Security Audit Report

Anoma Q3 2025: Token & Token Distributor

Last Revised
2025/09/03

Authors:
Aleksandar Stojanovic, Carlos Rodriguez

Contents

Audit overview

The project

In August 2025, Anoma engaged with Informal Systems to conduct a security audit of the XAN token system. It features an ERC-20 token supporting dual governance (voting through token locking and council) to decide on future underlying implementations of the contract. Tokens are deployed and distributed with a time-limited, Merkle-proof-based claim system for both unlocked and locked tokens.

Scope of this report

The audit focused on evaluating the correctness and security properties of the smart contracts in the `anoma/token`↗ and `anoma/token-distributor`↗ repositories, focusing specially on guaranteeing that no undesired behaviour manifests in the ERC-20 (XAN token contract), foreign reserve (fee recipient from allocation claiming) and token distributor (token allocation claim system) contracts.

Audit plan

The audit was conducted between August 18th, 2025 and August 27th, 2025 by the following personnel:

- Aleksandar Stojanovic
- Carlos Rodriguez

Conclusions

We performed a manual review supported by property-based tests. Most safety and liveness properties hold. Notable exceptions are that upgrading to a prior implementation can re-activate per-implementation governance state, causing unlocked balance underflow and governance misalignment with current ERC-20 balances. ERC-7201 storage isolation is sound with the caveat above. Configuration choices (fixed `MIN_LOCKED_SUPPLY`) can render minimum locked supply requirement unattainable if too much supply is burned after initial distribution period. In the kickoff, the team acknowledged that if a large fraction of voter's body (especially 100%) votes for the current implementation, gathering enough votes for a new upgrade can be very hard or impossible, impacting upgradeability. The team acknowledges the governance risk their design introduces. The voter body is sovereign, and if it reaches quorum and minimum locked supply for a malicious implementation, it will be chosen as a next implementation (the council cannot override it).

Audit Dashboard

Target Summary

- **Type:** Protocol and implementation
- **Platform:** Solidity smart contracts
- **Artifacts:**
  - For `token` (at commit hash `e4b0034`↗):
    `src/libs/Voting.sol`
    `src/libs/Parameters.sol`
    `src/libs/Locking.sol`
    `src/libs/Council.sol`
    `src/interfaces/IForeignReserveV1.sol`
    `src/interfaces/IXanV1.sol`
    `src/ForeignReserveV1.sol`
    `src/XanV1.sol`
  - For `token-distributor` (at commit hash `2b4e2bf`↗):
    `src/libs/Allocation.sol`
    `src/TokenDistributor.sol`
    `src/interfaces/ITokenDistributor.sol`

Engagement Summary

- **Dates**: August 18th, 2025 → August 27th, 2025
- **Method**: Manual code review, property-based testing

Severity Summary

Finding Severity	Number
Critical	0
High	0
Medium	1
Low	0
Informational	3
Total	4

System Overview

The XAN Token System consists of two interconnected contract repositories that together provide a comprehensive token distribution and governance platform:

- **Token repository** contains the core governance infrastructure with `XanV1.sol` serving as an advanced ERC-20 token featuring dual governance mechanisms, token locking for voting participation, and upgrade authorization. Alongside it, `ForeignReserveV1.sol` provides an arbitrary execution framework that can perform any contract operations as directed by the token's governance system.
- **Token distributor repository** implements a factory pattern through `TokenDistributor.sol`, which deploys both the XAN token and `ForeignReserveV1` contracts during its construction. It then orchestrates a time-bounded, Merkle-tree based distribution system that allows users to claim token allocations using cryptographic proofs, with support for both immediately available and governance-locked tokens.

The system's architecture enables initial token distribution followed by community-driven governance, with the flexibility to execute arbitrary operations through the `ForeignReserveV1` contract as the protocol evolves.

Architecture diagram

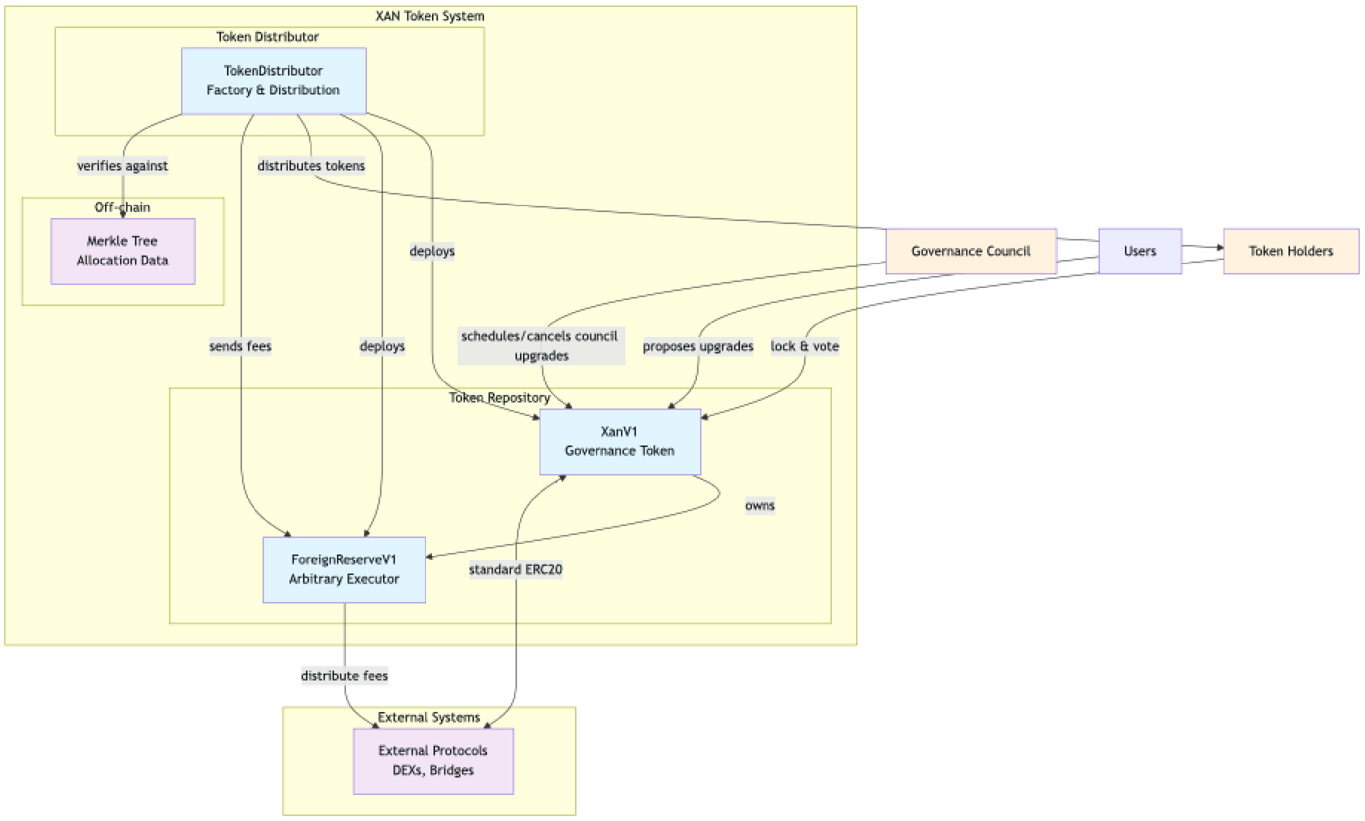

Figure 1: Architecture diagram

Token repository

The `token` repository serves as the governance backbone of the XAN system, housing two critical contracts that work in tandem to provide token management and execution capabilities.

