CIP-56 vs. ERC-20: A Comparative Security Model Analysis

Abstract

A fungible token is never just its interface. It reflects the ledger it lives on, and the chain underneath decides how state is stored, who gets to see it, and what "authorization" even means. This article compares two token standards that look alike on the surface but rest on opposite foundations: Ethereum's ERC-20, built on global, public, permissionless state, and Canton's CIP-56, built on private, multi-party, registry-mediated UTXO holdings. We walk through the CIP-56 model and its OpenZeppelin reference implementation, follow how value moves and settles, and look at why a whole family of ERC-20 attack classes (allowance abuse, reentrancy, missing return values, unchecked transfers) has nothing to attack on a private UTXO ledger. Then we turn the analysis around and tally the cost: the registry integrity, off-ledger infrastructure, and application-level invariants that CIP-56 trades away for the attack surface it removes.

A note on versioning: CIP-0056 vs. CIP-0112

CIP-0056 is the first version (V1) of the Canton Network Token Standard. It carries the status Final (approved March 2025) and is the standard this article focuses on throughout.

Since then, **CIP-0112** — the Canton Network Token Standard V2 — was approved more recently (June 2026). Its status is Approved rather than Final, and it is an improved, backward-compatible evolution of CIP-0056. In the words of its abstract: "This proposal outlines a set of improvements to the CIP-0056 token standard. These enhancements make the token standard more suitable for integration with trading and settlement venues across the TradFi and DeFi spectrum, as well as for on-chain securities that model holding and settlement via custody chains. They also include improvements to settlement efficiency at the protocol level as well as optimization to the funds that need to be allocated. A high level of backward compatibility with CIP-0056 means that the evolved standard will further foster interoperability between wallets, apps, and assets across crypto and traditional finance."

CIP-0056 remains the current deployed standard, so the security analysis below is framed around V1.

More information about the Canton standard lifecycle (and what statuses like Approved and Final mean) can be found here: Canton CIP lifecycle.

Introduction and Motivation

A fungible token looks like a simple thing: a name, a supply, and the ability to move units from one owner to another. But its behavior is inseparable from the ledger it lives on. The chain underneath dictates how state is stored, who is allowed to see it, and what "authorization" even means. Two chains can expose an almost identical token API and still rest on completely different assumptions about ownership, privacy, and trust. This article looks at that gap by comparing two token standards, Ethereum's ERC-20 and Canton's CIP-56, and, more to the point, the two very different worlds they are built in.

Ethereum: global state, public by default

Ethereum is a public, permissionless blockchain built around a global-state account model. Every account has a balance, and the entire world state is a single shared structure that every node agrees on and anyone can read. A token here is just a smart contract that keeps its own internal ledger, typically a mapping(address => uint256) recording how much each address holds. There is no privacy: balances, transfers, and contract code are all public, and any address may interact with any contract without asking permission. Authorization is implicit. The caller is whoever msg.sender says it is, either an EOA that signed the transaction or the contract that made the call, and the contract's own logic decides what that caller is allowed to do. This openness is Ethereum's defining strength. Tokens compose freely, and anyone can build on top of anyone else's contract.

Canton: private state, multi-party by design

Canton starts from the opposite premise. It is a privacy-preserving network with no single global balance sheet that everyone can read. Instead of an account model, Canton uses an extended UTXO model (Unspent Transaction Output) implemented in the Daml smart-contract language: state lives in individual contracts, with a one-to-one correspondence between a Daml contract and a UTXO. A holder's balance, then, is not one number in a shared map. It is a set of separate holding contracts. (We unpack how the UTXO model works a little later; what matters here is what it implies for trust and privacy.)

Each holding records who owns it and, unlike physical cash (a bearer instrument), is co-signed by the token's registry. These contracts are distributed only to the parties who have a legitimate stake in them; information is shared on a strict need-to-know basis rather than published to the whole network. Authorization is explicit and multi-party: every contract records its signatories, and no change to it can happen without the authority of all of them. For tokens, that means a holding is co-signed by both its owner and a registry (the admin party), and the registry keeps control over the structure of the workflows in which the token can move.

How the chain reshapes the token

Because the substrates differ so deeply, a token cannot simply be "ported" from one to the other. Its design has to change to fit the ledger it runs on. Several familiar ERC-20 concepts are rethought, or disappear entirely, once you move to CIP-56:

- **Balances become holdings.** A single mutable balance in a global map gives way to a collection of individual holding contracts (UTXOs), each owned jointly by the holder and the registry. Moving value means consuming and creating contracts, not editing a number.
- **Public reads become private, need-to-know views.** There is no global, world-readable balance for every address. A party sees only the holdings it is a stakeholder in, and even the token's total supply is reported by the registry rather than computed independently from a public ledger.
- **Implicit authorization becomes explicit co-signing.** Where ERC-20 leans on msg.sender and contract-internal checks, CIP-56 makes the registry a mandatory signatory on every holding, which gives it structural control over issuance and movement.
- **Standing allowances give way to scoped permissions.** CIP-56 describes itself as "inspired by the ERC20 standard, and covers all features of ERC20 except unconstrained allowances." The approve / allowance / transferFrom pattern is a standing, open-ended right for a third party to pull your tokens, and it does not carry over cleanly to a private UTXO ledger. Narrower, purpose-built mechanisms take its place.
- **Decimals stop being a scaling factor.** Every holding amount is a Daml Decimal (fixed-point, 10 places), so there is no per-token exponent an integration can misread. A token may still declare a decimals value in its metadata, but only as a display hint, never as a multiplier applied to a raw integer.

The table below sums up the two worlds at a glance; the sections that follow unpack each row in depth.

ERC-20 vs. CIP-56: the two worlds

Two token standards that look alike on the surface, resting on opposite foundations. Each row is unpacked in depth in the sections that follow.

Dimension	ERC-20 (Ethereum)	CIP-56 (Canton)
Ledger model	Global-state account balances (mapping(address => uint256))	Extended UTXO — one Daml contract per holding
Visibility	Fully public; readable by anyone	Private, need-to-know; holdings visible only to stakeholders
Access	Permissionless — any address may interact	Registry-controlled workflows
Authorization	Implicit via msg.sender + contract checks	Explicit multi-party signatories; registry co-signs every holding
Balance representation	One mutable number per address	A set of individual holding contracts (UTXOs)
Delegated spending	approve / allowance / transferFrom	No unconstrained allowances; scoped mechanisms instead
Decimals	Per-token, self-declared	Amounts always Decimal (10 dp); metadata decimals is display-only

A Short Refresher on ERC-20

ERC-20 is so widely deployed that most readers already carry a mental model of it, so this is a refresher rather than a deep dive. At its core, ERC-20 is a small interface: an agreed-upon set of functions a contract must expose so that wallets, exchanges, and other contracts can treat any token the same way. The standard defines six functions and two events:

```
function totalSupply() external view returns (uint256);
function balanceOf(address account) external view returns (uint256);
function transfer(address to, uint256 value) external returns (bool);
function approve(address spender, uint256 value) external returns (bool);
function allowance(address owner, address spender) external view returns (uint256);
function transferFrom(address from, address to, uint256 value) external returns (bool);

event Transfer(address indexed from, address indexed to, uint256 value);
event Approval(address indexed owner, address indexed spender, uint256 value);
```

All of the token's state lives inside the contract itself, in two mappings and a counter. balanceOf reads from a mapping(address => uint256) of balances, totalSupply is a single counter, and delegated spending is tracked in a nested mapping(address => mapping(address => uint256)) of allowances. transfer moves value from the caller (msg.sender) to a recipient, while the approve / transferFrom pair lets an owner grant a third party the right to move up to a set amount on their behalf. That is essentially the whole model: edit-a-number bookkeeping over a globally readable, permissionless ledger. It is simple, and that simplicity is a big part of why it won.

