` over the derived requirements (`spend_authorization_delegated`).

Stage 3: UTXO state changes (second part of `process_bundle`). Every `(utxo, _)` in `op.spend` is tombstoned (`UtxoDoesNotExist`, `UtxoAlreadySpent`, `utxos_spent`), then every `(utxo, amount)` in `op.create` is written (`UtxoAlreadyExists`, `InvalidCreateAmount`, `utxos_created`). Spends precede

creates, so a key spent by the bundle cannot be re-created by it. The stage closes with the balance check (`UnbalancedBundle`).

Stage 4: External legs (`execute_external_operations`). For every `(from, amount, conditions)` in `op.deposit` , in order: `from` authorizes over `conditions` (`depositors_authorized`), `transfer(from, contract, amount)` is invoked on `Asset` (`contract` being the address of this contract), and `Supply` is increased by `amount` (`AmountOverflow`). Then, for every `(to, amount, _)` in `op.withdraw` : `transfer(contract, to, amount)` is invoked, pre-authorized by the contract itself, and `Supply` is decreased by `amount` (`AmountUnderflow`). What a successful `transfer` means is `assume_asset_token_compliant` (`tokens_transferred`). Withdrawal recipients are not validated: a `withdraw` entry naming the channel itself burns claims without moving tokens (`withdraw_to_channel_locks_value`).

The whole body runs under `ReentrancyGuard` , set on entry and removed on exit (`guard_cleared`): a re-entering call would fail with `ReentrantCall` (never triggered on current hosts, cf. the note at `ReentrancyGuard`). The instance TTL is bumped to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`). The invocation emits no events (`BundleEvent` and `UtxoEvent` are compiled out of the deployed build).

Condition roles. `spend` and `deposit` entries carry condition lists declaring the effects their authorizers consent to. The role of a condition depends on the list carrying it, in three respects.

Conditions travel with the authorization of their entry. Those of a `spend` entry are forwarded to `UtxoAuth` as the auth requirements of `P256(utxo)` (`spend_authorization_delegated`). Those of a `deposit` entry are the arguments over which `from` must authorize, and nothing else of the bundle enters that authorization (`depositors_authorized`). Those of a `withdraw` entry are unsigned metadata, excluded from every authorization (`withdraw_conditions_inert`).

The signed-effects binding ties consent to execution: every `Create` and `ExtWithdraw` condition carried by a `spend` or `deposit` entry must appear, with identical key and amount, in `op.create` or `op.withdraw` (`signed_effects_executed`). `ExtDeposit` and `ExtIntegration` conditions travel to the authorizer but are not enforced for execution by this contract. The binding and the balance are what let independently signed intents settle in one bundle: each signature consents only to its own effects, the binding forces those effects to be executed, and the balance confines anything unsigned (a fee, say) to the value the signed conditions leave unallocated. Byte-identical signed effects across entries merge into one execution, with the difference joining that unsigned value (`identical_signed_effects_merge`).

Conflict checking spans the whole bundle, `withdraw` -entry conditions included: no two conditions of `op` may conflict (`BundleHasConflictingConditions`), and an address listed in both `op.deposit` and `op.withdraw` must carry identical condition lists in both (`ConflictingConditionsForAccount`).

Condition roles in summary:

	`spend` entries	`deposit` entries	`withdraw` entries
Travel to an authorizer	Yes: to `UtxoAuth`, as the auth requirements of `P256(utxo)` (`spend_authorization_delegated`)	Yes: to `from`, which must authorize over them and over nothing else of the bundle (`depositors_authorized`)	Never: unsigned metadata, excluded from authorization (`withdraw_conditions_inert`)
Enter the signed-effects binding	Yes (`signed_effects_executed`)	Yes (`signed_effects_executed`)	No (`withdraw_conditions_inert`)
Checked for conflicts	Yes (`BundleHasConflictingConditions`)	Yes (`BundleHasConflictingConditions`, `ConflictingConditionsForAccount`)	Yes (`BundleHasConflictingConditions`, `ConflictingConditionsForAccount`)

Calls

`Asset`	`transfer(from: Address, to: Address, amount: i128)`	`transfer(from, contract, amount)` per `(from, amount, _)` in `op.deposit`, then `transfer(contract, to, amount)` per `(to, amount, _)` in `op.withdraw`, in list order (stage 4), with `contract` the address of this contract (`e.current_contract_address()`). These are the only direct contract invocations of the function.