`XanV1.sol` represents a governance token that extends standard ERC-20 functionality. At its core, it implements a dual governance model where both token holders (the "voter body") and a designated governance council can propose and execute contract upgrades. The token features a locking mechanism that is implementation-specific, meaning tokens locked for governance participation in one version of the contract remain locked until that implementation is upgraded. This design ensures that governance participants have long-term alignment with the protocol's success.

The voting system allows locked token holders to cast votes for proposed implementations, with vote weight corresponding to their locked token balance. The contract maintains a quorum requirement of a minimum percentage of locked tokens and minimum locked supply threshold. When these requirements are met, the voter body can schedule upgrades with built-in time delays that provide opportunity for review and potentially cancelling the upgrade.

The governance council operates as a counterbalance to the voter body, with the ability to propose upgrades when the voter body hasn't reached quorum. This creates a system of checks and balances that prevents either governance mechanism from acting unilaterally. The upgrade authorization logic ensures that only legitimately proposed and scheduled upgrades can be executed, with different authorization paths for voter body versus council-initiated upgrades.

`ForeignReserveV1.sol` complements the governance token by providing an arbitrary execution framework. The `ForeignReserveV1` contract serves multiple purposes within the system: it receives fees from the token distribution process, and provides a mechanism to distribute accrued fees to other contracts/EoAs.

Execution flow

Governance participation

1. Token holders lock tokens for current implementation.
2. Locked token holders can vote on proposed implementations.
3. When quorum is reached, upgrades can be scheduled with time delays.
4. Council upgrade can be vetoed if a voter body implementation meets requirements.
5. After delay periods, authorized upgrades can be executed.

Token distributor repository

The `token-distributor` repository implements a token distribution system centered around the `TokenDistributor.sol` contract, which deploys both the `XanV1` and `ForeignReserveV1` contracts during its own construction. During deployment, it initializes the XAN token with an initial supply minted to itself, and establishes the `ForeignReserveV1` with the XAN token as its owner. This creates a clean ownership hierarchy where the governance council can schedule upgrades of the XAN token, which in turn controls the `ForeignReserveV1`.

The distribution mechanism is built around Merkle tree verification. The system supports different allocation structures through the `Allocation` data structure, which includes not only the recipient address and token amounts but also fee requirements and the distinction between locked and unlocked tokens. This allows for distribution strategies where some recipients receive immediately available tokens while others receive tokens that are permanently locked for governance participation. Each allocation can specify a fee that must be paid during claiming, and these fees are automatically forwarded to the `ForeignReserveV1` contract.

The contract enforces configurable start and end times for the claiming period, ensuring that distribution occurs within a specific window. The system tracks claimed allocations using a bitmap approach, preventing double-claiming while minimizing storage costs.

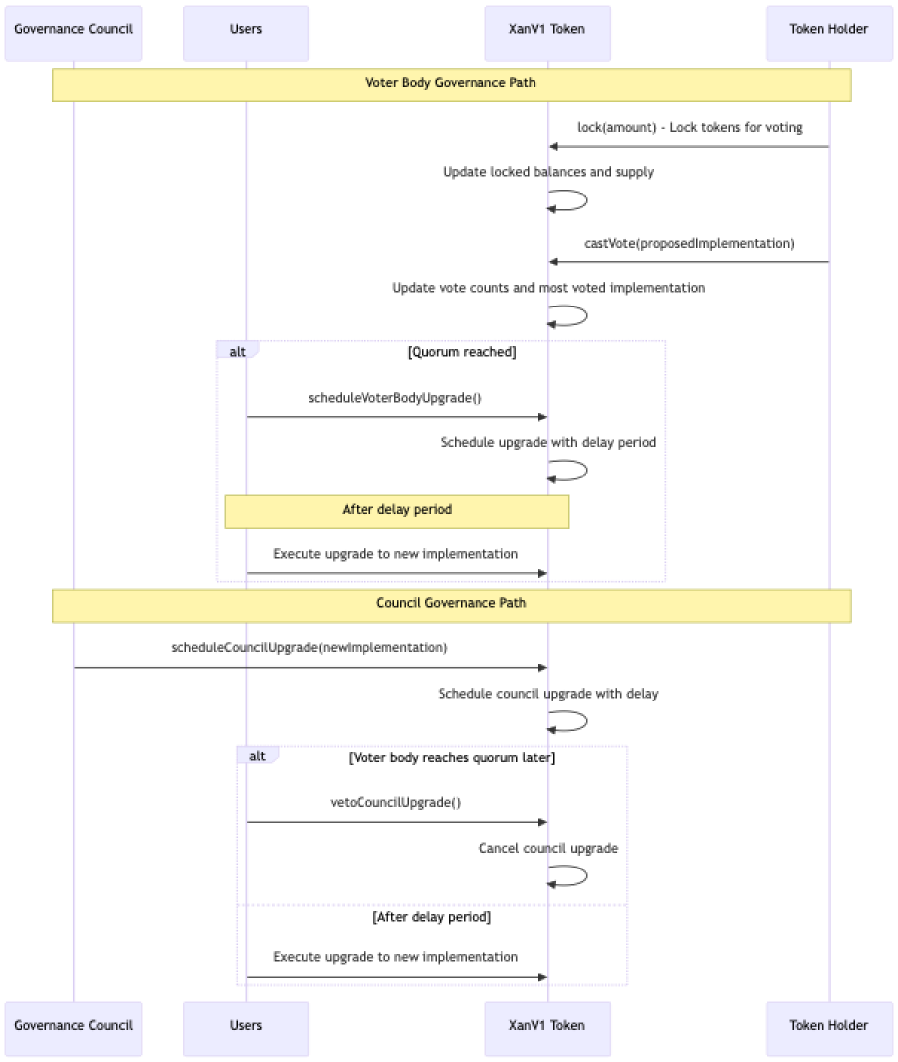
Governance Council
Users
XanV1 Token
Token Holder
Voter Body Governance Path
lock(amount) - Lock tokens for voting
Update locked balances and supply
castVote(proposedImplementation)
Update vote counts and most voted implementation
alt
[Quorum reached]
scheduleVoterBodyUpgrade()
Schedule upgrade with delay period
After delay period
Execute upgrade to new implementation
Council Governance Path
scheduleCouncilUpgrade(newImplementation)
Schedule council upgrade with delay
alt
[Voter body reaches quorum later]
vetoCouncilUpgrade()
Cancel council upgrade
[After delay period]
Execute upgrade to new implementation

Figure 2: Execution flow

The cleanup mechanism ensures that unclaimed tokens don't remain indefinitely in the distribution contract. After the distribution period ends, anyone can trigger the burning of remaining tokens, ensuring that the total supply reflects only the tokens that were actually claimed. This prevents the accumulation of unclaimed tokens that could potentially be exploited or create uncertainty about the true circulating supply (which would distort the requirement of minimum locked supply for voter body upgrades).