The de-facto standard: OpenZeppelin

In practice almost nobody writes an ERC-20 from scratch. The reference the ecosystem actually builds on is OpenZeppelin Contracts, whose audited ERC20.sol is the de-facto implementation of the standard. As of this writing, the latest stable release is v5.6.1 (February 2026), on the 5.x line. The core ERC20 contract holds exactly the state you would expect: two mappings, a counter, and the metadata fields.

```
abstract contract ERC20 is Context, IERC20, IERC20Metadata, IERC20Errors {
    mapping(address account ⇒ uint256) private _balances;
    mapping(address account ⇒ mapping(address spender ⇒ uint256)) private _allowances;
    uint256 private _totalSupply;
    string  private _name;
    string  private _symbol;
    // ...
}
```

Two design points from the current 5.x line are worth flagging, because they shape how token authors extend it today:

- **A single _update hook.** Earlier (4.x) versions exposed _beforeTokenTransfer and _afterTokenTransfer hooks for customization. The 5.x line removed both and funnels every balance change (mints, burns, and transfers) through one internal _update(from, to, value) function. As a result _transfer, _mint, and _burn are no longer virtual; any custom logic on balance changes (fees, pausing, supply caps) is expected to be added by overriding _update. This concentrates the token's most security-sensitive path into a single place.
- **Reverts instead of false.** The original EIP signals failure by returning false, which historically led to silently ignored failed transfers. OpenZeppelin's implementation instead reverts with typed custom errors (via IERC20Errors), so a failed transfer aborts the transaction rather than quietly returning false. The bool is kept for ABI compatibility: transfer() still returns true on success, it just never returns false.

One more detail is worth holding onto. OpenZeppelin's default decimals() returns 18 and must be overridden to change it. That makes precision a self-declared, per-token value. That is exactly the freedom CIP-56 removes: amounts are always Decimal, and a token's metadata decimals value is a display hint rather than a multiplier. It rounds out the model-level contrasts the rest of the article builds on.

The CIP-56 Model in Depth

Where ERC-20 is a single contract that anyone can deploy and call, CIP-56 (the "Canton Network Token Standard," approved 31 March 2025) describes a small ecosystem of cooperating parties and interfaces. It is deliberately not one monolithic contract. Instead of asking "what functions must a token expose?", CIP-56 asks "who are the participants, what interfaces do they speak, and what workflows do they run together?" Once you have the actors, the APIs, and the workflows straight, the security model becomes much easier to read.

The UTXO model: think of physical cash

Everything below rests on that extended UTXO model, and the easiest way to understand it is to think about the cash in your wallet. When you carry $73, you do not hold a single "73" figure that a bank increments and decrements. You hold a collection of individual bills: a $50, a $20, and three $1 notes. Each bill is a discrete, indivisible object with its own value. A UTXO ledger works the same way. Your balance is simply the sum of the separate "notes" (the unspent outputs) you currently hold.

The metaphor also explains how spending works. Say you want to pay someone $65. You cannot tear a corner off your $50 bill; instead you hand over the whole $50 and a $20 (worth $70 together) and get $5 in change. In UTXO terms, the two bills you handed over are consumed (they stop existing as valid notes), and two brand-new notes are created: a $65 note for the recipient and a $5 "change" note that comes back to you. An account model, by contrast, works like a bank statement. It never destroys or creates notes, it just edits the single number on your line of the ledger.

This "consume the inputs, create fresh outputs" mechanic is where the name Unspent Transaction Output comes from: the only notes that count are the ones that have not yet been spent. In Canton each of these notes is a holding contract rather than a paper bill, co-signed by its owner and the registry, and this same consume-and-create pattern shows up again when we walk the reference implementation's transfer engine.

Three kinds of actors

CIP-56 is built around three roles, and keeping them straight is the key to reading everything else:

- **Asset registries** are the parties that issue and maintain the ownership records for a tokenized asset. A registry is the mandatory co-signer (admin) on every holding of its token and controls the shape of the workflows in which that token can move. Real examples include the registry backing Canton Coin, or a stablecoin issuer minting on the network.
- **Wallets / custody solutions** are the software that lets an investor view and manage their holdings, potentially across many different registries at once. Wallets read a user's holdings and construct the transactions that move them.
- **Apps** are any other service that interacts with tokenized assets: exchanges, OTC desks, collateral managers, payment processors. Apps typically request that value be moved or settled; they do not hold the registry's signing authority.

Here is a useful contrast. On Ethereum these roles collapse into "the contract and whoever calls it." On Canton they are distinct parties with distinct authority, and much of CIP-56's safety comes from that separation.

The six APIs

CIP-56 factors a token's functionality into six APIs. Each has an on-ledger Daml interface (the authoritative, signed logic) and, optionally, an off-ledger HTTP (OpenAPI) part used to distribute data that is not shipped on-ledger. The table maps each to its rough ERC-20 analogue:

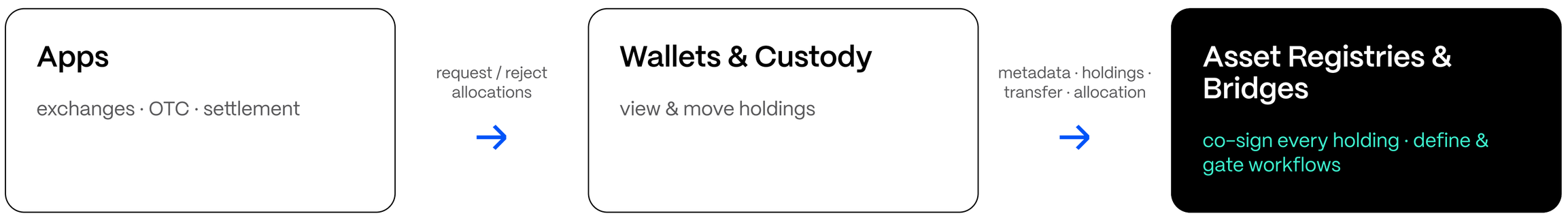
CIP-56 API	Purpose	Closest ERC-20 analogue
token metadata	**name, symbol, optional total supply, decimals (display hint)**	name(), symbol(), decimals(), totalSupply()
holdings	populate a portfolio view and transaction history	balanceOf()
transfer instruction	initiate and track a peer-to-peer (free-of-payment) transfer	transfer()
allocation	earmark specific holdings to settle a delivery-vs-payment trade	fine-grained control over approve / allowance for settlement
allocation request	a uniform way for an app to request an allocation	—
allocation instruction	a uniform way for a wallet to create an allocation	—

Source: CIP-0056 specification.

The first three cover the everyday "what do I own and how do I send it" surface that maps almost one-to-one onto ERC-20. The three allocation APIs are where CIP-56 diverges most. They are the standard's answer to delegated spending, built around single, scoped settlements rather than the standing allowances of ERC-20. Allocations get their own section further down; for now it is enough to know they exist and why there are three of them: one to request, one to create, and one that models the funded allocation itself.

How the actors interact

Apps, wallets, and registries speak the six APIs to one another. The registry is the authority at the centre of every flow — it co-signs every holding and owns the workflow definitions.



Apps also execute, cancel, and withdraw allocations directly with registries, and wallets read a token's metadata from the registry. Every arrow ultimately terminates at the registry, because no holding can be created, moved, or archived without its signature.

The six APIs

token metadata · holdings · transfer instruction · allocation · allocation request · allocation instruction

Each API is a Daml interface

The phrase "on-ledger Daml interface" is worth making concrete, because it is the mechanism that lets any wallet or app treat any registry's token uniformly. Each API is defined as a Daml interface: a viewtype plus a set of choices every implementation must provide. The holdings API, for example, is the Holding interface shipped in the standard's holding-v1 package:

```
interface Holding where
  viewtype HoldingView

data HoldingView = HoldingView with
    owner : Party
    instrumentId : InstrumentId
    amount : Decimal
    lock : Optional Lock
    meta : Metadata
```

Source: Splice.Api.Token.HoldingV1.

A concrete token then supplies an interface instance that projects its own template onto this view (we see exactly that in the next section). An app that knows only the interface can read any holding's owner, instrument, amount, and lock without knowing, or caring, which registry implemented it. This is CIP-56's analogue of "any wallet can call balanceOf on any ERC-20": the uniformity lives in the interface, not in a single shared contract.

No standardized issuance

One boundary is worth stating outright, because an ERC-20 reader will go looking for it: CIP-56 does not standardize issuance. There is no mint or burn API. New holdings come into existence under the registry's own policy, but never by the registry alone. Because a holding is signatory admin, owner, creating one needs the prospective owner's authority too. The reference template has no mint choice at all, and its test harness funds parties with submitMulti [admin, owner]. How much is issued, when, and against what backing is a registry-specific policy the standard deliberately leaves open. CIP-56 governs how value moves and settles, not how it is created or destroyed.

Three workflows

Those APIs come together in three concrete workflows:

- **Portfolio view.** A wallet reads the investor's holdings and in-progress transfers directly from the Ledger API of the validator node hosting that user's parties, and shows total supply as reported by the registry. Reporting total supply is optional: the spec leaves it to the registry's off-ledger metadata API, and a registry that keeps its authoritative records off Canton may not expose holdings or a supply figure at all. There is no global scan of a public chain; a user simply reads the private state they are a stakeholder in. (That total supply is reported rather than computed independently is a trust boundary we come back to.)

- **Direct peer-to-peer / free-of-payment (FOP) transfer.** The sender initiates a transfer, and the registry executes it within a deadline. Depending on the token, this may settle in a single Daml transaction or need additional registry-specific steps, and the registry is explicitly allowed to abort a transfer instruction it cannot execute. This is the everyday "send Alice 100 tokens" path.

- **Delivery-vs-payment (DvP) allocation.** Several transfers settle atomically, all or nothing, inside one Daml transaction. Each participant allocates the holdings they are contributing for a bounded window; once every required allocation exists, a settlement app submits the single transaction that triggers all the transfers at once. This is what makes "cash and asset change hands simultaneously, or neither does" a first-class, native operation rather than something bolted on with escrow contracts.

Key design choices (and why they matter)

Four decisions in the standard do most of the security work, and each ties back to the substrate:

- **Privacy / need-to-know.** Information about holdings and transfers is shared only with the parties that need it, not published to the whole network. There is no world-readable balance map to scrape or index.
- **Registry control.** Because the registry co-signs every holding and owns the workflow definitions, it can shape, gate, and even halt token movement. That level of control is central to regulated, institutional use cases.
- **Decimal amounts, not a scaling exponent.** All standard Canton tokens carry amounts as Daml's 38-digit fixed-point Decimal type (10 places after the point), so there is no per-token exponent applied to a raw integer for an integration to misread. The spec makes this choice explicitly "to reduce the likelihood of conversion errors in apps, wallets, or registries." A token's metadata may still declare a decimals value (0–10), but only as a display hint, so the whole class of 6-vs-18 decimal-confusion bugs that plague ERC-20 integrations does not arise.
- **Off-ledger HTTP for UTXO distribution.** Because holdings are private UTXOs and not shipped on-ledger, the parties that need them (wallets, registries) fetch them over standardized HTTP endpoints. This keeps the ledger private but adds an operational surface, the availability and integrity of those endpoints, that has no equivalent in ERC-20 and that we scrutinize later.

Put those together and you get a token that is private by default, mediated by a trusted registry, atomic in settlement, and free of both per-token decimal scaling and standing allowances. The next sections make this concrete, first by walking a real reference implementation, then by examining, class by class, how the model reshapes the token's attack surface.

Walking the Reference Implementation

Specifications are easiest to reason about once you can see the code. For CIP-56 the natural companion is OpenZeppelin's canton-token-template: roughly 800 lines of Daml that implement all six CIP-56 on-ledger interfaces, alongside 36 tests and its own verification report. It is explicitly a teaching and reference implementation. The README carries the blunt warning that "this software is experimental and not intended for production use," and it targets Daml SDK 3.4.10 on Daml-LF 2.1. Caveats aside, it is an excellent, self-contained way to see how the abstract standard becomes concrete Daml, and we use it here to ground the security discussion rather than as a production registry.

If you are reading Daml for the first time, three terms are worth knowing up front. A template is the definition of a contract type (its fields and its rules). A signatory is a party whose authority is required for the contract to exist; creating or archiving it needs their consent. A choice is an action that can be exercised on a contract, running with the authority of its controller. Almost everything about CIP-56's security model comes down to who signs what.

The seven templates

Template	Role
`SimpleHolding`	An unlocked holding — the basic "note" of value; signatories admin, owner
`LockedSimpleHolding`	A holding locked for a pending transfer or allocation; signatories admin, owner, lock.holders
`SimpleTokenRules`	The factory contract; implements both TransferFactory and AllocationFactory; signatory admin
`SimpleTransferInstruction`	A pending two-step transfer awaiting the receiver; signatories admin, sender
`SimpleAllocation`	Funds committed to a DvP settlement leg; signatories admin, sender
`SimpleAllocationRequest`	A minimal request-for-allocations used to drive DvP flows; signatory executor
`TransferPreapproval`	A receiver's standing consent to accept direct transfers; signatories admin, receiver

The holding templates

The SimpleHolding template is the whole model in miniature. Notice two things. The holding is co-signed by both the admin (the registry) and the owner, and it carries an ensure amount > 0.0 clause so a zero-value holding can never exist.

```
template SimpleHolding
  with
    admin : Party
    owner : Party
    instrumentId : InstrumentId
    amount : Decimal
    meta : Metadata
  where
    signatory admin, owner
    ensure amount > 0.0
    interface instance Holding for SimpleHolding where
      view = HoldingView with
        owner; instrumentId; amount; lock = None; meta
```

Source: simple-token/daml/SimpleToken/Holding.daml.

Because the registry is a mandatory signatory on every holding, no unit of the token can be created, moved, or destroyed without the registry's authority taking part in the transaction. This is the structural root of the "registry control" idea from earlier, and notice that the code never checks a policy to enforce it. It follows directly from who has to sign.

The locked holding and the Lock type

Value that is mid-flight, committed to a pending transfer or a DvP settlement, lives in a LockedSimpleHolding. It is the same "note" as a SimpleHolding plus a Lock and a set of extraObservers that widen visibility to exactly the counterparties who need it (a transfer receiver, a settlement executor). The Lock type comes straight from the holding-v1 interface:

```
data Lock = Lock with
    holders : [Party]            -- parties whose authority the lock binds
    expiresAt : Optional Time    -- absolute deadline at which the lock expires
    expiresAfter : Optional RelTime
    context : Optional Text      -- human-readable reason, shown by wallets
```

Source: Splice.Api.Token.HoldingV1.

The locked template adds two things that matter for the security model: the lock's holders are signatories (so in this implementation admin must co-sign the lock), and a single owner-only escape hatch that works only after the lock has expired:

```
template LockedSimpleHolding
  with
    admin : Party; owner : Party; instrumentId : InstrumentId
    amount : Decimal; lock : Lock; extraObservers : [Party]; meta : Metadata
  where
    signatory admin, owner, lock.holders
    observer extraObservers
    ensure amount > 0.0
    choice LockedSimpleHolding_Unlock : ContractId SimpleHolding
      controller owner
      do
        now ← getTime
        case lock.expiresAt of
          None   → fail "Lock has no expiry and cannot be owner-expired"
          Some t → assertMsg "Lock has not yet expired" (now ≥ t)
        create SimpleHolding with admin; owner; instrumentId; amount; meta
```

Source: simple-token/daml/SimpleToken/Holding.daml.

This is the mechanism behind the "self-expiring" property that allocations rely on later. The deadline does not archive anything by itself, but once it passes, the owner can unilaterally turn the locked holding back into a normal one.

The factory: SimpleTokenRules

Every transfer and every allocation flows through one contract: SimpleTokenRules. It is the registry's rulebook, and its shape encodes two facts that matter for security. First, it is signed by the admin alone. It is the registry's own contract, which is why archiving it is enough to halt all new transfers and allocations (a point we return to). Second, a single instance can serve many instruments at once, gated by a supportedInstruments allow-list:

```
template SimpleTokenRules
  with
    admin : Party
    supportedInstruments : [Text]      -- one factory can serve many instruments
  where
    signatory admin
    -- implements BOTH the TransferFactory and the AllocationFactory interfaces
```

Source: simple-token/daml/SimpleToken/Rules.daml.

The same contract implements both the TransferFactory and the AllocationFactory interfaces: the everyday transfer engine and the DvP settlement engine live side by side, sharing the same admin authority and the same input-handling helpers. The two factory choices are the everyday entry points, and the registry mediates both, but they are not the only sites that create holdings. The preapproval send, the instruction accept/reject/withdraw paths, allocation execute/cancel/withdraw, and the owner's expired-lock unlock all create one too. What constrains creation is not a narrow entry point but the signatory admin, owner pair and ensure amount > 0.0 on every create.

Before either engine does anything irreversible, it runs a battery of invariant checks. The transfer path validates that the caller named the right admin, the amount is positive, the request is not post-dated, the deadline is still in the future, the instrument is one this factory supports, and the inputs are non-empty. Only then does it archive and sum them:

```
assertMsg "expectedAdmin does not match factory admin"  (arg.expectedAdmin == admin)          -- #1
assertMsg "Transfer amount must be positive"            (transfer.amount > 0.0)               -- #6
assertMsg "requestedAt must not be in the future"       (transfer.requestedAt <= now)         -- #2
assertMsg "executeBefore must be in the future"         (transfer.executeBefore > now)        -- #3
assertMsg "instrumentId.admin must match factory admin" (transfer.instrumentId.admin == admin) -- #7
assertMsg "Instrument not supported by this factory"    (transfer.instrumentId.id `elem` supportedInstruments)
assertMsg "inputHoldingCids must not be empty"          (not $ null transfer.inputHoldingCids) -- #8
totalInput <- archiveAndSumInputs transfer.sender transfer.instrumentId transfer.inputHoldingCids
```

Source: simple-token/daml/SimpleToken/Rules.daml.

None of these are stylistic niceties. Each closes a concrete hole: a mismatched admin, a post-dated request, an input from the wrong instrument, an unsupported token. The allocation engine runs the same battery plus two extra deadline invariants (allocateBefore > now and allocateBefore <= settleBefore), because a settlement has both an allocation window and a settlement window. Only once every check passes does the factory reach the actual dispatch.

The three-way transfer dispatch

With those checks passed and the inputs summed, the transfer engine dispatches into one of three paths depending on the situation:

```
-- Invariant #18: sum(inputs) ≥ amount
assertMsg ("Insufficient funds: have " <> show totalInput <> " but need " <> show transfer.amount)
  (totalInput ≥ transfer.amount)

-- 3-way dispatch
if transfer.sender == transfer.receiver then
  selfTransfer admin transfer totalInput arg.extraArgs.meta          -- merge / split
else do
  optPreapprovalCid ← lookupFromContextU @(ContractId TransferPreapproval)
                        arg.extraArgs.context transferPreapprovalContextKey
  case optPreapprovalCid of
    Some preapprovalCid → directTransfer admin transfer totalInput preapprovalCid arg.extraArgs.meta
    None                → twoStepTransfer admin transfer totalInput
```

Source: simple-token/daml/SimpleToken/Rules.daml.

- **Self-transfer (sender == receiver)** merges or splits your own holdings (defragmenting your "notes"). The inputs are consumed and a single new holding is created, plus change if needed. Completes immediately.
- **Direct transfer (a preapproval is supplied in the choice context):** the factory exercises the receiver's TransferPreapproval, which creates the receiver's holding. This works only because the preapproval is co-signed by the receiver, so it carries the receiver's authority. Completes immediately.
- **Two-step transfer (no preapproval, and sender != receiver):** the funds are locked into a LockedSimpleHolding and a SimpleTransferInstruction is created. The transfer stays pending until the receiver explicitly Accepts it (or rejects it, or the sender withdraws).

That third path is CIP-56's structural answer to a classic ERC-20 hazard: sending tokens to a party (or contract) that cannot handle them, so they get stuck. Here the receiver's holding is only ever created with the receiver's own signatory authority, either through a standing preapproval they created or through an explicit accept. A transfer can never force value onto an unwilling or unaware receiver.

The two-step accept

The Accept choice makes this concrete. The receiver exercises it, so it runs with the receiver's authority. It validates the deadline before touching any state, archives the locked holding, and only then creates the receiver's holding (plus any change back to the sender):

```
transferInstruction_acceptImpl selfCid arg = do
  now ← getTime
  assertMsg "Transfer has expired (executeBefore passed)" (now < transfer.executeBefore)
  lockedHolding ← fetch lockedHoldingCid
  archive lockedHoldingCid
  receiverCid ← create SimpleHolding with
    admin; owner = transfer.receiver
    instrumentId = transfer.instrumentId
    amount = transfer.amount
    meta = arg.extraArgs.meta
  -- ... sender change created if lockedHolding.amount > transfer.amount ...
```

Source: simple-token/daml/SimpleToken/TransferInstruction.daml.

Because the receiver's SimpleHolding is created inside a choice the receiver controls, the registry and sender together still cannot conjure a holding onto an unwilling party. The receiver's signature ends up on the output by construction, not by a policy check.

Consuming inputs: how double-spend becomes a ledger conflict

The most security-relevant helper is archiveAndSumInputs. Before any new holding is created, it fetches each input holding, checks that it belongs to the sender and matches the instrument, enforces the lock rules (expired locks may be reused, unexpired ones may not), and, crucially, archives each input before returning its amount to be summed:

```
archiveAndSumInputs : Party → InstrumentId → [ContractId Holding] → Update Decimal
archiveAndSumInputs sender expectedInstrumentId holdingCids = do
  amounts ← forA holdingCids $ \holdingCid → do
    holdingI ← fetch holdingCid
    let hv = view holdingI
    assertMsg "Input holding owner does not match sender"          (hv.owner == sender)
    assertMsg "Input holding instrumentId does not match transfer" (hv.instrumentId == expectedInstrumentId)
    case hv.lock of
      None → pure ()
      Some lock → do
        now ← getTime
        let lockExpired = case lock.expiresAt of
              None   → False
              Some t → now ≥ t
        assertMsg "Cannot use holding with unexpired lock as transfer input" lockExpired
    archive holdingCid                    -- archived BEFORE any output is created
    pure hv.amount
  pure (sum amounts)
```

Source: simple-token/daml/SimpleToken/Rules.daml.

Archiving the inputs first is what makes double-spending a non-issue by construction. In Canton's UTXO model, two transactions that both try to consume the same holding contract conflict at the ledger level: exactly one commits and the other is rejected. There is no window in which the same holding backs two outputs. A double-spend attempt never turns into a subtle application-logic bug to reason about; it becomes an ordinary contention conflict that the ledger resolves, and the losing client simply retries. This mirrors how spending the same physical bill twice is impossible: once you have handed it over, you no longer have it.

The same discipline runs through the rest of the lifecycle. The two-step accept, allocation execution, and withdrawal choices all validate their deadline first, then archive the locked holding, then create the outputs. The token's core arithmetic (inputs equal the receiver's amount plus the sender's change) falls directly out of this consume-then-create structure: every output derives from the summed, already-archived inputs, so no path can create value that was not first consumed.

Authorization and the Allowance Question

If there is one place where the two models diverge most sharply, it is here. CIP-56 states it is "inspired by the ERC20 standard, and covers all features of ERC20 except unconstrained allowances." That one exception is not a gap. It is a deliberate design decision, and it removes the exact primitive behind the most damaging class of ERC-20 losses. Understanding why an ERC-20 allowance cannot be ported to a private UTXO ledger, and what CIP-56 puts in its place, is where the whole comparison turns.

How ERC-20 delegated spending works

ERC-20 lets an owner authorize a third party (a "spender") to move tokens on their behalf, using three interface members:

```
function approve(address _spender, uint256 _value) external returns (bool);
function allowance(address _owner, address _spender) external view returns (uint256);
function transferFrom(address _from, address _to, uint256 _value) external returns (bool);
```

The owner calls approve to set a standing allowance, a number recorded in the token's allowances mapping. From then on, the spender may call transferFrom to pull tokens from the owner, repeatedly, up to that amount, until the owner changes it back. This is the mechanism behind virtually every DeFi interaction: you approve a router or protocol once, and it moves your tokens whenever you trade. It is enormously convenient, and that convenience is exactly the problem.

Three structural weaknesses

Three structural weaknesses follow from a standing, mutable, on-chain grant:

- **The approve-race.** Changing an existing allowance is not atomic with respect to the spender's ability to use it. If an owner has approved a spender for X and later wants to lower it to Y, a malicious spender watching the mempool can spend the original X before the change lands, then spend Y again afterward, pulling X + Y in total. The EIP-20 specification itself acknowledges this and can only suggest a UI convention (set the allowance to 0 first) rather than a fix, because enforcing it in the contract would break backward compatibility.
- **Infinite approvals.** To avoid re-approving on every interaction, wallets and dApps routinely request the maximum possible allowance. The result is a standing, effectively unlimited right to drain the owner's balance, sitting on the token indefinitely and easy to forget about.
- **Off-chain signature phishing.** Extensions like EIP-2612 Permit and Uniswap's Permit2 let an owner grant an allowance by signing a message off-chain, with nothing recorded at signing time. A victim tricked into signing such a message may see nothing on-chain until the attacker submits the signature later to drain their tokens.

What ties these together is that the authority is standing (it persists), ambient (the spender chooses when and how much to pull), and hard to reason about (a signature or approval made once can be exercised much later, in a context the owner has forgotten).

The cost of standing allowances

The losses this primitive produces are not hypothetical — Scam Sniffer's annual wallet-drainer reports isolate the approval/signature vector by excluding direct hacks, exchange breaches, and contract exploits.

$494M
lost to wallet drainers in 2024, across roughly 332,000 victims

56.7%
of >$1M losses from Permit signatures alone (60.2% with increaseAllowance)

$83.85M
lost in 2025 — down 83% — across 106,106 victims

$6.5M
single largest 2025 theft: one malicious Permit signature

They put losses at $494M across roughly 332,000 victims in 2024. Scam Sniffer breaks signature types down only for the 30 cases above $1M ($171M combined), and there Permit alone accounts for 56.7%, or 60.2% once increaseAllowance is counted with it. In 2025 the figure fell 83% to $83.85M across 106,106 victims, yet the mechanism was unchanged: Permit/Permit2 signatures still accounted for 38% of all losses over $1M, including the year's single largest theft, $6.5M drained by one malicious Permit signature. So the standing allowance and its off-chain-signature descendants have been, by a wide margin, the dominant wallet-draining technique on Ethereum, and Scam Sniffer's own caveat is that the decline may partly reflect a shift toward harder-to-track attack vectors, not that the primitive became any safer.

Source: Scam Sniffer 2025 Annual Report, via Cointelegraph.

Ethereum's safer primitives are still unfinished

Ethereum is not standing still here, but the safer primitives are still explicitly unfinished. OpenZeppelin's 5.x line ships only a draft ERC20TemporaryApproval extension (added in v5.1): an ERC-7674 transient-storage approval that lives for a single transaction. As of this writing it is still shipped under the draft- file prefix and carries the warning that "this is a draft contract [and] the corresponding ERC is still subject to changes." There is no stabilized, non-draft release, because ERC-7674 itself is still in Review status, not yet Final. Permit2, likewise, layers expiries and nonces on top of raw allowances as an opt-in add-on. The direction of travel is telling: the ecosystem is visibly converging toward narrower, more scoped grants, the same direction CIP-56 takes by construction. But on Ethereum these remain opt-in bolt-ons, still stabilizing, layered over a primitive that is unconstrained by default. CIP-56 simply never introduces the unconstrained primitive in the first place.

Why CIP-56 has no allowances

CIP-56 does not offer a weaker allowance or a safer allowance — it offers none at all, and the spec is explicit about why:

> "This CIP does no[t] standardize an allowance API for two reasons: (1) We expect allocations to cover the majority of use-cases for allowance. (2) Allowances as structured in ERC20 do not work well with a UTXO model and privacy: the third-party holding the allowance is neither privy to the UTXOs that they could spend nor would they know which ones they can use without causing undue contention with concurrent activity by the owner or other parties with an allowance."
>
> CIP-0056 specification, "Relation to ERC20"

Source: CIP-0056 specification, "Relation to ERC20".

Read against the substrate laid out earlier, the reason is almost mechanical. An ERC-20 allowance works because the spender can read the owner's single global balance and pull from it. On Canton there is no such global balance to point at: a holder's value is a set of private holding contracts that the spender cannot even see, let alone select. And even if the spender somehow knew which holdings to spend, picking them unilaterally would collide with the owner's own concurrent activity, producing exactly the contention conflicts described earlier. A standing "spend up to X from wherever" grant has nothing coherent to attach to in a private UTXO world. The spec notes that a UTXO-compatible allowance API "is left to a future CIP"; for now, the primitive is deliberately absent.

Executing the settlement

The contrast comes down to one line: an ERC-20 allowance is a standing, open-ended permission to pull value, while a CIP-56 allocation is a scoped, expiring, single-purpose lock of value. The first optimizes for composability and pays for it with a large standing attack surface; the second optimizes for controlled settlement and accepts more ceremony per trade.

Creating an allocation only funds a leg; the settlement itself is a separate step. Each allocation exposes an allocation_executeTransferImpl choice that, like every other lifecycle choice, validates the deadline first, archives the locked holding, and then creates the receiver's holding plus any sender change:

```
allocation_executeTransferImpl _selfCid arg = do
  now ← getTime
  assertMsg "Settlement has expired (settleBefore passed)"
    (now < allocation.settlement.settleBefore)
  lockedHolding ← fetch lockedHoldingCid
  archive lockedHoldingCid
  let leg = allocation.transferLeg
  receiverCid ← create SimpleHolding with
    admin; owner = leg.receiver; instrumentId = leg.instrumentId
    amount = leg.amount; meta = arg.extraArgs.meta
  -- ... sender change created if lockedHolding.amount > leg.amount ...
```

Source: simple-token/daml/SimpleToken/Allocation.daml.

On its own this executes a single leg. The delivery-versus-payment guarantee comes from Daml's atomicity: a settlement app exercises the execute choice on every leg's allocation inside one Daml transaction. Because a Daml transaction commits all-or-nothing, either every leg settles or none does. There is no intermediate state in which the cash side moved but the asset side did not. That is what lets DvP be a native ledger property rather than something reconstructed with escrow contracts and a trusted intermediary holding both sides. (It also carries a structural precondition we return to: every holding the settlement touches must live on the same synchronizer.)

The replacement: allocations

What CIP-56 offers instead is the allocation: not a standing permission to pull value, but a specific, pre-funded commitment of value to a single settlement. In the reference implementation, creating an allocation locks the sender's funds into a LockedSimpleHolding and records a SimpleAllocation. The shape of that locked holding tells the whole story:

```
lockedCid ← create LockedSimpleHolding with
  admin
  owner = leg.sender
  instrumentId = leg.instrumentId
  amount = leg.amount                          -- a specific amount, locked now
  lock = Lock with
    holders = [admin]
    expiresAt = Some settlement.settleBefore -- auto-releases at the deadline
    expiresAfter = None
    context = Some "allocation"
  extraObservers = [settlement.executor, leg.receiver]  -- scoped visibility
  meta = emptyMetadata
```

Source: simple-token/daml/SimpleToken/Rules.daml.

Every property here is the opposite of a standing allowance:

- **Pre-funded, not ambient.** The exact amount is locked out of the sender's holdings at allocation time. There is no "pull whenever you like up to a limit"; the value is already committed to this settlement and nothing else.
- **Time-bounded and self-expiring.** The lock's expiresAt is the settlement deadline (settleBefore). The spec states as design intent that holdings are "only locked to an allocation until that deadline, and become available again to their owner immediately thereafter." A precise reading matters here: expiry does not automatically archive the locked holding or move the funds. The LockedSimpleHolding stays on-ledger until the owner acts. What the deadline changes is authority. Once expired, the owner can reclaim the funds unilaterally (via LockedSimpleHolding_Unlock, which only the owner needs to exercise, since the admin's authority rides along from the locked holding's signatory set), and an expired lock is accepted as an input to a new transfer. So a forgotten allocation does not linger as a live risk (the lock stops binding at the deadline) even though the contract itself lingers until someone spends or unlocks it.
- **Scoped to a specific counterparty and executor.** The allocation names its receiver and its settlement executor, who are the only extra parties that can see it. There is no broad "any spender, any time" grant.
- **Single-purpose.** The allocation exists to execute one settlement leg. Executing it (allocation_executeTransferImpl) validates the deadline, archives the locked holding, and creates the receiver's holding; withdrawing or cancelling returns the funds to the sender. Once resolved, it is gone, with no reusable residual authority left behind.

The attack classes this removes

Because the primitive itself is absent, an entire family of ERC-20 attack classes has no foothold in CIP-56, not because the code defends against them but because there is nothing to attack. The approve-race, infinite-approval drains, stale forgotten approvals, and Permit/Permit2 off-chain signature theft all presuppose a standing, mutable grant that CIP-56 never creates. Authority to move value comes from signatories on a specific transaction, not from a reusable signed permission. (The next section places these alongside the rest of the ERC-20 attack surface in a single consolidated matrix.)

This is the sharpest illustration of the article's central theme. ERC-20's allowance is the price of its permissionless composability, a standing grant that anyone's contract can build against. CIP-56 refuses that trade. It gives up the frictionless "approve once, use forever" model, and in exchange the whole approval-phishing and allowance-race surface does not exist. What you gain in removed attack surface, you pay for in the registry-mediated, per-settlement ceremony that the next section examines from the other direction.

Attack Class by Attack Class

Allowances are the headline, but they are not the only place where the substrate reshapes the threat model. Many of the attack classes that experienced Solidity auditors check for reflexively either disappear or change shape entirely under CIP-56, usually not because the Daml code defends against them but because the ledger model gives them nothing to exploit. The table below is the consolidated view, and the notes after it explain the structural reason behind each row.

One caveat before the table

"Eliminated" here means eliminated as a class inherent to the token model, in the reference implementation. A production registry that re-adds features (fees, external ledgers, governance) can bring the risk back, which is the subject of the next section. The comparison is between the two standards as designed, not a claim that any Canton deployment is unconditionally safe.

The consolidated attack matrix

Attack class	ERC-20 exposure	CIP-56 stance	Structural reason
Allowance abuse (approve-race, infinite approval, Permit/Permit2 phishing)	Standing, mutable approve/allowance grants enable races, unlimited drains, and off-chain signature theft	Eliminated	No allowance primitive exists on a private UTXO ledger; delegated spending is a scoped, pre-funded, self-expiring allocation instead
Reentrancy	Callback hooks (ERC-777, ERC-1363) let a token hand control to attacker code mid-transfer	Eliminated	Daml transactions are atomic and have no "call out and resume" step; control never passes to attacker code mid-execution
Missing / non-standard return values	transfer/approve that return no bool (e.g. USDT) break naive integrations — the reason SafeERC20 exists	Eliminated	Choices have typed results; there is no silent "returned nothing" ambiguity to paper over
Fee-on-transfer / rebasing	Tokens that move less than requested, or silently change balances, break amount in == amount out assumptions	Structurally different	Transfers consume and create explicit holdings with explicit amounts; the reference implementation is zero-fee — amount in == amount out
Integer overflow / decimals confusion	Historic overflow bugs; 6-vs-18 decimal mismatches across integrations	Mitigated	Decimal fixed at 10 places, metadata decimals is display-only (no scaling exponent); positive-amount and sufficient-funds invariants bound the arithmetic
Double-spend / front-running / MEV	Public mempool enables front-running, sandwiching, and MEV extraction	Mitigated / different	Inputs archived before outputs, so a double-spend is a ledger-level contention conflict (one commits); no public mempool of pending transactions
Unchecked transfer to a contract / lost tokens	transfer to a contract that can't handle tokens strands them (why safeTransferFrom exists)	Eliminated	The receiver's holding is only created with the receiver's own signatory authority — via a standing preapproval or an explicit two-step accept

Attack classes in detail

Reentrancy

Reentrancy is the archetypal Ethereum vulnerability: a contract (or a transfer hook, as in ERC-777 and ERC-1363) hands control to attacker code before its own state has settled, which is why checks-effects-interactions and reentrancy guards exist. Daml has no analogue. A transaction is built and validated as one atomic tree, with no point at which it pauses to run another party's code and then resume. The pattern has nowhere to live.

Missing return values and non-standard tokens

Much of ERC-20's defensive tooling, SafeERC20 above all, exists to cope with tokens (like USDT) whose transfer/approve do not return a bool, so a naive require(token.transfer(...)) misbehaves. CIP-56's on-ledger interfaces are typed: a choice returns a typed result, so the "did nothing come back, and is that success or failure?" ambiguity never arises.

Fee-on-transfer and rebasing

Fee-on-transfer and rebasing tokens break the assumption that amount received equals amount sent, a recurring source of DeFi integration bugs. A CIP-56 transfer is not a mutation of a shared counter that can be silently taxed; it consumes specific holdings and creates specific ones with explicit amounts. The reference implementation is strictly zero-fee: the receiver gets exactly transfer.amount and the sender's change is exactly sum(inputs) − amount, an identity fixed by the transfer logic itself. A production registry could model fees, but only as explicit, visible holdings, never as a hidden discrepancy.

Attack classes in detail (continued)

Overflow and decimals confusion

Raw integer overflow is largely historical since Solidity 0.8+ reverts on it, but decimals confusion is alive: integrating a 6-decimal token as if it were 18-decimal silently misprices everything. As covered earlier, CIP-56 fixes all standard tokens to the constant-precision Decimal type, so there is no per-token scaling exponent to misread (any metadata decimals value is a display hint, not a multiplier), and the positive-amount and sufficient-funds invariants from the reference implementation bound the rest of the arithmetic.

Double-spend, front-running, and MEV

A public mempool is what makes front-running, sandwiching, and broader MEV extraction possible on Ethereum. CIP-56 undercuts this on two fronts, both already established while walking the reference implementation. Because inputs are archived before any output, racing transactions conflict at the ledger level rather than in a public ordering game. And Canton's privacy means there is no public mempool broadcasting pending transfers to race against.

The honest caveat, and it points straight to the next section, is that this risk is relocated, not eliminated. The registry co-signs every holding and the synchronizer sequences transactions, so the ability to observe, reorder, delay, or censor pending activity is concentrated in those trusted parties. The anonymous, permissionless MEV searcher is gone; "no public MEV" just means the party that could front-run is one you have already chosen to trust.

Unchecked transfers and lost tokens

On Ethereum, a plain transfer to a contract that cannot handle tokens strands them permanently, which is the reason safe-transfer variants and receiver hooks exist. As the reference implementation showed, a CIP-56 receiver's holding can only be created with the receiver's own signatory authority, via a standing TransferPreapproval or an explicit Accept. Consent is structural, so value cannot be pushed onto a party that has not agreed to receive it.

Taken together, this is the constructive half of the thesis: a surprising number of the attack classes that dominate ERC-20 audit checklists are simply not expressible against a private, atomic, multi-signature UTXO ledger. But eliminating these classes is only one side of the ledger. The model achieves it by concentrating authority and moving data off-chain, and that concentration is itself an attack surface, which the next section takes on directly.

The New Trust CIP-56 Introduces

A comparison that stopped at the previous section would be marketing, not analysis. Every property that removes an ERC-20 attack class is bought with a new assumption, and an auditor's job is to add up the cost. The honest framing is a trade, not a win: the trust you removed (permissionless, anyone-can-exploit attack surface) is swapped for trust you added, in the form of registry integrity and operational infrastructure. This section is the other side of that trade.

The registry is a trusted party

The single most important fact about CIP-56 is that the registry (admin) co-signs every holding. That is what makes registry control possible, and it also means the registry is a mandatory participant in every movement of the token, and a party you must trust. On a public ERC-20 chain there is no such party; the contract is the only authority. Two concrete consequences follow.

First, total supply is not independently verifiable. On Ethereum, anyone can call totalSupply() and trust the answer because the contract's state is public and the computation is trustless. Under CIP-56 total supply is the sum of all active holdings, but those holdings are private, so no third party can recompute it. The number is reported by the registry, and the spec is candid about this:

> "...in contrast to public chains, shipping the total supply information on-ledger does not allow independent verification of the total supply, as the holdings from which it is computed are private. Note that this does not weaken the security, as the registries are already trusted to maintain the private holding records and can thus also be trusted to report the total supply correctly."
>
> CIP-0056 specification, "Alternatives Considered — On-Ledger Reporting of Total Supply"

Source: CIP-0056 specification, "Alternatives Considered — On-Ledger Reporting of Total Supply".

The spec's reasoning is internally consistent: if you already trust the registry to keep the holding records, trusting its supply figure adds nothing new. But an auditor should read that sentence for what it is, an explicit statement that the model's integrity rests on the registry, and that a core invariant every ERC-20 holder takes for granted (that supply is publicly checkable) is, by design, a matter of trust here.

Off-ledger endpoints & bearer capabilities

A single point of control

Second, the registry is a single point of control. Because it signs everything and owns the workflow definitions, a registry can gate, delay, or halt token movement; in the reference implementation, archiving the factory contract is enough to stop new transfers and allocations. For regulated assets this is often a feature (issuers need exactly this kind of control), but it is also a single point of failure and a censorship vector with no ERC-20 equivalent.

Off-ledger HTTP endpoints and contract-ids as bearer capabilities

Because holdings are private UTXOs distributed off-ledger, registries serve them over HTTP APIs exposed to the public internet, and the standard deliberately does not require those endpoints to be authenticated:

> "The standard does not require requests to these HTTP endpoints be authenticated, as all of them either access data that is expected to be public (e.g., registry metadata) or data that is protected by a difficult to guess identifier (e.g., the contract-id of an allocation)."

Sources: CIP-0056 specification, "Details — Off-Ledger API Discovery and Access" for the first quote, and "Rationale — Alternatives Considered — No Authentication on Registry Off-Ledger APIs" for the deferral.

So a contract-id functions as an unguessable bearer capability: anyone who knows it can fetch the corresponding data. The spec's stated protections are that genuinely sensitive data (a holder's own holdings and in-progress transfers) can only be read from the holder's own validator node, and that everything served over HTTP is guarded by hard-to-guess identifiers; authentication was deferred "to accelerate the delivery of the first version." For an auditor this is a familiar shape with familiar risks. Bearer capabilities leak through logs, browser referrers, error reports, and shared URLs, and the standard as written has no revocation, rotation, or expiry story for a leaked contract-id. It is a reasonable delivery-time trade-off, but one to flag explicitly for any deployment handling sensitive data.

Synchronizer liveness & upgradeability

Same-synchronizer requirement and liveness

Atomic DvP has a structural precondition: all the contracts a transaction touches must live on the same synchronizer.

> "...the input contracts (i.e., UTXOs) referenced by a Daml transaction must all be assigned to the same synchronizer."
>
> CIP-0056 specification, "Details — Implementation Requirements"

Source: CIP-0056 specification, "Details — Implementation Requirements".

This is why the standard recommends connecting to the Global Synchronizer: it maximizes the chance that a multi-party settlement's inputs can all be brought together on one synchronizer. But it also introduces concerns ERC-20 does not have in the same form. The synchronizer is a commit bottleneck; moving contracts between synchronizers (reassignment) leaves them temporarily unavailable; and because UTXO distribution rides on off-ledger HTTP, the token's liveness now depends on that HTTP infrastructure being up and correct, not only on the ledger. Availability becomes part of the security model, not just an operational nicety.

Upgradeability: who controls the code

An attack surface neither standard advertises, but that dominates real audit findings, is the question of who can change the token's logic after it ships, and it looks very different on the two substrates.

Upgradeability: two substrates

On Ethereum

Most production tokens are not the immutable contracts the ERC-20 mental model assumes; they sit behind an upgradeable proxy (Transparent or UUPS). Logic lives in an implementation contract that an upgrade admin can swap out, while state stays in the proxy. That pattern is the source of an entire genre of high-severity findings: storage-layout collisions between old and new implementations, uninitialized or seizable implementation contracts, unprotected upgradeTo functions, and, most consequentially, the plain fact that whoever holds the upgrade key can replace a benign token with one that mints, freezes, or drains at will. Auditors treat the upgrade admin as a full-trust role and check whether it is a timelock or multisig rather than a single externally-owned account. Upgradeability, then, is opt-in on Ethereum, but once opted into, it quietly reintroduces a central point of control on a chain whose whole premise is that there isn't one.

On Canton

The logic lives in Daml packages, and code evolves through Smart Contract Upgrades (SCU): a template is shipped as a versioned package, and a newer version can take over existing contracts only when its changes are backward-compatible under Daml's rules (for example, adding optional fields is allowed; changing the meaning of existing ones is not). This constrains the blast radius of an upgrade in a way proxy storage layouts do not, because the platform mechanically rejects many incompatible changes rather than leaving them as a latent collision bug. But it does not make code evolution trustless. Which package version a contract actually runs under depends on package vetting decided per participant node, and because the registry is a signatory on every holding, the registry remains the party that authors and drives the template definitions the token is bound to. So the Ethereum question "who holds the upgrade key?" reappears here as "who authors the packages, and whose participant nodes vet them?" It is a smaller, better-bounded lever than an arbitrary proxy swap, but a registry-controlled lever nonetheless.

The reference implementation makes the churn concrete in a different way. As the next subsection notes, it pins a fast-moving pre-release toolchain (Daml SDK 3.4.10 on Daml-LF 2.1), and that LF version's removal of contract keys already forced a design compromise. The lesson for an auditor is symmetrical across both chains: "can the code change, and who decides?" is a first-order question on either substrate, and the trustless-immutable-token assumption is wrong in both.

Even the reference implementation needed hardening

One caveat frames everything in this subsection. As noted while walking the reference implementation, this template is explicitly experimental and not production-ready: its README warns that "this software is experimental and not intended for production use," it pins a fast-moving pre-release toolchain (Daml SDK 3.4.10 on Daml-LF 2.1), and it may still change materially, and contain bugs, before anything built on it ships. Its findings should therefore be read as the ordinary growing pains of an evolving reference codebase, not as indictments of the standard itself. With that framing, the template ships its own verification report (docs/AUDIT.md), and it matters for the trust story: even a minimal, carefully written CIP-56 registry surfaced findings during review.

- **Two HIGH-severity issues**, a missing ensure amount > 0.0 on holdings and a preapproval path that accepted an unvalidated amount, were caught and patched before the repo was published; the audit cites commit 509e1f8, which predates the squashed public history and is reachable only by direct SHA. The current code carries exactly the positive-amount invariants shown in the reference-implementation walkthrough. They are the kind of checks that are easy to omit and material when omitted.

- **A MEDIUM finding remains acknowledged as unfixable under the current platform:** Daml-LF 2.1 dropped contract keys, so the singleton factory and preapproval contracts cannot enforce uniqueness at the ledger level. Uniqueness must instead be maintained by the application, a guarantee the ledger used to provide for free, and this is more than a transient toolchain gap. Contract keys are slated to return in a later Daml-LF revision, but without the uniqueness guarantee, because ledger-enforced key uniqueness is fundamentally incompatible with Canton's multi-synchronizer architecture. A CIP-56 registry should not expect to get this guarantee back.

- **Three unbounded list fields remain,** which the audit deems safe because creation requires the admin as signatory. That holds for the factory's supportedInstruments, but not for SimpleAllocationRequest.senders, which is signed only by settlement.executor and whose template has no admin field at all. So the rationale is both a design-time trust assumption and, in one case, simply inaccurate.

None of these are damning; they were found, triaged, and either fixed or consciously accepted, which is exactly how a pre-production codebase under active development should behave. The durable lesson is twofold. First, a small standard grows its invariant set as soon as it meets reality (the reference implementation's checks expanded during implementation, and would expand further in production). Second, several safety properties Ethereum developers lean on, such as public verifiability and ledger-enforced key uniqueness, are either absent or must be re-established by hand.

Conclusion

A token is a reflection of its ledger. ERC-20's attack surface is the shape of Ethereum's core promises (global state, public reads, permissionless calls, implicit msg.sender authority), and the most damaging losses in its history (the approve-race, infinite approvals, Permit phishing) are the price of the standing, ambient allowance that permissionless composability demands. CIP-56 starts from the opposite substrate, private UTXO holdings, multi-party signatories, a mandatory registry, and in doing so an entire family of ERC-20 attack classes loses its foothold: reentrancy has no mid-transaction callout to exploit, missing return values cannot occur against typed choices, unchecked transfers cannot strand tokens on an unwilling receiver, and there is no allowance primitive to race, over-approve, or phish. Conservation of value, awkward to guarantee across the open ERC-20 universe, is instead a structural property of the transfer engine: value is only ever created by consuming existing holdings.

But none of that is free, and the honest verdict is a trade, not a win. The permissionless, anyone-can-exploit surface CIP-56 removes is exchanged for trust it adds: a registry that co-signs everything and reports a total supply no third party can recompute; off-ledger endpoints whose availability becomes part of the security model; contract-ids that behave as unguessable bearer capabilities; and platform realities like the loss of contract keys under Daml-LF 2.1, which forces singleton uniqueness to be enforced in the application rather than by the ledger. Even the minimal, carefully written reference implementation grew its invariant set and needed hardening during review, and a production registry, re-adding fees, governance, and delegation, brings back much of the complexity the lean core avoided.

Neither model, importantly, is the trustless, immutable token its simplest mental picture suggests. On Ethereum the upgradeable proxy quietly reintroduces a central upgrade key on a chain built to avoid one; on Canton the registry authors the packages and drives the vetting that decides which logic a holding runs under. "Can the code change, and who decides?" is a first-order question on both substrates; it just resolves to an upgrade multisig or timelock on one, and to registry-controlled package vetting on the other. The two ecosystems are also at very different points on the maturity curve. ERC-20 is battle-tested and served by a deep bench of audited libraries, static analyzers, and hard-won conventions (SafeERC20, reentrancy guards, allowance hygiene), whereas CIP-56 is young, its reference implementation is explicitly experimental, and its toolchain is still moving underfoot. That gap is not an argument for or against either standard; it is a reminder that a smaller inherent attack surface is not the same as a smaller operational one.

For a practitioner, the takeaway is that the two models relocate where the auditor must look rather than removing the need to look. An ERC-20 review still lives and dies on allowances, proxy-upgrade safety, and the integration assumptions (bool return values, fee-on-transfer, decimals) that a permissionless, composable world makes fragile. A CIP-56 review shifts almost entirely onto the trusted core: the registry's integrity and key management, the confidentiality and availability of the off-ledger endpoints, same-synchronizer liveness, and the application-level invariants (uniqueness above all) that the ledger no longer guarantees for free. Different questions, different checklists, backed by the same discipline.

THE WHOLE THESIS IN ONE LINE

ERC-20's risk comes from what it makes public and permissionless; CIP-56's risk comes from what it makes private and registry-controlled. An auditor's job is not to crown a winner but to add up the cost each model presents, and to check it line by line.