Postconditions

Name	Property	Proof
`spend_authorization_delegated`	On success, `UtxoAuth` has authorized the invocation via `require_auth_for_args`, with argument vector `[req]` where `req` is the `AuthRequirements` map holding `P256(utxo) -> conditions` for every `(utxo, conditions)` in `op.spend`, or with argument vector `[]` when `op.spend` is empty.	`process_bundle` calls `require_auth_for_args` on the address held in `UtxoAuth` before any `UTXO` write. The argument vector holds exactly one value, the `AuthRequirements` produced by `calculate_auth_requirements`, which inserts one entry `SignerKey::P256(utxo) -> conditions` per element of `op.spend` and nothing else. The argument vector `[]` is passed instead exactly when that map is empty, and the map is empty exactly when `op.spend` is (one entry is inserted per spend element unconditionally). Success of the invocation implies the host found this authorization satisfied.
`utxos_spent`	On success, `UTXO(sha256(utxo)) = 0` for every `(utxo, _)` in `op.spend`.	`process_bundle` iterates every `(utxo, _)` in `op.spend` and calls `Store::spend`, which overwrites the entry with `0` after the `UtxoDoesNotExist` and `UtxoAlreadySpent` guards, so success implies every `spend` entry was tombstoned. Nothing later re-writes such an entry: the only subsequent `UTXO` writes are the creates of the same bundle, and `Store::create` refuses any key with an existing record (`UtxoAlreadyExists` guard), where a just-tombstoned key reads as `0`, not as absent. Stage 4 writes no `UTXO` entry.
`utxos_created`	On success, `UTXO(sha256(utxo)) = amount` for every `(utxo, amount)` in `op.create`.	After the spend loop, `process_bundle` iterates every `(utxo, amount)` in `op.create` and calls `Store::create`, which writes `amount` to the entry after the `UtxoAlreadyExists` and `InvalidCreateAmount` guards. No two creates touch one entry: the `RepeatedCreateUtxo` guard makes the keys distinct, and distinct 65-byte keys yield distinct entries (up to collisions of `sha256(utxo)`). Nothing after the create loop writes `UTXO`.
`bundle_balanced`	On success, the sum of the stored `UTXO` amounts of the entries spent via `op.spend`, plus the total amount of `op.deposit`, equals the total amount of `op.create` plus the total amount of `op.withdraw`.	`process_bundle` accumulates a running balance, the total deposit plus the spent amounts minus the created amounts, each spent amount being the stored `UTXO` value read immediately before its tombstoning write. After the loops it asserts equality of the running balance with the total withdrawal (`UnbalancedBundle` guard). The deposit and withdrawal totals are computed in stage 1 with `checked_add` (`AmountOverflow` guard). The accumulator itself uses plain `i128` arithmetic, which traps on overflow by `assume_overflow_checks`, so a wrapped sum cannot pass the equality check.
`supply_updated`	On success, `Supply` (reading unset as `0`) increases by the total amount of `op.deposit` and decreases by the total amount of `op.withdraw`.	Stage 4 performs one `increase_supply(amount)` per `(_, amount, _)` in `op.deposit` and one `decrease_supply(amount)` per `(_, amount, _)` in `op.withdraw`, each a read-modify-write of `Supply` through `read_supply` (an `unwrap_or(0)`, so an unset entry reads as `0`) and `write_supply_unchecked`, with the arithmetic checked (`AmountOverflow` and `AmountUnderflow` guards). No other statement of the invocation writes the slot, so on success the net change is the sum of the per-entry updates: plus the total deposit, minus the total withdrawal, both finite per the stage 1 `AmountOverflow` guards.
`depositors_authorized`	On success, for every `(from, _, conditions)` in `op.deposit`, `from` has authorized the invocation via `require_auth_for_args` with argument vector `[conditions]`.	The deposit loop of `execute_external_operations` calls `from.require_auth_for_args` with argument vector `[conditions]` for every `(from, _, conditions)` in `op.deposit`, before invoking the token transfer of the entry. The vector holds `conditions` and nothing else, so the authorization of a depositor binds only its own entry.
`tokens_transferred`	On success, for every entry `(from, amount, _)` of `op.deposit`, the contract received `amount` of `Asset` from `from`, and for every entry `(to, amount, _)` of `op.withdraw`, the contract sent `amount` of `Asset` to `to`.	For every `(from, amount, _)` in `op.deposit`, stage 4 invokes `transfer(from, contract, amount)` on `Asset` after the `require_auth_for_args` of `from`, and for every `(to, amount, _)` in `op.withdraw` it invokes `transfer(contract, to, amount)`, pre-authorized via `authorize_as_current_contract`. Success of `transact` implies every such invocation returned successfully. That a successful `transfer` moves exactly the named amount between the named balances is `assume_asset_token_compliant`: the contract source proves only the successful invocations.
`signed_effects_executed`	On success, every `Create` and `ExtWithdraw` condition carried in the conditions of `op.spend` or `op.deposit` is executed by the bundle: it appears, with identical key and amount, in `op.create` or `op.withdraw`.	`assert_signed_effects_are_executed` collects, keyed by canonical XDR bytes, every `Create` and `ExtWithdraw` condition of every `spend` and `deposit` entry into an authorized set, renders `op.create` as `Create` conditions and `op.withdraw` as `ExtWithdraw` conditions into an executed set, and asserts the authorized set is a subset of the executed set (`UnauthorizedOperation` guard). XDR equality pins variant, key or address, and amount, and the variant tag keeps a `Create` from matching a `withdraw` entry (and vice versa), so membership is exactly the stated property. The comparison has set semantics: duplicate condition bytes, within one list or across signers, are satisfied by a single executed occurrence, so byte-identical signed effects merge (one executed `ExtWithdraw` can discharge the identical condition of several signers, the residual value falling to the unsigned remainder, cf. `identical_signed_effects_merge`). `withdraw`-entry conditions are not collected: they are unsigned and grant nothing.

`withdraw_conditions_inert`	For every `(_, _, conditions)` in `op.withdraw`, `conditions` enters no authorization argument and no state change: it is read only by the conflict checks guarding `BundleHasConflictingConditions` and `ConflictingConditionsForAccount`.	The three authorization sites of the body draw their arguments from elsewhere: the `require_auth_for_args` on `UtxoAuth` passes requirements derived from `op.spend` alone (`spend_authorization_delegated`), the per-depositor `require_auth_for_args` passes `[conditions]` of the respective `deposit` entry (`depositors_authorized`), and the pre-authorization of each outgoing transfer (`authorize_as_current_contract`) contains the transfer invocation only, the withdrawal loop binding `(to, amount, _)` and discarding the condition list. The signed-effects check collects conditions from `op.spend` and `op.deposit` only. The state changes (the `UTXO` writes, the `Supply` updates, the transfers) take keys, addresses, and amounts, never condition lists. The remaining reads are the two conflict checks.
`guard_cleared`	On success, `ReentrancyGuard` is unset.	The last statement of the body, `exit_reentrancy_guard`, removes `ReentrancyGuard`, and success requires reaching it. No statement after `enter_reentrancy_guard` sets the entry again. On failure the temporary write rolls back with the transaction, so the guard never persists either way.

`ChannelAuthContract`

A Soroban custom account serving as the authorizer for Moonlight privacy channels. The contract holds no assets and stores no channel state: it maintains a provider registry, approves or rejects authorization requests, and records channel lifecycle changes as events. All administration is gated on a single owner role.