When allocations specify locked tokens, the distributor uses the XAN token's `transferAndLock` function to both transfer and immediately lock the tokens in a single operation. This ensures that recipients of locked allocations are immediately eligible to participate in governance, creating a seamless transition from distribution to governance participation.

Execution flow

System deployment:

1. Deploy `TokenDistributor` with Merkle root, time bounds, and governance council address.
2. `TokenDistributor` automatically deploys `XanV1` proxy (mints initial supply to itself).
3. Governance council address becomes council of `XanV1`.
4. `TokenDistributor` automatically deploys `ForeignReserveV1` proxy (owned by XAN token).
5. System is ready for time-bounded claiming period.

Token claiming process:

1. Claimer generates Merkle proof for their allocation off-chain.
2. Claimer calls `claim()` with allocation data, proof, and required fee.
3. `TokenDistributor` verifies time bounds, claim status, and Merkle proof.
4. Fee is forwarded to `ForeignReserveV1` if applicable.
5. Locked tokens are transferred and permanently locked via `XanV1.transferAndLock()`.
6. Unlocked tokens are transferred directly to claimer.
7. Allocation is marked as claimed in bitmap.

Post-distribution operations

1. After end time, anyone can call `burnUnclaimedTokens()`.
2. All remaining tokens in `TokenDistributor` are burned.
3. System transitions to normal governance-controlled ERC20 operations.
4. `ForeignReserveV1` continues to provide arbitrary execution capabilities.

Implementation overview

The XAN token system utilizes OpenZeppelin contracts library as its foundation for security and standardization. The architecture embraces upgradeability throughout, with `XanV1` and `ForeignReserveV1` contracts implemented as a UUPS (Universal Upgradeable Proxy Standard) proxy to enable evolution over time. The token contract extends OpenZeppelin's comprehensive ERC20 suite, incorporating permit functionality for gasless approvals and burnable capabilities for supply management, while the `ForeignReserveV1` contract combines ownership controls with reentrancy protection to safely execute arbitrary operations. The distribution mechanism leverages cryptographic merkle proofs for verification and OpenZeppelin's `SafeERC20` patterns for secure token handling. Custom governance libraries manage the complex dual-governance model, token locking mechanisms, and voting systems, while the ERC-7201 storage pattern ensures upgrade safety by preventing storage collisions.

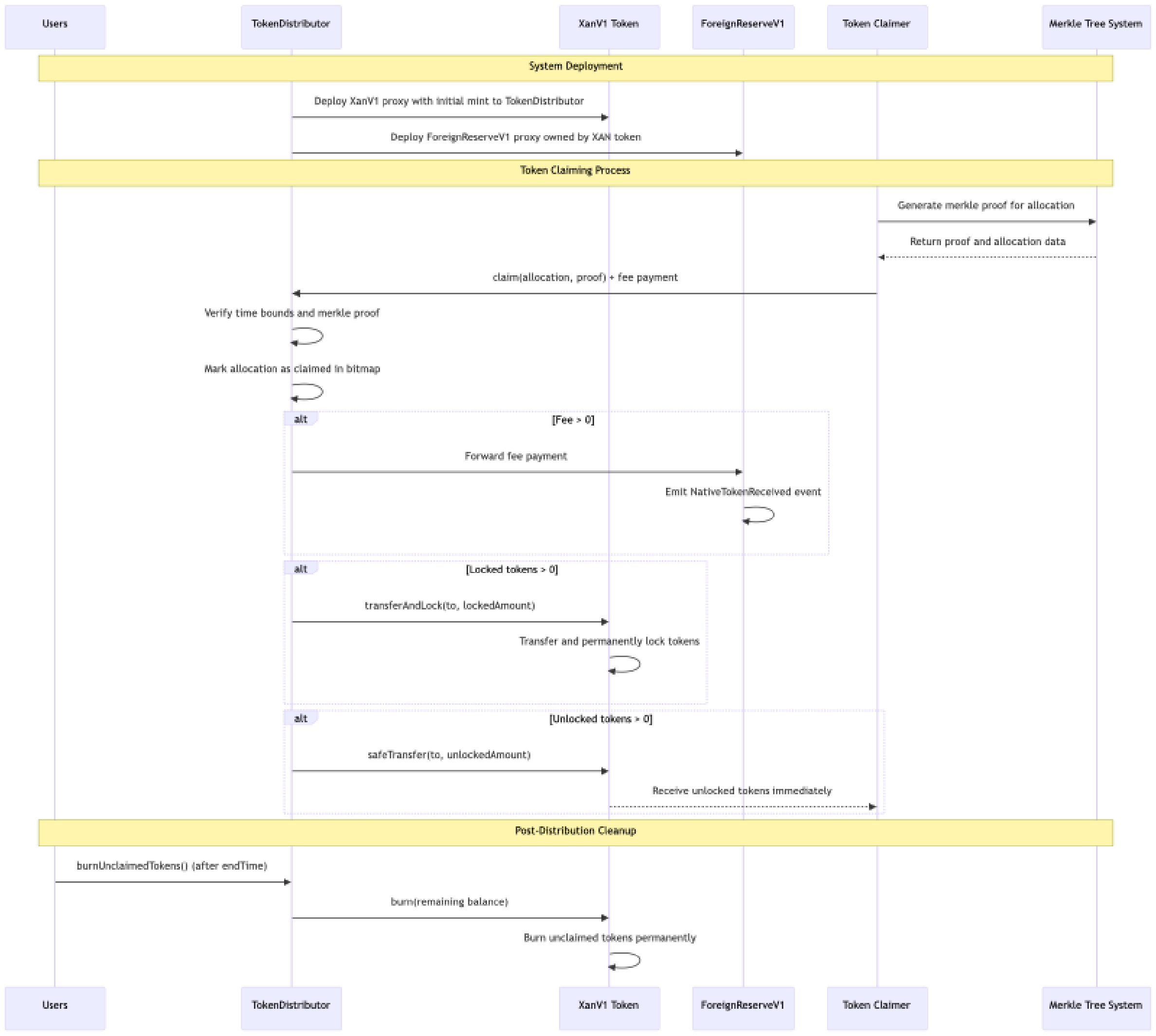
Users
TokenDistributor
XanV1 Token
ForeignReserveV1
Token Claimer
Merkle Tree System
System Deployment
Deploy XanV1 proxy with initial mint to TokenDistributor
Deploy ForeignReserveV1 proxy owned by XAN token
Token Claiming Process
Generate merkle proof for allocation
Return proof and allocation data
claim(allocation, proof) + fee payment
Verify time bounds and merkle proof
Mark allocation as claimed in bitmap
alt [Fee > 0]
Forward fee payment
Emit NativeTokenReceived event
alt [Locked tokens > 0]
transferAndLock(to, lockedAmount)
Transfer and permanently lock tokens
alt [Unlocked tokens > 0]
safeTransfer(to, unlockedAmount)
Receive unlocked tokens immediately
Post-Distribution Cleanup
burnUnclaimedTokens() (after endTime)
burn(remaining balance)
Burn unclaimed tokens permanently