Providers and authorization. The core feature of the contract. Registered providers (`AuthorizedProvider`, managed by `add_provider` and `remove_provider`, queried by `is_provider`) hold signing authority. Whenever an operation names this contract as authorizer, the host invokes `__check_auth`, which approves only if two checks pass.

1. A provider check, gating every authorization on provider consent.
2. A UTXO check, letting `P256` key holders pre-authorize specific operations without knowing the enclosing transaction.

The mechanics of both checks (what each signature covers, how signatures expire, and the limits of those guarantees) are specified at `__check_auth`.

Ownership. A single administrative role, `Owner` (surfaced as the admin in the interface), set at deployment by `__constructor` and readable via `admin`. Transfers are two-step: `set_admin` records a pending transfer in `PendingOwner`, `accept_admin` completes it. The role is never vacant (`owner_always_set`).

Channel lifecycle. `enable_channel` and `disable_channel` record lifecycle changes of channels by emitting `ChannelStateChanged`, each record naming the channel and its asset. Event-only: the contract stores no channel or asset state. The lifecycle consequently has no effect on authorization: no channel state exists for `__check_auth` to consult, and events are not readable by contracts, so a channel whose latest recorded change is a disable authorizes under exactly the same checks as any other.

Upgrade. `upgrade` replaces the contract executable with owner authorization, preserving the contract address and all storage. Every property in this document describes the current executable: invariants and postconditions hold only up to an upgrade, since a replacement executable is free to change them.

Assumptions

Name	Property	Description
`assume_max_live_until_ledger`	Every read of `e.ledger().max_live_until_ledger()` returns a value that is strictly positive, at least the current ledger sequence (`e.ledger().sequence()`), and exceeds the current sequence by at least the initial TTL granted to a newly created temporary entry (the minimum temporary-entry TTL of the network).	The maximum live-until ledger is the highest ledger sequence to which the TTL of a storage entry may be extended, computed by the host as the current ledger sequence plus the maximum entry TTL of the network (`max_entry_ttl` , a network configuration parameter) minus one. On any live Stellar network the maximum entry TTL is strictly positive and the ledger sequence is strictly positive, so the returned value is at least the current sequence and never `0` . Moreover, the maximum entry TTL (millions of ledgers) exceeds the minimum temporary-entry TTL (tens of ledgers) by orders of magnitude, giving the final clause. That clause is what lets `set_admin` end the TTL of a freshly written `PendingOwner` entry exactly at the returned deadline: the extension applies only because the deadline lies beyond the initial TTL of the fresh entry (cf. `TransferExpired`). A violation could arise only in an artificial environment such as a misconfigured test ledger. This is a property of the runtime environment, not of the contract source.
`assume_check_auth_payload`	Every `payload` passed to `__check_auth` by the host is the SHA-256 of the `HashIdPreimageSorobanAuthorization` of the authorization entry, and the host checks the signature expiration ledger of the entry and consumes its nonce within the same authorization, immediately after the call returns successfully: if either step fails, the authorization as a whole fails.	The host authenticates a custom account by building the payload preimage from the network ID, the per-entry nonce, the signature expiration ledger, and the authorized invocation tree, then invokes `__check_auth` and, only on success, checks the expiration and consumes the nonce. A failure of either step fails the authorization just as a rejection would, so provider signatures are single-use and time-bounded independently of the contract-level expiry check. Since the contract-side checks run first, `__check_auth` itself may be invoked for an entry whose nonce is already consumed or whose expiration has passed: its approval only becomes an authorization once the host-side steps pass. (The nonce check contains an early return that skips both steps when the authorizing address is the transaction source account. That branch can never be taken for this contract, since a transaction source is always a Stellar account, never a contract address.) This is a property of the runtime environment, not of the contract source.
`assume_crypto_verify_traps`	Every call to `e.crypto().ed25519_verify()` and `e.crypto().secp256r1_verify()` either verifies the signature successfully or panics, trapping the transaction: neither returns control to the caller on an invalid signature.	The signature-verification host functions of soroban-sdk 25.3.x return `()` and panic on failure: they expose no success value to branch on. The `moonlight-auth` wrappers `verify_p256_signature` and `verify_ed25519_signature` therefore return `Ok(())` unconditionally and rely entirely on this panic-on-failure semantic, as their `MOON-04` safety comments document. The assumption is load-bearing for every verification claim in this document (cf. `provider_quorum` , `utxo_requirements_checked`): were a future SDK to return a result instead of panicking, `verify_signature` would silently accept invalid signatures. The `signature_verification_*` regression tests lock the current behavior and must be re-validated on every SDK upgrade. This is a property of the runtime environment, not of the contract source.

Invariants

Name	Property	Proof
`owner_always_set`	`Owner` is always set.	Established by `__constructor` , which always sets `Owner` . `accept_admin` overwrites it but never unsets it (cf. `ownership_transferred`). No other function writes the slot, and there is no remover for it: `renounce_ownership` , the only remover in the `stellar-access` library, is not exposed by this contract.
`provider_entries_unit`	Every `AuthorizedProvider` entry holds the unit value `()` .	Entries are written only by `add_provider` , which stores `()` (cf. `provider_registered`), and removed only by `remove_provider` . No other function writes them.

Constants

Name	Type	Value	Description
`DAY_IN_LEDGERS`	`u32`	`17_280`	Number of ledgers in approximately one day, at the nominal five-second ledger close time.
`INSTANCE_BUMP_AMOUNT`	`u32`	`7 * DAY_IN_LEDGERS`	TTL, in ledgers, to which the contract instance is extended by `__constructor` and `__check_auth`. Approximately seven days.
`INSTANCE_LIFETIME_THRESHOLD`	`u32`	`INSTANCE_BUMP_AMOUNT - DAY_IN_LEDGERS`	Remaining-TTL threshold, in ledgers, below which the instance TTL is extended to `INSTANCE_BUMP_AMOUNT`. Approximately six days.
`PROVIDER_THRESHOLD`	`u32`	`1`	Minimum number of valid provider signatures required by `__check_auth`. A function-local constant of `require_provider` in `moonlight-auth`, fixed at `1`: no storage or configuration affects it, so raising the threshold requires replacing the executable (`upgrade`).

Storage

Name	Type	Lifetime	Description
`AuthorizedProvider`	`Set<Address>`	`instance`	The set of registered providers, encoded as one instance entry `AuthorizedProvider(provider) -> ()` per registered `provider`, with membership as key existence. Providers are registered by `add_provider` and removed by `remove_provider`. Registered providers hold signing authority in `__check_auth`, which matches a `Provider` signer key against this set by deriving the account address from its Ed25519 public key. Registration is therefore only effective for the Ed25519 account address of a provider signing key: any `Address` can be registered, but no other kind can ever match (`provider_registration_ed25519_only`).
`Owner`	`Address`	`instance`	Address of the current contract owner, surfaced as the admin in the contract interface. The sole administrative role of the contract: it registers and removes providers (`add_provider`, `remove_provider`), records channel state changes (`enable_channel`, `disable_channel`), performs contract upgrades (`upgrade`), and initiates ownership transfers (`set_admin`). Set at construction by `__constructor` and thereafter only by `accept_admin` when a transfer completes.
`PendingOwner`	`PendingTransfer`	`temporary`	The pending ownership transfer, held as a `PendingTransfer { address, live_until_ledger }` pair: the proposed new owner and the ledger sequence through which the transfer may be accepted. Set (or overwritten) by `set_admin` and cleared when the transfer completes (`accept_admin`). Its sole privilege: `PendingOwner.address` may complete the transfer by calling `accept_admin`, becoming the new owner. A temporary entry whose TTL normally ends at its `live_until_ledger` deadline, so an expired pending transfer disappears on its own.

Events

Event	Fields	Description
`ContractInitialized`	`admin: Address` *(topic)* — The initial contract owner.	Emitted by `__constructor` on deployment, recording the initial contract owner.
`ProviderAdded`	`provider: Address` *(topic)* — The registered provider.	Emitted by `add_provider` when a provider is registered.
`ProviderRemoved`	`provider: Address` *(topic)* — The deregistered provider.	Emitted by `remove_provider` when a provider is deregistered.
`ChannelStateChanged`	`channel: Address` *(topic)* — Address recorded as the channel whose state changed. `asset: Address` *(topic)* — Address recorded as the asset of the channel. `enabled: bool` — `true` when the channel is enabled or re-enabled (`enable_channel`), `false` when disabled (`disable_channel`).	Emitted by `enable_channel` to record that a channel was enabled or re-enabled, and by `disable_channel` to record it was disabled. The contract holds no channel or asset state: this event is the only on-chain artifact of the channel lifecycle.
`Upgraded`	`wasm_hash: BytesN<32>` *(topic)* — Hash of the installed Wasm blob the contract was upgraded to.	Emitted by `upgrade` to record the hash of the Wasm the contract was upgraded to.
`OwnershipTransfer`	`old_owner: Address` — The current owner, who initiated the transfer and authorized the call. `new_owner: Address` — The proposed new owner. `live_until_ledger: u32` — The ledger sequence through which the transfer may be accepted.	Emitted by `set_admin` when an ownership transfer is initiated (or a pending one overwritten).
`OwnershipTransferCompleted`	`new_owner: Address` — The new owner, who authorized the acceptance.	Emitted by `accept_admin` when a pending ownership transfer is accepted.

Errors

Error	Code	Description
`BadArg`	`1000`	Raised by `__check_auth` when the first argument of a contract context does not decode as `AuthRequirements`.
`MissingSignature`	`1002`	Raised by `__check_auth` when the `AuthRequirements` of a context specify a `P256` signer for which no signature was submitted.
`InvalidSignatureFormat`	`1004`	Raised by `__check_auth` when a signature variant does not match its signer key.
`NoConditions`	`1008`	Raised by `__check_auth` when a signer in the `AuthRequirements` of a context has an empty condition list.
`UnexpectedContext`	`1009`	Raised by `__check_auth` when an authorization context is not a contract context.
`SignatureExpired`	`1010`	Raised by `__check_auth` when a checked signature has its `valid_until_ledger` below the current ledger sequence.
`ProviderThresholdNotMet`	`1011`	Raised by `__check_auth` when fewer than `PROVIDER_THRESHOLD` `Provider`-keyed signature entries were submitted, each submitted one having passed its checks.
`ProviderAlreadyRegistered`	`1012`	Raised by `add_provider` when the provider is already registered.
`ProviderNotRegistered`	`1013`	Raised by `remove_provider` when the provider is not registered, and by `__check_auth` when a submitted provider signature does not belong to a registered provider.
`OwnerNotSet`	`2100`	Declared by every owner-gated function (`set_admin`, `upgrade`, `add_provider`, `remove_provider`, `enable_channel`, `disable_channel`) for the case that `Owner` is not set. Never triggered: `Owner` is always set per `owner_always_set`.
`OwnerAlreadySet`	`2102`	Declared by `__constructor` for the case that `Owner` is already set. Never triggered: the constructor runs exactly once, at deployment, against fresh instance storage.
`NoPendingTransfer`	`2200`	Raised by `accept_admin` when no ownership transfer is pending: `PendingOwner` is not set, or the pending entry has expired. (Also declared on a cancel path in `set_admin` that is never taken.)
`InvalidLiveUntilLedger`	`2201`	Declared by `set_admin` for the case that the requested acceptance deadline is out of range. Never triggered: the deadline passed is always in range (cf. `assume_max_live_until_ledger`).
`InvalidPendingAccount`	`2202`	Declared on a cancel path in `set_admin` that is never taken. Never triggered.
`TransferExpired`	`2203`	Raised by `accept_admin` when the current ledger sequence exceeds the `live_until_ledger` of the pending transfer. (Reachable only if the entry TTL was extended past the deadline: an expired transfer normally reads as absent.)