Figure 3: Execution flow

Threat Model

Assumptions

- All inherited OpenZeppelin contracts are properly audited, maintain standard compliance, and have no critical security vulnerabilities.

Protocol properties

Safety properties for `token`

Property PS-1 (locked balance constraints): Locked balances never exceed total balances for any address

- Threat a: Manipulation of lock operations to exceed total balance

  Conclusion: The threat doesn't hold. Lock operations can't push a user's locked balance above their total balance. Locking requires `value <= unlockedBalanceOf(account)`, where `unlocked = balance - locked`. Any attempt to lock more reverts before incrementing locked amounts (code ref↗). Transfers and burns only move unlocked tokens. Any attempt to move more than the unlocked portion reverts. (code ref↗)

Property PS-2 (locked token transfer immutability): Locked tokens cannot be transferred, sold, or moved between addresses while they remain locked

- Threat a: Exploiting vulnerabilities in the `_update()` function or unlocked balance calculation to transfer locked tokens

  Conclusion: The threat doesn't hold. `_update()` is overridden to require `unlockedBalanceOf(from) >= value` for any non-mint move (code ref↗). Locked amounts are excluded from spendable balance. All transfer-/burn paths route through `_update()`.

- Threat b: Manipulation of `implementationSpecificData` mapping to affect locked balance calculations

  Conclusion: The threat doesn't hold. Locks are intentionally scoped per implementation and reset on upgrade by design (different slots under different namespace are used for implementation specific data). External manipulation of the mapping is not possible and upgrades must pass authorization.

- Threat c: Exploiting edge cases in ERC-20 transfer logic to move locked tokens

  Conclusion: The threat doesn't hold. All balance changes (`transfer()`, `transferFrom()`, `_burn()`) go through `_update()` and thus the unlocked-balance guard. `_mint()` bypasses the guard but does not spend locked tokens. Custom `transferAndLock()` transfers then locks at the receiver.

Property PS-3 (quorum enforcement): A voter body upgrade may only be scheduled when quorum (50% + 1 of locked supply) is reached and minimum locked supply (25% of total) is met

- Threat a: Manipulation of vote counting to bypass quorum requirements (e.g., counting same votes multiple times)

  Conclusion: The threat doesn't hold. A voter's contribution to a given proposal's tally is capped at their current locked balance. Repeated calls without increased locks revert, preventing double-count within the same proposal (code ref↗).

- Threat b: Exploiting locked supply calculation errors to underreport total locked supply (and thus lowering requirements threshold)

  Conclusion: The threat doesn't hold. Locked supply is tracked centrally per implementation and only increases via `_lock()`; the quorum threshold derives directly from this value↗ and scheduling enforces both threshold and minimum locked supply.

Property PS-4 (vote monotonicity): Individual vote counts can only increase (never decrease) for any voter-implementation pair

- Threat a: Vote update logic is exploited to decrease votes

  Conclusion: The threat doesn't hold. `castVote()` forces strictly increasing per-voter votes for a given implementation by requiring `newVotes > oldVotes` and setting votes to `lockedBalance`, then only adding the positive delta to the total locked supply (code ref↗). There is no path to reduce either the per-voter votes or total votes. `lockedBalance` never decreases (no unlock; transfers/burns can only move unlocked), so new votes cannot be less than prior votes. Attempts to recast without increased locked amount revert.

Property PS-5 (upgrade mutual exclusion): At most one upgrade can be scheduled at any time (either voter body or council, never both)

- Threat a: Race condition in upgrade scheduling logic causes concurrent upgrade

  Conclusion: The threat doesn't hold. Voter's body scheduling cancels any existing council schedule (code ref↗). Council scheduling is forbidden when voter quorum is (or remains) reachable (code ref↗) and upgrade authorization asserts that both cannot be scheduled simultaneously (code ref↗).

- Threat b: State inconsistency between voter body and council upgrade tracking

  Conclusion: The threat doesn't hold. Both subsystems use the same scheduling predicate `(impl != 0 and endTime != 0)`, voter's body scheduling clears any existing council schedule and upgrade authorization clears the corresponding schedule after success.

Property PS-6 (voter body upgrade cancellation authority): A voter body upgrade may only be cancelled when (a) the delay period has elapsed and (b) the scheduled upgrade no longer meets quorum requirements or is no longer the most voted

- Threat a: Cancelling legitimate voter body upgrades before delay period elapsed or while still meeting requirements

  Conclusion: The threat doesn't hold. Cancellation requires the delay to have ended, and it reverts if the scheduled implementation still meets quorum and remains the most-voted (code ref↗).

- Threat b: Preventing cancellation of voter body upgrades that no longer meet requirements

  Conclusion: The threat doesn't hold. Once the delay has elapsed, if the scheduled implementation either loses quorum/minimum locked supply or stops being the most-voted, cancellation proceeds and clears the schedule.

Property PS-7 (council upgrade cancellation authority): A council upgrade may be cancelled either (a) only by the council if the council upgrade is scheduled and has not executed yet, or (b) automatically when a voter body upgrade becomes scheduled

- Threat a: Non-council entities cancelling scheduled council upgrades using `cancelCouncilUpgrade` function

  Conclusion: The threat doesn't hold. `cancelCouncilUpgrade` is restricted by `onlyCouncil`, which reverts for any caller not equal to `governanceCouncil()` (code ref1↗, ref2↗).

- Threat b: Preventing council from cancelling its own scheduled upgrades

  Conclusion: The threat doesn't hold. When a council upgrade is scheduled, the council can cancel it at any time before execution. The function only checks scheduling status and emits cancellation event, then clears the schedule.

- Threat c: Scheduling voter body upgrades without properly cancelling a scheduled council upgrade

  Conclusion: The threat doesn't hold. Scheduling a voter body upgrade automatically clears any existing council schedule (code ref↗).