Functions

`__constructor`

Categories	`constructor`
Authorization	—
Parameters	`env: Env` `admin: Address` — Address of the initial contract owner.
Returns	`()`
Reads	`Owner` — To check that it is not already set.
Mutates	`Owner` — Set to `admin`.
Raises	`OwnerAlreadySet` — If `Owner` is already set. Never triggered: the constructor runs exactly once, at deployment, against fresh instance storage in which `Owner` is unset.
Emits	`ContractInitialized` — On success: `ContractInitialized { admin: admin }`.

Initializes the contract, setting `admin` as the contract owner. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`).

Postconditions

Name	Property	Proof
`owner_established`	On success, `Owner` is set to `admin`.	`ownable::set_owner` checks that `Owner` is unset, raising `OwnerAlreadySet` otherwise, then writes `admin` to the slot unconditionally. Nothing later in the function touches it.

`admin`

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

Returns the address of the current contract owner (admin).

Postconditions

Name	Property	Proof
`admin_return_value`	The result is `Owner`.	Direct read via `get_owner(e).unwrap()`, which cannot fail: the slot is set per `owner_always_set`. A violation would surface as a codeless `unwrap` trap, not as `OwnerNotSet`, which this function does not declare.

`set_admin`

Categories	`ownership`
Authorization	`Owner`
Parameters	`e: Env` `new_admin: Address` — Address of the proposed new owner.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner, and to record it in the emitted `OwnershipTransfer`.
Mutates	`PendingOwner` — Set to `PendingTransfer { address: new_admin, live_until_ledger: m }`, where `m` is `e.ledger().max_live_until_ledger()` at the time of the call, overwriting any previous value. The temporary entry TTL is extended to end at ledger `m` (if it would otherwise expire sooner).
Raises	`OwnerNotSet` — If `Owner` is not set. Never triggered: `Owner` is always set per `owner_always_set`. `NoPendingTransfer` — Only raised on the cancel path of `transfer_role` (deadline `0`), which is never taken (cf. the description). Never triggered. `InvalidPendingAccount` — Only raised on the cancel path of `transfer_role` (deadline `0`), which is never taken (cf. the description). Never triggered. `InvalidLiveUntilLedger` — If the requested deadline exceeds the maximum live-until ledger or is below the current ledger sequence. Never triggered: the deadline passed is exactly the maximum live-until ledger, which is at least the current sequence by `assume_max_live_until_ledger`.
Emits	`OwnershipTransfer` — On success: `OwnershipTransfer { old_owner: Owner, new_owner: new_admin, live_until_ledger: m }`, where `m` is `e.ledger().max_live_until_ledger()` at the time of the call.

Initiates a two-step ownership transfer: records `new_admin` as the pending owner, to take over once it accepts via `accept_admin`. Until then the current owner keeps its privileges.

The acceptance deadline is set to the maximum live-until ledger, the farthest a storage entry can live, so a pending transfer stays acceptable for the longest window the network allows (`pending_transfer_max_window`). Calling again overwrites any previous pending transfer. The cancel path of the underlying `transfer_role` helper (a deadline of `0`) is never taken, since the passed deadline is positive by `assume_max_live_until_ledger`. The contract therefore exposes no way to cancel a pending transfer, only to overwrite it.

Postconditions

Name	Property	Proof
`pending_transfer_established`	On success, `PendingOwner` holds `PendingTransfer { address: new_admin, live_until_ledger: m }`, where `m` is `e.ledger().max_live_until_ledger()` at the time of the call.	After `enforce_owner_auth` (a read-only authorization check), `transfer_role` runs with deadline `m`. By `assume_max_live_until_ledger` `m` is non-zero, so the cancel branch is skipped, and `m` passes the range check (`m` is trivially at most the maximum live-until ledger, and at least the current sequence by the same assumption). The helper then writes the `PendingTransfer` value unconditionally, overwriting any previous one.
`owner_unchanged`	On success, `Owner` is unchanged.	On this path `transfer_role` writes only `PendingOwner`, and `enforce_owner_auth` only reads `Owner`. Nothing else in the function touches storage.

`accept_admin`

Categories	`ownership`
Authorization	`PendingOwner`
Parameters	`e: Env`
Returns	`()`
Reads	`PendingOwner` — To verify a transfer is pending, check its deadline, authenticate the caller as the pending owner, set `Owner`, and record the new owner in the emitted `OwnershipTransferCompleted`.
Mutates	`Owner` — Set to `PendingOwner.address` at entry (the accepted new owner). `PendingOwner` — Unset on success.
Raises	`NoPendingTransfer` — If `PendingOwner` is not set: no transfer is pending, or the pending entry has expired (its TTL ends at its `live_until_ledger` deadline). `TransferExpired` — If the current ledger sequence exceeds `PendingOwner.live_until_ledger`. Since `set_admin` sets the entry TTL to end exactly at that deadline (the extension applies, rather than leaving the initial TTL of the fresh entry in place, because the deadline clears that initial TTL by `assume_max_live_until_ledger`), an expired transfer normally reads as absent (raising `NoPendingTransfer` instead). This error thus fires only if the entry TTL was extended past the deadline (which is possible because TTL extension is permissionless on Stellar).
Emits	`OwnershipTransferCompleted` — On success: `OwnershipTransferCompleted { new_owner: a }`, where `a = PendingOwner.address` at entry (equivalently, the value of `Owner` on return, per `ownership_transferred`).

Completes a pending two-step ownership transfer: the pending owner recorded by `set_admin` accepts, becoming the new owner (admin). There is no waiting period: the transfer may be accepted immediately after being requested (i.e. the recorded deadline is an expiry, not a time-lock). Clears the pending transfer state on success.

Postconditions

Name	Property	Proof
`ownership_transferred`	On success, `Owner` is set to `PendingOwner.address` at entry. In particular, `Owner` is written, never unset.	The `NoPendingTransfer` guard requires `PendingOwner` to be set. After the `TransferExpired` guard and the authorization of the pending `address`, `accept_transfer` removes `PendingOwner` and writes its `address` field to `Owner` (a `set`, never a removal), and nothing later in the function touches the slot.
`pending_transfer_cleared`	On success, `PendingOwner` is unset.	Removed unconditionally by `accept_transfer` before it writes `Owner` and returns.

`upgrade`

Categories	`upgrade`
Authorization	`Owner`
Parameters	`e: Env` `wasm_hash: BytesN<32>` — Hash identifying the installed Wasm blob to upgrade to.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	—
Raises	`OwnerNotSet` — If `Owner` is not set. Never triggered: `Owner` is always set per `owner_always_set`.
Emits	`Upgraded` — On success: `Upgraded { wasm_hash: wasm_hash }`.

Replaces the contract executable with the Wasm identified by `wasm_hash`.

The replacement is performed via the `stellar-contract-utils` helper `upgradeable::upgrade`, a thin wrapper around the host code-replacement call: it takes effect only after the current invocation completes, and the referenced Wasm must already be installed on the ledger, otherwise the host traps.

Postconditions

Name	Property	Proof
`wasm_replaced`	On success, the contract executable is replaced by the Wasm identified by `wasm_hash`, taking effect once the current invocation completes.	After the authorization check, `upgradeable::upgrade` unconditionally calls `update_current_contract_wasm` with `wasm_hash`. The deferred activation and the requirement that the Wasm blob be installed are host behavior, per the `soroban-sdk` documentation of `update_current_contract_wasm`.
`storage_unchanged`	The function writes no storage entry (the code replacement modifies the executable reference of the contract instance, not its storage).	The body is `enforce_owner_auth` (a read-only authorization check), an event publication, and the host code-replacement call. None writes a storage entry, and no TTL is extended.

`is_provider`

Categories	`provider` , `view`
Authorization	—
Parameters	`e: Env` `provider: Address` — Address to check.
Returns	`bool` — `true` if `provider` is a registered provider, `false` otherwise.
Reads	`AuthorizedProvider` — To check whether an entry for `provider` exists.
Mutates	—
Raises	—
Emits	—

Returns whether `provider` is currently a registered provider.

Postconditions

Name	Property	Proof
`is_provider_return_value`	The result is `true` if and only if the entry `AuthorizedProvider(provider)` exists.	Direct existence check on `AuthorizedProvider(provider)` . The typed read (`get::<_, ()>`) cannot trap, as every stored entry holds `()` per `provider_entries_unit` .

`add_provider`

Categories	`provider`
Authorization	`Owner`
Parameters	`e: Env` `provider: Address` — Address of the provider to register.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `AuthorizedProvider` — To check that `provider` is not already registered.
Mutates	`AuthorizedProvider` — The entry `AuthorizedProvider(provider) -> ()` is added.
Raises	`OwnerNotSet` — If `Owner` is not set. Never triggered: `Owner` is always set per `owner_always_set` . `ProviderAlreadyRegistered` — If the entry `AuthorizedProvider(provider)` already exists.
Emits	`ProviderAdded` — On success: `ProviderAdded { provider: provider }` .

Registers `provider` as a provider, adding it to `AuthorizedProvider` . Fails if it is already registered. The address is not validated: registration is effective only for the Ed25519 account address of a provider signing key (`provider_registration_ed25519_only`).

Postconditions

Name	Property	Proof
`provider_registered`	On success, the storage holds an entry `AuthorizedProvider(provider) -> ()`.	After the `ProviderAlreadyRegistered` guard, `register_provider` unconditionally writes `AuthorizedProvider(provider) -> ()`, and nothing later in the function touches the entry.

`remove_provider`

Categories	`provider`
Authorization	`Owner`
Parameters	`e: Env` `provider: Address` — Address of the provider to deregister.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner. `AuthorizedProvider` — To check that `provider` is registered.
Mutates	`AuthorizedProvider` — The entry `AuthorizedProvider(provider)` is removed.
Raises	`OwnerNotSet` — If `Owner` is not set. Never triggered: `Owner` is always set per `owner_always_set`. `ProviderNotRegistered` — If the entry `AuthorizedProvider(provider)` does not exist.
Emits	`ProviderRemoved` — On success: `ProviderRemoved { provider: provider }`.

Deregisters `provider`, removing it from `AuthorizedProvider`. Fails if it is not registered.

Postconditions

Name	Property	Proof
`provider_deregistered`	On success, the entry `AuthorizedProvider(provider)` does not exist.	After the `ProviderNotRegistered` guard, `deregister_provider` unconditionally removes `AuthorizedProvider(provider)`, and nothing later in the function touches the entry.

`enable_channel`

Categories	`channel`
Authorization	`Owner`
Parameters	`e: Env` `channel: Address` — Address recorded as the channel being enabled. `asset: Address` — Address recorded as the asset of the channel.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	—
Raises	`OwnerNotSet` — If `Owner` is not set. Never triggered: `Owner` is always set per `owner_always_set`.
Emits	`ChannelStateChanged` — On success: `ChannelStateChanged { channel: channel, asset: asset, enabled: true }`.

Records a channel as enabled by emitting `ChannelStateChanged` with `enabled` set to `true`. Also used to re-enable a previously disabled channel, the two cases are not distinguished.

Event-only: the contract stores no channel or asset state, and performs no validation of `channel` or `asset`.

Postconditions

Name	Property	Proof
`storage_unchanged`	The function writes no storage entry.	The body is `enforce_owner_auth` (a read-only authorization check) followed by an event publication. Neither writes any storage entry, and no TTL is extended.

`disable_channel`

Categories	`channel`
Authorization	`Owner`
Parameters	`e: Env` `channel: Address` — Address recorded as the channel being disabled. `asset: Address` — Address recorded as the asset of the channel.
Returns	`()`
Reads	`Owner` — To authenticate the caller as the current owner.
Mutates	—
Raises	`OwnerNotSet` — If `Owner` is not set. Never triggered: `Owner` is always set per `owner_always_set`.
Emits	`ChannelStateChanged` — On success: `ChannelStateChanged { channel: channel, asset: asset, enabled: false }`.

Records a channel as disabled by emitting `ChannelStateChanged` with `enabled` set to `false`. A disabled channel can be re-enabled via `enable_channel`.

Event-only: the contract stores no channel or asset state, and performs no validation of `channel` or `asset`. The change has no effect on `__check_auth`: no channel state exists to consult.

Postconditions

Name	Property	Proof
`storage_unchanged`	The function writes no storage entry.	The body is `enforce_owner_auth` (a read-only authorization check) followed by an event publication. Neither writes any storage entry, and no TTL is extended.

`__check_auth`

Categories	`auth`
Authorization	—
Parameters	`e: Env` `payload: Hash<32>` — Host-computed hash of the authorization payload. This is what providers sign. The SHA-256 of the `HashIdPreimageSorobanAuthorization` of the authorization entry, committing to: • `network_id` : the network identifier, • `nonce` : the nonce of the entry, consumed by the host immediately after this function returns successfully (a failed consumption fails the authorization), • `signature_expiration_ledger` : the expiration of the entry, checked by the host at the same site, • `invocation` : the authorized invocation tree. That the host builds the hash this way and enforces the committed fields is `assume_check_auth_payload` . `signatures: Signatures` — The submitted signatures, supplied by the transaction submitter in the authorization entry. A wrapper around `Map<SignerKey, (Signature, u32)>` (`Signatures` in `moonlight-primitives`), holding at most one entry per signer key: • each key is a `SignerKey` , identifying a signer: a `P256` SEC1-uncompressed public key, a plain `Ed25519` public key, or the `Provider` Ed25519 public key of a provider account, • each value pairs a `Signature` with its `valid_until_ledger` , the ledger sequence through which it remains valid. `contexts: Vec<Context>` — The authorization contexts being approved, one per authorization requirement attributed to this account. A `Context` is either a contract invocation (`Contract` : the called contract, the function name, and the call arguments) or one of two contract-creation host functions. Only contract invocations pass the check.
Returns	`Result<(), MoonlightError>` — `Ok(())` if the authorization is approved. For the failure cases, see the raised errors: some are returned as `Err` , some raised by panic, but the host escalates both to the same failed call carrying the error code, so they are not treated separately here. (One failure mode lies outside the error type: an invalid signature of the correct variant fails in the crypto host function without a `MoonlightError` code, cf. `assume_crypto_verify_traps` .)
Reads	`AuthorizedProvider` — To check that each `Provider` -keyed entry of `signatures` belongs to a registered provider.
Mutates	—
Raises	`ProviderNotRegistered` — If a `Provider` -keyed entry of `signatures` does not belong to a registered provider. `InvalidSignatureFormat` — If a checked signature variant does not match its signer key: the provider check requires an Ed25519 signature for each `Provider` key it verifies, and the UTXO check a P256 signature for each `P256` key. (The underlying `verify_signature` also accepts `Ed25519` keys with Ed25519 signatures, but no path in this contract checks an `Ed25519` -keyed entry. The `Secp256k1` and `BLS12_381` signature variants match no key kind and always raise this error when checked.) `SignatureExpired` — If a checked signature (`Provider` -keyed in the provider check, `P256` -keyed in the UTXO check) has its `valid_until_ledger` below the current ledger sequence. (For `Provider` -keyed entries the deadline is submitter-supplied and not covered by the signature, cf. `provider_expiry_submitter_supplied` .) `ProviderThresholdNotMet` — If fewer than `PROVIDER_THRESHOLD` `Provider` -keyed entries are present in `signatures` . Reached only when every present entry passed its checks: a failing entry raises its own error instead. While no providers are registered, authorization thus fails unavoidably, with this error if no `Provider` -keyed entry was submitted and with `ProviderNotRegistered` otherwise. `UnexpectedContext` — If an entry of `contexts` is not a contract context. `BadArg` — If the first argument of a contract context does not decode as `AuthRequirements` . (A second, defensive raise site, a bounds-checked argument fetch, is never triggered.) `NoConditions` — If a signer in the `AuthRequirements` of a context has an empty condition list. `MissingSignature` — If the `AuthRequirements` of a context specify a `P256` signer for which `signatures` holds no entry. (A second raise site in the provider check is never triggered: the looked-up key is drawn from the map being looked up.)
Emits	—

The custom account entry point: the host invokes it whenever an operation names this contract as authorizer (via `require_auth` or `require_auth_for_args`). The single exception, invoker authorization for a contract calling directly on its own behalf, cannot arise here: the contract invokes no other contracts. The host is also the only possible caller: contract functions whose name starts with `__` are reserved and their direct invocation rejected, so neither the checks nor the TTL extension below can be triggered outside an authorization flow. The authorization is approved if the call returns successfully, and rejected on any error or trap. Bumps the instance TTL to `INSTANCE_BUMP_AMOUNT` (if it would otherwise expire sooner than `INSTANCE_LIFETIME_THRESHOLD`). Like every effect of the call, the extension persists only if the enclosing transaction succeeds. In particular it persists only on approval, though approval alone does not suffice: a transaction may still fail after this call, rolling the extension back.

The check has two phases, run in order: the first error rejects the authorization.

1. Provider check (`require_provider` in `moonlight-auth`). Every `Provider`-keyed entry of `signatures` is checked and must pass, so one failing entry fails the whole check, and at least `PROVIDER_THRESHOLD` such entries are required. Each entry must satisfy:

- *membership*: the Stellar account address derived from the Ed25519 public key of the signer key is registered in `AuthorizedProvider` (cf. `provider_registration_ed25519_only`),
- *expiry*: the `valid_until_ledger` of the entry is at least the current ledger sequence,
- *verification*: the signature is an Ed25519 signature verifying over `payload`.

2. UTXO check (`handle_utxo_auth` in `moonlight-auth`). Every entry of `contexts` must be a contract context, and contexts without arguments are skipped: an argument-less invocation of any contract is gated by the provider check alone (`argless_contexts_provider_gated`). Each context with arguments must satisfy:

- *decoding*: the first argument decodes as `AuthRequirements`, a map from signer keys to condition lists,
- *conditions*: every signer of the map has a non-empty condition list,
- *verification*: every `P256` signer of the map has an entry in `signatures` whose `valid_until_ledger` is at least the current ledger sequence and whose P256 signature verifies over the message reconstructed by `hash_payload`, the hash of a byte encoding of the signer's condition list, the `valid_until_ledger` of the entry, and the address of the requesting contract. The message binds that address but not the function name or the remaining arguments of the context. Nor does it bind the order of the condition list: `hash_payload` buckets the conditions by variant (`Create`, then `ExtDeposit`, `ExtWithdraw`, `ExtIntegration`), preserving order only within each bucket, so any context of that contract whose condition list is a variant-stable reordering of the signed one (identical per-variant subsequences) is covered by the same signature. And it binds even those subsequences only up to the collisions of the encoding (next paragraph). Signers of other kinds are skipped. This is the only consultation of `signatures` in this phase.

The `hash_payload` encoding is not injective: distinct condition lists can produce the same signed message. The byte layout and the collision cases are recorded in `hash_payload_not_injective`.

Because `signatures` holds at most one entry per signer key, a `P256` signer named in several contexts can pass only if every occurrence reconstructs the identical message, with the single entry supplying one shared `valid_until_ledger`. Two contexts naming the same `P256` signer under different contracts can therefore never both be approved in one call, since the contract address opens the encoding. Two under the same contract can, but only if their condition lists encode identically: variant-stable reorderings of each other, or a collision as above.

Two edge cases pass this phase vacuously: an empty `contexts` vector, which reduces the whole check to the provider phase, and a decoded requirements map with no signers, which imposes nothing.

The two signature kinds carry asymmetric guarantees:

	Phase 1: `Provider`	Phase 2: `P256`
Message signed	`payload`	the `hash_payload` message over the per-variant subsequences of the signer's condition list and the address of the requesting contract, both from the `contexts` entry, and the `valid_until_ledger` of its `signatures` entry (a non-injective encoding, cf. `hash_payload_not_injective`)
Deadline covered by the signature?	No: submitter-supplied metadata, the check constrains only cooperative submitters (`provider_expiry_submitter_supplied`)	Yes: bound into the signed message
Effective expiry	the signature expiration ledger committed to by `payload`, enforced by the host	the signed `valid_until_ledger` of its `signatures` entry, enforced by this phase
Replay protection	the per-entry nonce committed to by `payload`, consumed by the host	none in this check: an unexpired `signatures` entry can be reused across authorizations. Single-use must come from the requesting contract (`p256_signatures_reusable_in_check`)

All other entries of `signatures` are ignored entirely: `Ed25519`-keyed entries (consulted by neither phase) and `P256`-keyed entries named in no context are neither verified nor expiry-checked, and their presence causes no error. For entries that are checked, an invalid signature of the correct variant traps the transaction at the host rather than returning an error (`assume_crypto_verify_traps`): `InvalidSignatureFormat` indicates a mismatched signer-key and signature variant, not a failed verification.

Postconditions

Name	Property	Proof
`provider_quorum`	On success, at least `PROVIDER_THRESHOLD` distinct registered providers have signed `payload` with Ed25519 signatures that verified and were unexpired per the `valid_until_ledger` of their `signatures` entries.	`require_provider` increments `provider_quorum` only after the signer passed the `ProviderNotRegistered` guard, the `SignatureExpired` check, and `verify_signature` (which traps on an invalid signature, `assume_crypto_verify_traps`). Map keys are distinct, and the address derivation embeds the public key bytes into the address, so distinct `Provider` keys yield distinct provider addresses: the count ranges over distinct providers. The `ProviderThresholdNotMet` guard enforces the threshold.
`utxo_requirements_checked`	On success, all of the following hold: • every entry of `contexts` is a contract context, • in every context carrying arguments, the first argument decodes as `AuthRequirements`, • every signer of every decoded `AuthRequirements` map has a non-empty condition list, • every `P256` signer of every decoded `AuthRequirements` map has an entry in `signatures`, • every such entry is unexpired: its `valid_until_ledger` is at least the current ledger sequence, • every such entry carries a P256 signature that verified against the `hash_payload` message over the signer's condition list, the `valid_until_ledger` of the entry, and the requesting contract (a binding exact only up to the encoding collisions recorded in `hash_payload_not_injective`).	`handle_utxo_auth` iterates all contexts in order, returning `UnexpectedContext` upon reaching a non-contract one (so success implies none was present) and skipping only argument-less ones (the skip continues to the next context, it never short-circuits the loop). Within a context, the decoding is guarded by `BadArg`, every signer passes the `NoConditions` guard, and every `P256` signer additionally passes the `MissingSignature` and `SignatureExpired` guards and `verify_signature` over the `hash_payload` message (trap on an invalid signature, `assume_crypto_verify_traps`), before the loop advances. That the message depends on the condition list only through its per-variant subsequences follows from the bucketing in `hash_payload`, which groups conditions by variant before hashing. That it pins them down only up to encoding collisions is recorded in `hash_payload_not_injective`. (The per-signer condition lookup is an `unwrap` on a key drawn from the map being iterated, therefore it cannot fail.)
`storage_unchanged`	The function writes no storage entry (it extends the instance TTL, changing no entry value).	The body is a TTL extension and the two read-only check phases. No path performs a storage write.