Property PS-8 (council upgrade veto authority): A council upgrade may be vetoed if the council upgrade is scheduled and an alternative implementation meets quorum and minimum locked supply requirements

- Threat a: Vetoing council upgrades without sufficient voter body consensus

  Conclusion: The threat doesn't hold. Veto requires a council upgrade to be scheduled and the voter body's most-voted implementation to satisfy quorum and minimum locked supply requirements, otherwise it reverts (code ref↗).

- Threat b: Preventing legitimate voter body veto of council upgrades

  Conclusion: The threat doesn't hold. Anyone can call `vetoCouncilUpgrade()` and it succeeds exactly when a council upgrade is scheduled and the voter body's most-voted proposal meets quorum and minimum locked supply requirements, after which the council schedule is cleared.

- Threat c: Manipulation of quorum calculations to enable unauthorized vetos

  Conclusion: The threat doesn't hold. Quorum requires 50% of current locked supply plus one, and also requires a minimum locked supply. Both derive from single sources of truth (locked supply, total votes), preventing threshold underreporting.

Property PS-9 (upgrade cancellation token persistence): Cancelling a scheduled voter body upgrade does not unlock any tokens, reset vote counts, or modify locked balances

- Threat a: Unauthorized token unlocking via cancellation exploitation

- Threat b: Vote history manipulation through cancellation operations

- Threat c: Locked supply manipulation during cancellation

  Conclusion: The threat's a, b, and c don't hold. `cancelVoterBodyUpgrade()` only checks schedule existence, enforces the delay, validates that the scheduled implementation no longer meets requirements, emits an event, and clears scheduling fields. It does not touch locked balances, total locked supply, or any vote tallies.

Property PS-10 (only authorized upgrade execution): `execute(upgrade_u)` implies `u` is a proposal that (a) was validly scheduled and (b) its schedule has matured (`now >= eta`) under the correct track (voter or council)

- Threat a: Executing upgrades without proper quorum and minimum locked supply

- Threat b: Delay period bypass through timestamp manipulation

- Threat c: Executing upgrades for implementations that are no longer most voted

- Threat d: Executing council upgrades despite voter body meeting quorum and minimum locked supply requirements

- Threat e: Manipulation of voter body vote counts to enable council execution

Conclusion: The threats a to e don't hold. Execution is only allowed if exactly one track scheduled the upgrade, the delay matured, and the correct per-track conditions hold. Voter body's track requires still-most-voted and quorum/minimum locked supply. Council track is blocked if the voter track meets quorum/minimum locked supply.

Liveness properties for `token`

Property PL-1 (upgrade schedulability): If a voter body implementation meets quorum and minimum locked supply requirements, an upgrade can eventually be scheduled

- Threat a: Conditions are created (e.g. deadlock) where no upgrade can be scheduled

  Conclusion: The threat doesn't hold. Scheduling is permissionless and succeeds whenever the most-voted proposal meets quorum and minimum locked supply. Any existing council schedule is auto-canceled, preventing deadlock, also in order to schedule new voter's body upgrade existing upgrade has to be canceled.

Property PL-2 (vote castability): Token holders with locked tokens may always cast votes for implementations

- Threat a: Voters are prevented from casting votes

  Conclusion: The threat doesn't hold. Voting is permissionless. Any holder with at least 1 locked token can vote for a implementation, since the first cast requires `lockedBalanceOf(voter) >= 1` and only reverts if trying to restate the same or fewer votes on that proposal. Locking is also permissionless, enabling any holder to obtain voting power.

Property PL-3 (token transferability): Unlocked tokens may always be transferred between addresses

- Threat a: Conditions are created that prevent token transfers

  Conclusion: The threat doesn't hold. Transfers route through `_transfer()` → `_update()` , which only reverts when sending from/to zero address or when `value > unlockedBalanceOf(from)` . There are no additional gates, so any positive unlocked balance is freely transferable.

Property PL-4 (lock functionality): Token holders may always lock their unlocked tokens for governance participation

- Threat a: Making lock operations fail or unavailable

  Conclusion: The threat doesn't hold. Locking is permissionless and always available. Holders can call `lock(value)` anytime, which only reverts if value exceeds their current unlocked balance.

- Threat b: Contract state corruption preventing lock operations

  Conclusion: The threat doesn't hold in most cases except in case described in finding "Upgrading to a previous version re-activates per-implementation locking, voting… states causing potential underflows and balances/-governance inconsistencies". The lock state is maintained in a dedicated per-version namespaced storage and updated only by `_lock()` . No other function mutates `lockedBalances` or `lockedSupply`.

Property PL-5 (vote weight progression): Users may increase their voting power by locking more tokens

- Threat a: Conditions are created that prevent users from increasing their locked token amounts

- Threat b: Lock operation failures when users try to increase participation

  Conclusion. Threat a and b don't hold. Locking is permissionless and only requires `value <= unlockedBalanceOf(account)` . Users can call `lock(value)` anytime to increase `lockedBalances` and `lockedSupply`. After increasing locks, `castVote()` accepts an update only if the voter's new locked balance exceeds their prior recorded votes, ensuring vote weight can be raised.

Safety properties for `token-distributor`

Property SP-15 (double claiming prevention): No allocation may be claimed more than once

- Threat a: Attacker attempts to reset claimed bits in `_claimedBitMap`

  Conclusion: The threat doesn't hold. The `_claimedBitMap` variable is `internal`, therefore an external contract or EOA cannot directly modify the mapping. However, derived contracts have full access on the mapping, and this introduces a potential risk. We have documented a improvement in finding "Miscellaneous code improvements" to change the visibility to `private` and control access to the mapping by derived contracts through internal functions.

- Threat b: Multiple allocations share the same index

  Conclusion: The threat doesn't hold. If multiple allocations had the same index, only the first allocation to be claimed would succeed. The index is used to set a specific bit in `claimedBitMap`↗; if another allocation has the same index, the check done by calling `isClaimed`↗ would fail.

  For any allocation index `n`, we can express it uniquely as `n = 256 × q + r` where:

  ```
  - q = floor(n ÷ 256) (quotient, becomes claimedWordIndex)
  - r = n % 256 (remainder, becomes claimedBitIndex)
  - 0 <= r < 256
  ```

  By the Division Algorithm from number theory: for every integer `n >= 0`, there exists a unique pair `(q, r)`. The mapping function `f(index) = (floor(index ÷ 256), index % 256)` is therefore injective (one-to-one) and no two different allocation indices will ever write to the same key in `_claimedBitMap` and target the same bit position in the value associated with that key.

- Threat c: Claim function is re-entered before updating the bitmap

  Conclusion: The threat doesn't hold. The `_claimedBitMap` is updated↗ before any external calls (and thus follows the checks-effects-interactions pattern for this state change). However, the `claim()` function is still vulnerable to re-entrancy: calls to `_XAN.transferAndLock()` and `_XAN.safeTransfer()` could potentially trigger reentrancy, and also the call to `_FOREIGN_RESERVE` could re-enter from its `receive()` function. Even though the implementation of the contracts pointed to by `_XAN` and `_FOREIGN_RESERVE` are not arbitrary, we have documented a code improvement in finding "Miscellaneous code improvements" to add a reentrancy guard to the `claim()` function (e.g., OpenZeppelin's `ReentrancyGuard` modifier).

- Threat d: Bitmap overflow or underflow causing incorrect claimed status

  Conclusion: The threat doesn't hold. The value of `claimedBitIndex` is always between 0 and 255 and therefore the shift operation `(1 << claimedBitIndex)` will produce a value in the range of the `uint256` type.

Property SP-16 (Merkle proof integrity): Only valid allocations with correct proofs may be claimed

- Threat a: Successful verification of fake proofs for unauthorized allocations

  Conclusion: The threat doesn't hold. Assuming the Merkle tree of allocations is correctly constructed, the contract uses `keccak256` hashing with structured data encoding (`abi.encode(allocation)`) to create leaf hashes. This includes the allocation index, recipient address, fee amount, and token quantities, making it computationally infeasible to find different allocation data that produces the same hash due to `keccak256`'s collision resistance properties. The `_verifyProof()` function dynamically computes the allocation hash during verification using `allocation.hash()`, then validates that the provided proof path reconstructs exactly to the stored `_ALLOCATION_TREE_ROOT`. Any tampering with allocation data will result in a different leaf hash, causing proof verification to fail. Since `_verifyProof()` computes the allocation hash, it's not possible either for an attacker to provide as a leaf a hash value of an intermediary node for which a Merkle proof would be accepted as valid.

- Threat b: Modification of the stored Merkle root

  Conclusion: The threat doesn't hold. The `_ALLOCATION_TREE_ROOT` variable is `immutable`: once set in the `constructor()` it is not possible to modify it.

- Threat c: Proof replay attacks using valid proofs for wrong allocations

  Conclusion: The threat doesn't hold. All proofs must reconstruct to the immutable `_ALLOCATION_TREE_ROOT` that was set during contract deployment. This root represents the cryptographic commitment to the entire allocation set, and any unauthorized allocation would require breaking the hash function's one-way property.

Property SP-17 (time window enforcement): Claims may only be made within the designated time window

- Threat a: Conditions are created to claim successfully outside of the designated window

  Conclusion: The threat doesn't hold. The `claim()` function checks that the block timestamp `t` lies in the range `_START_TIME <= t < _END_TIME`. Validators cannot manipulate the block timestamp, as in Ethereum's Proof-of-Stake consensus, time is divided into fixed 12-second slots, and each slot has exactly one valid timestamp: `genesis_time + slot × 12`. When a validator proposes a block, the protocol requires the block's execution payload timestamp to equal this predetermined slot time; any deviation makes the block invalid and rejected by the network. This enforcement means that block timestamps always advance in strict 12-second multiples from genesis, regardless of when the block is actually propagated. If a validator misses their slot, no block is created, but the next valid block must still align to the next slot's timestamp, ensuring the chain's clock ticks forward in precise 12-second increments.

Property SP-18 (fee integrity): Exact fee amount must be paid for each claim

- Threat a: Successful claim paying less than required fee

  Conclusion: The threat doesn't hold. The `claim()` function guarantees that claims are only processed if the exact fee amount specified in the allocation is sent with the function call↗.

Property SP-19 (token supply conservation): Total distributed tokens cannot exceed the initial supply.

- Threat a: Exploiting XAN token contract to mint additional tokens

  Conclusion: The threat doesn't hold. The contract `XanV1.sol` doesn't expose functionality to mint additional tokens after the initial minting done in the function `initializeV1()`↗.

Liveness properties for `token-distributor`

Property LP-6 (claim availability): Valid claims should be processable during the claim window

- Threat a: Claim transactions fail due to gas limit issues

  Conclusion: The threat doesn't hold with the current implementation, but it might hold in the future. The `claim()` function makes external call to the `_FOREIGN_RESERVE` contracts. The foreign fee transfer call forwards all remaining gas (~63/64 of available gas) to the foreign reserve contract. Even though `ForeignReserveV1` implementation's of `receive()` is correct at the moment, if the foreign reserve is upgraded via proxy to include gas-consuming logic in its `receive()` function, the entire claim transaction would revert. As a consequence, users would pay gas fees but receive no tokens, with their allocation remaining un-claimable. Since the development team does not plan to upgrade the `ForeignReserveV1` contract before the claim window has ended, there is no risk at the moment with the current implementation of the `receive()` function.

- Threat b: Claim transactions fail because calls to foreign reserve contract or XAN contract fail

  Conclusion: The threat doesn't hold with the current implementation, but it might hold in the future. As mentioned in Threat LP-6.a, the foreign fee transfer might fail if the implementation of the `receive()` function is buggy.

This is not the case for the current version, and it's unlikely for future versions. With respect to the calls on the `_XAN` contract, the current implementation of `transferAndLock()` is correct and it will succeed for any non-zero receiver address as long as the contract has sufficient funds (same conditions apply to `safeTransfer()`). Since the development team does not plan to upgrade the `ForeignReserveV1` and `XanV1` contracts before the claim window has ended, there is no risk at the moment with the current implementations.

Property LP-7 (fee collection): Claim fees should be successfully transferred to the foreign reserve

- Threat a: Call to foreign reserve contract fails (e.g., because of a bug or gas limit issues)

  Conclusion: The threat doesn't hold with the current implementation, but it might hold in the future. As explained in Threat LP-6.a, the external call to the `_FOREIGN_RESERVE` maybe fail if the `receive()` function consumes all available gas or contains a bug. At the moment, the implementation is honest and correct, but if the contract is upgrade via proxy, the development team needs to be careful and make sure that, if the claim window is still active, the new version of the contract should not fail when receiving the fees. Since the development team does not plan to upgrade the `ForeignReserveV1` contract before the claim window has ended, there is no risk at the moment with the current implementation.

Property LP-8 (token distribution): Claimed tokens should be properly distributed to beneficiaries

- Threat a: Failure in XAN contract token transfer (e.g., insufficient balance, transfer restrictions, gas limit, etc)

  Conclusion: The threat doesn't hold. For any non-zero receiver address, the most likely failure that can occur in the calls `_XAN.transferAndLock()` and `_XAN.safeTransfer()` is that the sender has insufficient balance. The generation of the allocations is out of the scope of this audit, but the development team has informed us that they will have checks in their deployment scripts to make sure that the sum of all allocations does not exceed the initial minted supply received by the `_XAN` contract.

- Threat b: Failure in XAN contract token locking mechanism

  Conclusion: The threat doesn't hold. As long as the `_XAN` contract has sufficient balance, it should be possible to transfer tokens to the recipients.

Property LP-9 (cleanup completion): Unclaimed tokens should be burnable after the claim period.

- Threat a: XAN contract token burn function malfunctions or reverts

  Conclusion: The threat doesn't hold. The call `_XAN.burn()` is unlikely to fail (the balance hold by `_XAN` is passed as argument, so it's not possible that it is attempted to burn more tokens than available).

- Threat b: Tokens can be burn before the claim period ends

  Conclusion: The threat holds. The `burnUnclaimedTokens()` function checks that the block timestamp is >= `_END_TIME`↗. As mentioned in Threat SP-17.a, validators cannot manipulate the block timestamps and they always advance in strict 12-second multiples from genesis.

Integration properties

For `token`

Property IS-1 (proxy upgrade authorization): Only the authorized upgrade mechanism can change the implementation contract

- Threat a: Exploiting UUPS upgrade mechanism without proper authorization

  Conclusion: The threat doesn't hold. Upgrades must be invoked through the proxy and call `_authorizeUpgrade()`. `XanV1` overrides this function to require exactly one valid schedule (voter or council), enforce quorum/minimum locked supply (voter's body track) and require delay maturity.

- Threat b: Implementation contract replacement without proper validation

  Conclusion: The threat doesn't hold. The UUPS upgrade routine validates the new implementation via ERC-1822 `proxiableUUID` against the ERC-1967 implementation slot and reverts if incompatible. `XanV1` also rejects zero address and enforces scheduling rules.

Property IS-2 (storage layout compatibility): Upgrades maintain storage layout compatibility using ERC-7201 standard

- Threat a: Storage layout incompatibility between different implementation versions
- Threat b: Incorrect ERC-7201 namespace calculation causing data corruption
- Threat c: Malicious storage layout changes to access unauthorized data

  Conclusion: The threats a, b and c don't hold. Storage uses ERC-7201 namespaced slots: each version may define its own slot, while V1's slot (`_XAN_V1_STORAGE_LOCATION`) holds the governance state and isolates per-implementation data via `implementationSpecificData[implementation()]`, preventing layout collisions across versions/implementations.

Property IS-3 (upgrade-induced governance state reset): Upon implementation upgrade, the new implementation is isolated from the previous implementation's governance state. Although locked balances, locked supply, voter body data, and council data from the previous implementation remain in storage, the new implementation cannot access them and must start with fresh governance state (zero locked balances/supply and empty voter/council data)

- Threat a: New implementation still accesses locked balances an locked supply from previous implementation
- Threat b: New implementation still accesses voter body and council data from previous implementation

  Conclusion: While the property cannot be technically guaranteed due to the inherent flexibility of upgradeable contracts, the combination of social governance, best practices, monitoring systems, and the mandatory delay period creates a robust defense-in-depth strategy. The 2-week delay period is particularly crucial as it provides the community with adequate time to identify and respond to potentially harmful upgrades, ensuring that governance state isolation remains a practical reality even without technical enforcement.

  This approach balances the need for upgrade flexibility with security considerations, acknowledging that some security properties must be maintained through governance processes rather than purely technical constraints.

Property IS-4 (timestamp dependency security): Assuming that normal behaviour of Ethereum validators, the way timestamps are used is secure

- Threat a: Manipulation of block timestamps to bypass delay periods (e.g., execute upgrades prematurely) or postpone delay period expiration

  Conclusion: Threat doesn't hold. The function `_checkDelayCriterion()` checks that the block timestamp is < `endTime`, and as mentioned in Threat SP-17.a, validators cannot manipulate block timestamps.

Property IS-5 (upgrade executability): Properly scheduled upgrades can eventually be executed through the proxy system

- Threat a: Failures preventing execution of scheduled upgrades

  Conclusion: The threat doesn't hold. There are no known failures that could prevent execution of scheduled upgrade. Execution is permitted only when exactly one track (council or voter's body) has a valid scheduled upgrade and the delay has matured. Upon success, the schedule is cleared, ensuring progress.

- Threat b: State corruption preventing upgrade completion

  Conclusion: The threat doesn't hold. The UUPS routine validates the new implementation via ERC-1822 `proxiableUUID` against the ERC-1967 implementation slot, and governance state uses a fixed ERC-7201 slot. Both reduce the risk of corrupted state blocking execution.

For `token-distributor`

Property IS-6 (XAN token contract integration): XAN token contract must maintain expected behavior

- Threat a: XAN contract upgrade breaking interface compatibility

  Conclusion: `TokenDistributor` calls (via the proxy) functions `transferAndLock()` and `burn()` from `XanV1.sol`. The threat might hold if 1) the `XanV1` contract is upgraded before the claim period is over (which is **not** the plan of the development team), and 2) the upgraded contract doesn't inherit from `XanV1` or it doesn't implement the expected functions. Therefore, please make sure to have implementations of both functions available on new XAN contract implementations, as long as claiming is still possible or unclaimed tokens need to still be burned.

Property IS-7 (foreign reserve contract integration): Foreign reserve contract must accept and handle fee payments correctly

- Threat a: Foreign reserve contract rejecting ETH payments

  Conclusion: The threat doesn't hold. The contract `ForeignReserveV1.sol` properly implements the `receive()` function↗, which is called (via the proxy) from `TokenDistributor` with empty calldata and a value > 0. However, if the contract is upgraded before the claim period is over, please make sure that the new version of the contract either inherits `ForeignReserveV1` or it also correctly implements the `receive()` function.

Implementation properties

Property IMP-1 (integer overflow protection): All arithmetic operations are protected against integer overflow and underflow

Conclusion: The property holds.

Property IMP-2 (division by zero protection): All division operations are protected against division by zero

Conclusion: The property holds. There's only one division↗ and the denominator is a non-zero constant.

Property IMP-3 (function visibility): All functions are marked with appropriate visibility (no unnecessary exposure of internal functions)

Conclusion: The property holds.

Property IMP-4 (modifier correctness): Access control modifiers correctly implement authorization logic and are applied to all necessary functions

Conclusion: The property holds.

Property IMP-5 (state variable protection): Critical state variables are properly protected against unauthorized modification

Conclusion: The property holds.

Property IMP-6 (CEI pattern): Checks-Effects-Interactions pattern is applied consistently

Conclusion: The property holds.

Property IMP-7 (reentrancy protection): All external calls are protected against reentrancy attacks

Conclusion: The property doesn't hold. The function `claim()` of `TokenDistributor` makes external calls, but given that the `_FOREIGN_RESERVE` and `_XAN` contracts will not be updated before the token claim period is over, then there is no risk in not protecting against reentrancy.

Property IMP-8 (gas limit protection): All operations are designed to complete within reasonable gas limits (e.g. no unbounded loops)

Conclusion: The property holds.

Property IMP-9 (event emission): Events are emitted for all state changes

Conclusion: The property holds.

Findings

Finding	Type	Severity	Status
Minimum locked supply requirement may be impossible to satisfy if more than 75% of total supply is burned	Protocol	Medium	Risk Accepted
Upgrading to a previous version re-activates per-implementation locking, voting... states causing potential underflows and balances/governance inconsistencies	Implementation	Informational	Risk Accepted
Claim window can be as short as 1 second	Implementation	Informational	Risk Accepted
Miscellaneous code improvements	Implementation	Informational	Acknowledged

Minimum locked supply requirement may be impossible to satisfy if more than 75% of total supply is burned

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Protocol **Status** Risk Accepted

Involved artifacts

- `token:src/XanV1.sol`↗
- `token-distributor:src/TokenDistributor.sol`↗

Description

When the claim period is over, anyone can burn the remaining tokens in supply by calling `burnUnclaimedTokens()`↗. If more than 75% of the supply is burned, then the minimum locked supply requirement of 1/4 of supply↗ would be impossible to satisfy, since the supply value is the initial minted amount↗, not the actual circulating supply.

Problem scenarios

If it's impossible to meet the minimum locked supply requirement, then no voter body implementation could be scheduled.

The development team's original intention was to have a fixed minimum that would not depend on how many tokens are eventually burned, but they acknowledge that this is an important consideration to keep in mind to guarantee a safe deployment (i.e., that at least 1/4 of tokens must be claimed).

Recommendation

Consider using ERC20's `totalSupply()`↗ in `MIN_LOCKED_SUPPLY` calculation.

Alternatively, the `claim()` function could allow claiming until at least 25% of supply has been claimed (even if the claim period has elapsed). And additionally, the `burnUnclaimedTokens()` function could also be adjusted to allow burning only if the period is over and at least 25% of the supply has been claimed.

Mitigation

Team acknowledges this issue and plans to mitigate the risk by deploying the contracts and orchestrating `claim()` calls on behalf of >50% of total supply.

If any issue arises during this bootstrap phase, the contract can be redeployed and the bootstrap repeated.

Upgrading to a previous version re-activates per-implementation locking, voting… states causing potential underflows and balances/governance inconsistencies

Severity Informational Impact: Exploitability:

Type Implementation **Status** Risk Accepted

Involved artifacts

- `token:src/XanV1.sol`↗

Description

Per-implementation governance state (locks, votes, council) is stored under a single ERC-7201 slot in `XanV1` and isolated via a mapping keyed by the current `implementation()` address; upgrading to a previous implementation re-selects that mapping key and re-activates its historic state.

ERC-20 balances persist across implementations, but lock state is per-implementation. After upgrading to a previous implementation, revived `lockedBalances` may exceed current ERC-20 balances; `unlockedBalanceOf = balanceOf - lockedBalance` can underflow and revert, breaking transfers and any functionality that relies on unlocked balance checks.

Voting or proposing previous implementations is not restricted, so older vote/lock totals can be revived even when they no longer reflect current ERC-20 balances, impeding further governance progress.

Problem Scenarios

Unlocked balance underflow**:**

- On V1: A has 100 XAN, locks 50 (keeps 50 unlocked).
- Upgrade to V2: per-implementation locks reset; A has 100 unlocked and transfers 70 → A's balance is 30 now.
- Upgrade to previous implementation (V1): A's balance remains 30, but V1's old lock state revives (50 locked tokens).
- On V1: `unlockedBalanceOf(A) = 30 - 50` underflows and reverts; operations depending on unlocked balances (e.g., transfers) fail, and governance state is misaligned with the live ERC-20 balances.

Mitigation

The client acknowledges the issue. While in-code mitigations could be possible (e.g., a more complex proxy pattern or blacklisting prior implementation addresses), those approaches introduce additional complexity and risk. The client therefore plans to treat downgrade avoidance primarily as an operational control (update the README and governance playbooks to forbid downgrades, and rely on monitoring and emergency remediation) using any low-risk code mitigations only as complements.

Claim window can be as short as 1 second

Severity Informational | Impact: | Exploitability:

Type Implementation | **Status** Risk Accepted

Involved artifacts

- `token-distributor:src/TokenDistributor.sol`↗

Description

The `TokenDistributor` constructor↗ allows claim windows of 1 second at a minimum.

Problem scenarios

Such short claim windows would make it impossible in practice to claim the tokens.

However, the development team does not consider this to be a problem, as they can redeploy the contract in case the values for `startTime` or `endTime` that would allow such short window are used in the first place.

Recommendation

Consider implementing a minimum, long enough claim window.

Miscellaneous code improvements

Severity Informational Impact: Exploitability:

Type Implementation **Status** Acknowledged

`token`

- `src/ForeignReserveV1.sol`:
  - Function `execute()` is decorated with the `nonReentrant` modifier, but it is not the first modifier applied. There is a risk that other modifiers, such as `onlyOwner`, could execute code that might inadvertently allow reentrancy vulnerabilities to be exploited before the `nonReentrant` check is enforced. To mitigate this risk, the `nonReentrant` modifier should be placed before all other modifiers to ensure that the reentrancy protection is applied as early as possible in the function execution.

`token-distributor`

- `src/TokenDistributor.sol`:
  - The variable `_claimedBitMap` ↗ is `internal` but opens the possibility for derived contracts to modify as they wish the claimed status of allocations. A better approach would be to make `_claimedBitMap` private and provide controlled access through internal functions (`_isClaimed`, `_setClaimed`, etc).
  - The constant value 256 is used multiple times in functions `_setClaimed()` and `_isClaimed()` ↗, thus consider creating a constant state variable and reference it throughout the contract.

Appendix: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1↗, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score↗, and the Exploitability score↗. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale↗, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)

ImpactScore	Examples
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

ExploitabilityScore	Examples
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

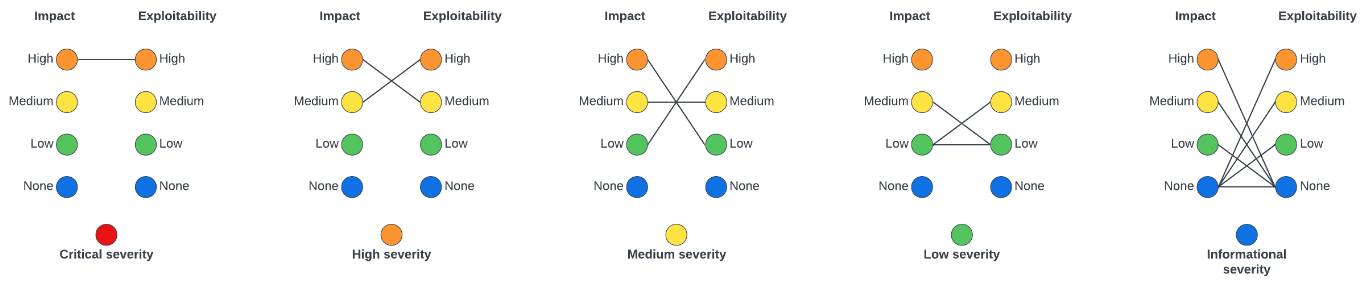

Figure 4: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

SeverityScore	Examples
Critical	Halting of chain via a submission of a specially crafted transaction
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer