

Security Audit & Remediation

Security · Published by UnyKorn LLC (Wyoming), technology provider · September 2026

DEMONSTRATION DOCUMENT — NOT AN OFFERING, NOT A BOUND CONTRACT. All entities, identifiers, reserves, serials, vaults, policies, counterparties and figures in this document are illustrative and synthetic. UnyKorn LLC is not a broker-dealer, exchange, alternative trading system, investment adviser, bank, trust company, qualified custodian, money transmitter, insurer or issuer, and does not hold client assets or funds. No entity described here is registered with the SEC, FINRA, the FCA or any other regulator. No refinery, vault operator, insurer, broker, auditor or qualified person named or implied is a counterparty to UnyKorn LLC. Nothing here is an offer to sell or a solicitation of an offer to buy any security, or investment, legal, tax or accounting advice.

DGX Permissioned Issuance Layer — Security Audit

Scope: `contracts/src/` — 14 Solidity files, Solidity 0.8.20, optimizer 200 runs. **Method:** adversarial review against the stated invariants, plus an executable invariant suite run on a real EVM (`contracts/test/trex_invariants.py`).
Date: September 2026.

Remediation status

Every Critical and High finding has been remediated and carries a regression test that **failed before the fix**. The suite now runs **54 assertions, 54 passing**, of which 14 are audit regressions.

ID	Severity	Finding	Status
C-1	Critical	Reserve cap enforced against a private counter, not live supply — a routine compliance migration silently reset it to zero	Fixed. The module holds an immutable token reference and derives headroom from <code>totalSupply()</code> . The counter and its bookkeeping are deleted, so there is nothing left to desync.
C-2	Critical	Unbounded future <code>asOf</code> — one attesor could permanently disable the staleness halt and lock out every later attestation	Fixed. <code>asOf > block.timestamp</code> refused; freshness computed as <code>block.timestamp - asOf</code> so it cannot overflow; empty evidence hash refused; governance ceiling added.
C-3	Critical	A claim's own topic was never compared to the bucket it came from — one genuine KYC claim satisfied AML, accreditation and country	Fixed. <code>if (claimTopic != topic) continue;</code> . Registration now also requires the identity to prove it holds a management key for the wallet.
H-1	High	Zero-value transfers restarted the lockup clock — anyone could lock any holder forever	Fixed. Zero-value transfers revert at the token; the module ignores zero amounts; an administrative <code>resetLockup</code> was added.
H-2	High	<code>forcedTransfer</code> bypassed every compliance module — one agent key breached the holder cap, country list and all lockups	Fixed. <code>forcedTransfer</code> now runs <code>canTransfer</code> . Genuine court orders use <code>forcedTransferOverride</code> , gated on a separate recovery-agent role and emitting a distinguishable <code>ComplianceBypassed</code> event.
H-3	High	Reentrancy through <code>compliance.created</code> could double-spend reserve headroom	Fixed. <code>nonReentrant</code> on every value-moving entry point, and the C-1 fix rejects the reentrant mint by construction.
H-4	High	<code>try/catch</code> stopped a reverting claim issuer but not a gas-burning one; the claim loop was unbounded	Fixed. The loop is bounded at 16 claims per topic and the issuer is called through a <code>staticcall</code> with a fixed gas stipend.
H-5	High	Nine contracts had permanently non-transferable ownership	Fixed. Two-step <code>transferOwnership</code> / <code>acceptOwnership</code> on all of them, so admin can move to a Safe or timelock.
H-6	High	Burning re-opened mint headroom before the attestation was refreshed	Fixed by C-1: headroom is attestation minus live supply, so a burn only returns headroom that was already backed.
H-7	High	Module-to-compliance binding was one-shot, making a mis-bind a total DoS	Fixed. The cap module's compliance binding is owner-repointable.
H-8	High	<code>AtomicSettlementDvP</code> performs no settlement	Deprecated. Superseded by the T-REX stack; marked DEPRECATED in-file and removed from the issuance path.
H-9	High	<code>DGXBullionToken</code> : unlimited mint per bar, oracle never consulted, sender never whitelist-checked	Deprecated. Same — superseded and marked.

Medium, Low and Informational findings are recorded in full below and tracked for the next pass; none of them are on the issuance path's critical invariants.

Standing limitation

This is an internal adversarial audit, not an independent third-party engagement. These contracts are a reference architecture for a demonstration environment and have not held custody of value. An allocator should require an independent audit before any live issuance.

Original findings

Summary

Severity	Count
Critical	3
High	9
Medium	13
Low	8
Informational	6
Total	39

#	Severity	Contract · Function	Title
C-1	Critical	ReserveCapModule · issuedUnits	Reserve cap tracks a private counter, never token.totalSupply() — routine migration silently resets it to zero
C-2	Critical	ReserveAttestationRegistry · publish	Unbounded future asOf — one attester permanently disables the staleness halt and locks out all future attestations
C-3	Critical	DGXIdentityRegistry · _topicSatisfied	Claim's own topic never compared to the bucket topic — one KYC claim satisfies AML, accreditation and country
H-1	High	LockupModule · moduleTransferAction	Zero-value transfers restart the lockup clock — anyone can lock any holder forever
H-2	High	DGXSecurityToken · forcedTransfer	Bypasses every moduleCheck — holder cap, country allow-list and lockup are all agent-breachable
H-3	High	DGXSecurityToken · mint	Reentrancy through compliance.created double-spends reserve headroom
H-4	High	DGXIdentityRegistry · _topicSatisfied	Unbounded claim loop + hostile-issuer gas burn — try/catch does not protect against gas exhaustion
H-5	High	9 contracts	Ownership is non-transferable — module and registry admin is permanently the deploy EOA
H-6	High	ReserveCapModule · moduleBurnAction	Burn re-opens mint headroom before the attestation is refreshed; redeemed metal is counted twice
H-7	High	DGXModularCompliance · addModule	Module ↔ compliance binding is one-shot and unverified — total transfer DoS, unrecoverable module state
H-8	High	AtomicSettlementDvP · executeAtomicSettlement	Settles nothing — no value moves, either party marks it Settled unilaterally
H-9	High	DGXBullionToken · mintBullion	Unlimited mint against a single bar; oracle never consulted; sender not whitelist-checked
M-1..M-13	Medium	see below	
L-1..L-8	Low	see below	
I-1..I-6	Info	see below	

Critical

C-1 — The reserve invariant is enforced against a counter, not against supply

```
src/trex/modules/ReserveCapModule.sol:36,90-95,108-128 · issuedUnits , availableToIssue ,  
moduleCheck
```

The module's stated invariant is "issued supply can never exceed the attested reserve." What the code actually enforces is `issuedUnits ≤ attestedUnits`, where `issuedUnits` is a counter the module increments in its own `moduleMintAction`. **The module holds no reference to the token and never reads `totalSupply()`**. Nothing anywhere compares the two. `issuedUnits` is not a measurement; it is a hope.

`ReserveCapModule.setCompliance` is one-shot (`ComplianceAlreadySet`, line 73) and `DGXModularCompliance.bindToken` is one-shot (`AlreadyBound`, line 56). Together these make module migration mandatory whenever compliance is replaced — and migration resets the counter.

Exploit — no compromise required, this is the documented upgrade path: 1. Token is live, `totalSupply = 1,000e18`, `attestedUnits = 1,000e18`, `issuedUnits = 1,000e18`. Headroom zero. Correct. 2. Operations needs a new policy module but `DGXModularCompliance` is at `MAX_MODULES`, or a module needs re-pointing. The only path is a fresh `DGXModularCompliance` (the old one's `token` is immutably bound). 3. The existing `ReserveCapModule` **cannot** be attached to the new compliance — `setCompliance` reverts `ComplianceAlreadySet`. A new `ReserveCapModule` must be deployed, with `issuedUnits = 0`. 4. `token.setCompliance(newCompliance)` (owner, one call, one event). 5. `availableToIssue()` now returns `1,000e18` against a supply that is already `1,000e18`. An agent mints another `1,000e18`. **Supply is 2× the attested reserve and the module reports full compliance.**

Additional unconstrained mint paths, all silent: - `DGXModularCompliance.removeModule(capModule) → mint → addModule(capModule)`. `issuedUnits` is preserved across the gap, so it now permanently understates supply. Later audits of the module read clean. - `token.setCompliance()` to any compliance with an empty `_modules` array — `canTransfer` returns `true` vacuously (line 93). - `capModule.setAttestationSource(hostile)` — module owner, no timelock, `issuedUnits` carries over. - The cap module is simply never added to `_modules`. Nothing on the token asserts a reserve module exists.

Second-order: after any desync where `issuedUnits < totalSupply`, `moduleBurnAction` reverts `BurnExceedsIssued` once cumulative burns pass `issuedUnits` (line 124) → **`token.burn` is permanently DoS'd and holders cannot redeem.**

Fix: give the module an immutable `token` reference and derive headroom from live supply:

```
function availableToIssue() public view returns (uint256) {  
    (uint256 attested, uint256 asOf) = attestationSource.attestation();  
    if (maxAge != 0 && block.timestamp > asOf + maxAge) return 0;  
    uint256 outstanding = IERC20(token).totalSupply();  
    return attested <= outstanding ? 0 : attested - outstanding;  
}
```

Delete `issuedUnits`, `moduleMintAction` and `moduleBurnAction` bookkeeping entirely. Additionally: make `setCompliance` re-pointable by the owner (guarded), and have `DGXSecurityToken.setCompliance` require `IModularCompliance(c).token() == address(this)` plus emit the outgoing compliance address.

Test gap: the suite asserts `issuedUnits ≤ attested` but never asserts `issuedUnits == totalSupply`, and never removes/re-adds a module or swaps compliance.

C-2 — A single attester can permanently disable the staleness halt, or permanently brick issuance

`src/trex/modules/ReserveCapModule.sol:176-184` · `ReserveAttestationRegistry.publish`

```
if (asOf <= _asOf) revert BackwardsAttestation(asOf, _asOf);
_attestedUnits = attestedUnits;
_asOf = asOf;
```

`asOf` is caller-supplied with **no upper bound and no** `asOf <= block.timestamp` check, and monotonicity is enforced only upward. There is no owner override to lower `_asOf`. The constructor makes the deployer the sole attester (line 163), so `isAttester` is a set of size one in the default deployment.

Exploit A — unlimited issuance, permanent:

```
attestor.publish(type(uint256).max / 2, block.timestamp + 3153600000, 0x00) // asOf = +100 years
```

- `attestationIsFresh()` → `block.timestamp <= asOf + maxAge` → `true`, forever. The "issuance halts when the attestation goes stale" property is **permanently off**.
- `availableToIssue()` → `attested - issuedUnits` $\approx 2^{255}$. Every mint passes.
- Monotonicity now blocks every honest attestation: any real `asOf` is `<= _asOf` and reverts. **The registry can never be corrected.** `setAttester` does not help — the state is the problem, not the key.
- Only recovery is `capModule.setAttestationSource(newRegistry)`, held by the module owner — which per **H-5** is a non-transferable EOA.

Exploit B — permanent issuance DoS (one fat-fingered publish):

```
attestor.publish(1000e18, type(uint256).max, 0x00)
```

With `maxAge != 0`, `asOf + maxAge` overflows. Checked arithmetic in 0.8 panics, so `availableToIssue()`, `attestationIsFresh()`, `moduleCheck(address(0), ...)` and therefore **every mint() revert forever**. (Secondary transfers survive — `moduleCheck` short-circuits on `from != address(0)` at line 113 — so this is an issuance kill, not a transfer kill.)

Fix:

```
if (asOf <= _asOf) revert BackwardsAttestation(asOf, _asOf);
if (asOf > block.timestamp) revert FutureAttestation(asOf);
if (asOf + maxDrift < block.timestamp) revert TooOld(asOf);    // reject backdated-but-forward
publishes
if (evidenceHash_ == bytes32(0)) revert EmptyEvidence();
```

Use `asOf + maxAge` in unchecked-safe form (`asOf > type(uint256).max - maxAge` guard, or compare `block.timestamp - asOf > maxAge` after an `asOf <= block.timestamp` check). Require k-of-n attestors (this is the contract's stated purpose — "multi-signed or replaced by a threshold scheme" — but `publish` is 1-of-n today). Add an owner-held emergency reset that can lower `_asOf` behind a timelock. Bound `attestedUnits` to a governance-set ceiling.

Test gap: 'backdated attestation is refused' covers `asOf < _asOf`. Nothing covers `asOf > now`.

C-3 — A single KYC claim satisfies every required claim topic

`src/trex/DGXIdentityRegistry.sol:107-120` · `_topicSatisfied`

```
bytes32[] memory claimIds = id.getClaimIdsByTopic(topic);
for (uint256 j = 0; j < claimIds.length; j++) {
    (uint256 claimTopic,, address issuer, bytes memory sig, bytes memory data,) =
    id.getClaim(claimIds[j]);
    if (!trustedIssuers.hasClaimTopic(issuer, topic)) continue;
    try IClaimIssuer(issuer).isClaimValid(id, claimTopic, sig, data) returns (bool ok) {
```

`claimTopic` is read out of the claim and passed to the issuer, but is **never compared to `topic`** — the topic the registry is trying to satisfy. The registry trusts an arbitrary contract at `_identity[wallet]` (registered with no key-purpose proof, see M-3) to put claims in honest buckets.

Exploit: 1. Attacker is genuinely KYC'd — they hold a real, valid topic-1 claim from `KYCProvider`, which the issuer trusts for topics 1, 2, 3 and 4 (a single-provider deployment is the norm and is what the suite's fixture does). 2. Attacker is **not** accredited (topic 3) and sits in a **blocked** country (topic 4). 3. Attacker deploys `EvilId`: `solidity function getClaimIdsByTopic(uint256) external view returns (bytes32[] memory) { return _all; // returns the single genuine topic-1 claim id for EVERY topic } function getClaim(bytes32) external view returns (uint256,uint256,address,bytes memory,bytes memory,string memory) { return (1, 1, KYC_PROVIDER, realSig, realData, ""); // honestly reports topic 1 } function keyHasPurpose(bytes32, uint256) external pure returns (bool) { return true; }` 4. Agent calls `registerIdentity(attackerWallet, address(evilId), 840)` — the registry performs no key-purpose or claim check at registration. 5. `isVerified(attackerWallet): for topic = 3, hasClaimTopic(KYCProvider, 3) is true; isClaimValid(evilId, 1, realSig, realData)` verifies a **genuine** signature over `(evilId, topic 1,`

`data`) and returns `true` → topic 3 satisfied. Same for 2 and 4. 6. `isVerified` → `true`. The attacker is now a fully verified holder of a Reg D instrument with none of the claims that matter. `CountryRestrictModule` is separately defeated because `registerIdentity` took the country as an unverified argument.

The cost is one contract deployment. There is no on-chain trace.

Fix — one line:

```
if (claimTopic != topic) continue;
```

Additionally: `registerIdentity` should require

`IIdentity(onchainId).keyHasPurpose(keccak256(abi.encode(wallet)), 1)` — the same proof `recoveryAddress` already demands (line 274). It is inconsistent that the recovery path proves wallet ↔ identity control and the primary registration path does not.

Test gap: the suite builds identities with `MockOnchainId`, which always stores `topic` consistently with the bucket. It never models a non-compliant ONCHAINID, which is exactly the adversary.

High

H-1 — Anyone can permanently lock any holder with zero-value transfers

`src/trex/modules/PolicyModules.sol:228-246` · `LockupModule.moduleTransferAction` / `_startLockup`

```
function moduleTransferAction(address, address to, uint256) external onlyCompliance {
    _startLockup(to);    // amount is ignored
}
```

`moduleCheck` returns `true` unconditionally for `amount == 0` in **every** module (`ReserveCapModule:115`, `MaxHoldersModule:67`, `CountryRestrictModule:157`, `LockupModule:222`), and `_gatedTransfer` has no minimum. So a zero-value transfer completes the full path and reaches `_startLockup(to)`, which sets `unlockAt[to] = block.timestamp + lockupSeconds`.

Exploit: 1. Griefer `G` is any verified, unfrozen holder (or is onboarded as one — this is a Reg D investor, not an outsider). 2. Victim `V` holds 10,000 units with `unlockAt[V]` about to expire; `lockupSeconds = 180 days`. 3. `G` calls `token.transfer(V, 0)`. Cost: ~80k gas. `unlockAt[V] = now + 180 days`. 4. `V` can no longer sell. `G` repeats monthly. **The Rule 144 holding period never ends.**

Aggravating: `forcedTransfer` also routes through `compliance.transferred` (line 258), so a court-ordered seizure to a custodian restarts *the custodian's* lockup for the full period. And `recoveryAddress` restarts the recovered wallet's lockup — a holder who loses a key is penalised with a fresh 180-day hold on a position they have held for years.

Fix: `if (amount == 0) return;` at the top of `moduleTransferAction`, and reject `amount == 0` in `_gatedTransfer`. Separately, track the lockup against the `identity` (`identityRegistry.identity(wallet)`) rather than the wallet, so recovery and forced transfer preserve the original clock; and have `LockupModule` expose a compliance-owner `resetLockup(wallet, until)`.

H-2 — `forcedTransfer` bypasses every compliance module check

src/trex/DGXSecurityToken.sol:243-261

`forcedTransfer` checks `to != 0`, `isVerified(to)`, and sufficient balance. It **never calls** `compliance.canTransfer` — but it *does* call `compliance.transferred`, so the modules' state is updated for a transfer they were never asked to approve.

Consequences, each reachable with a single agent call: - **MaxHoldersModule**: `moduleTransferAction` → `_credit(to)` → `holderCount += 1` with no cap test. `holderCount` can be driven **above** `maxHolders`. For a Reg D 506(b) 2,000-holder or Reg S limit this is a securities-law breach committed by a routine operational call. Afterwards `setMaxHolders` reverts `CapBelowCurrent` for any sane value, so the admin cannot even re-assert the cap downward. - **CountryRestrictModule**: units are delivered into a wallet whose `investorCountry` is explicitly denied. - **LockupModule**: an agent moves units out of a locked wallet, defeating the holding period; and locks the recipient (H-1).

Sequence: `maxHolders = 5`, `holderCount = 5`. `token.transfer(newHolder, x)` correctly reverts `ComplianceRejected`. `token.forcedTransfer(anyHolder, newHolder, x)` succeeds. `holderCount == 6`.

Fix: `forcedTransfer` should call `compliance.canTransfer(from, to, amount)` and revert unless it passes, with an explicit, separately-role-gated `forcedTransferOverride` (a distinct `RECOVERY_AGENT` role, ideally a timelock) for genuine court orders — and that override should emit a distinguishable event so an indexer can flag every compliance-bypassing move.

Test gap: the suite tests the holder cap only on the `transfer` path (`'MAX HOLDER CAP BLOCKS A NEW HOLDER'`). It never calls `forcedTransfer` at the cap, or into a closed country.

H-3 — Reentrancy through `compliance.created` double-spends reserve headroom

src/trex/DGXSecurityToken.sol:183-192 and **DGXModularCompliance.sol:104-109**

`mint` is check → effect → **external call**, but the *cap module's* accounting happens inside that external call:

```
if (!compliance.canTransfer(address(0), to, amount)) revert ComplianceRejected(); // reads
issuedUnits
totalSupply += amount;
_balances[to] += amount;
compliance.created(to, amount); // issuedUnits is updated HERE, in a loop over arbitrary
modules
```

The token's own balances are CEI-correct. `issuedUnits` is not. `addModule` accepts **any address** (line 61) and the `DGXModularCompliance` owner is a *different* principal from the token owner, so the module set is not necessarily under the token owner's control.

Exploit (`_modules = [HostileModule, ReserveCapModule]`, `attested = X`, `issuedUnits = 0`): 1. Agent calls `mint(to, X)`. 2. `canTransfer` → `Hostile.moduleCheck` → `true`; `ReserveCap.moduleCheck` → `X <= availableToIssue()` → `true`. 3. `totalSupply = X`; `_balances[to] = X`. 4. `compliance.created(to, X)` → `_modules[0].moduleMintAction = Hostile`, which reenters `token.mint(to, X)` :- `canTransfer` → `ReserveCap.moduleCheck` → **`issuedUnits` is still 0** → `X <= X` → `true`. - `totalSupply = 2X`; `created` → `Hostile` (guards, no further recursion) → `ReserveCap.moduleMintAction(X)` → `issuedUnits = X`. 5. Unwind: `_modules[1].moduleMintAction(X)` → `issuedUnits = 2X`. 6. **`totalSupply = issuedUnits = 2X` against `attested = X`**. The invariant is broken and the module's own counter now *records* the breach.

Exploitability depends on the hostile module preceding `ReserveCapModule` in `_modules` — and `removeModule` uses swap-and-pop (line 77), so array order changes unpredictably as policy is edited and is not something an operator can rely on. The same shape applies to `transferred` after `_move`, and will become exploitable the moment any module makes an outbound call (a price oracle, an off-chain-triggered allow-list, an ERC-777-style receiver hook).

Fix: a `nonReentrant` guard on `mint`, `burn`, `transfer`, `transferFrom`, `forcedTransfer` and `recoveryAddress`. Better: adopt the C-1 fix, at which point `ReserveCapModule` reads live `totalSupply()` and the reentrant mint is rejected at step 4 by construction.

H-4 — `try/catch` stops a reverting issuer but not a gas-burning one; the claim loop is unbounded

`src/trex/DGXIdentityRegistry.sol:107-120`

The contract's own NatSpec (line 91) warns that `isVerified` is `0(topics × claims)` and caps topics at 8 (`DGXClaimTopicsRegistry:171`). **The inner dimension is uncapped and is controlled by the party being verified.** `getClaimIdsByTopic` returns an arbitrary-length array from the investor's own ONCHAINID, and each entry with a trusted issuer costs a `STATICCALL`.

Two distinct attacks the suite's 'a reverting issuer does not brick verification' check does not reach:

(a) Hostile / compromised trusted issuer — gas exhaustion, not revert. `try/catch` recovers from a `revert`. It does not recover from an issuer that consumes all forwarded gas:

```
function isClaimValid(...) external view returns (bool) { while (true) {} }
```

Because of EIP-150's 63/64 rule, each such call returns to `_topicSatisfied` having burned 63/64 of the gas then available. With a handful of claims, or one issuer and a caller with a modest gas limit, the outer `isVerified` runs out of gas and the **entire transfer reverts** — this does *not* "fail closed and continue," it halts. Every holder whose only claim for

a topic came from that issuer is frozen, and `untrustIssuer` does not help: `hasClaimTopic` is checked *before* the call, so untrusting fixes it — but only after the owner notices, and `untrustIssuer` then unverifies those holders anyway.

(b) Counterparty-funded griefing. A holder loads 5,000 junk claims into topic 1 of their ONCHAINID, all from a trusted issuer, all invalid, with the one valid claim last (the loop returns on first success, line 115). Every `transfer(_ , thatWallet, _)` makes the **sender** pay 5,000 staticcalls, or exceed the block gas limit. `canTransfer` — the dry-run every front-end and allocator uses to explain a revert — also reverts out of gas, so the failure is undiagnosable.

Fix: bound the inner loop (`for (uint256 j = 0; j < claimIds.length && j < MAX_CLAIMS_PER_TOPIC; j++)`), and replace the `try/catch` with an explicit-stipend low-level staticcall:

```
(bool ok, bytes memory ret) = issuer.staticcall{gas: 100_000}(  
    abi.encodeCall(IClaimIssuer.isClaimValid, (id, topic, sig, data)));  
if (ok && ret.length == 32 && abi.decode(ret, (bool))) return true;
```

Long-term, index claims by `(issuer, topic)` so verification is $O(1)$ per topic rather than a scan.

H-5 — Nine of thirteen contracts have permanently non-transferable ownership

`ReserveCapModule` , `ReserveAttestationRegistry` , `MaxHoldersModule` , `CountryRestrictModule` ,
`LockupModule` , `DGXIdentityRegistry` , `DGXClaimTopicsRegistry` , `DGXTrustedIssuersRegistry` ,
`MineOperatorRegistry`

Only `DGXSecurityToken` (line 316) and `DGXModularCompliance` (line 118) implement `transferOwnership` . Everything else sets `owner = msg.sender` in the constructor with no setter. These contracts are not upgradeable and there are no proxies.

The deploying key therefore holds, permanently and irrevocably: | Power | Contract | ---|---| | `setAttestationSource` , `setMaxAge` (incl. 0 = disable staleness) | `ReserveCapModule` || `setAttestor` — add/remove who can publish reserves | `ReserveAttestationRegistry` || `setMaxHolders` | `MaxHoldersModule` || `setCountryAllowed` | `CountryRestrictModule` || `setLockup` | `LockupModule` || `setAgent` , `setTopicsRegistry` , `setTrustedIssuers` | `DGXIdentityRegistry` || `addClaimTopic` / `removeClaimTopic` | `DGXClaimTopicsRegistry` || `trustIssuer` / `untrustIssuer` | `DGXTrustedIssuersRegistry` || `setAgent` (who can accept operators) | `MineOperatorRegistry` |

The token's own NatSpec instructs "Deploy the owner as a multisig or timelock; never collapse both onto one EOA" — but for nine contracts there is **no mechanism to ever move ownership to a multisig after deployment**, and `DGXIdentityRegistry.setAgent` alone lets the deploy key make itself an identity agent and register arbitrary wallets. If the deploy key is a hot key (and it is at least an EOA that also holds `isAgent` at genesis per L-5), the entire compliance perimeter is held by it forever, and key compromise is unrecoverable without redeploying the whole stack and migrating every holder.

Fix: add two-step `transferOwnership / acceptOwnership` (OpenZeppelin `Ownable2Step`) to all nine, and make the deploy script hand ownership to a Safe + timelock in the same transaction bundle as deployment. Add `pendingOwner` events to all of them.

H-6 — Burning re-opens mint headroom before the attestation is refreshed

`src/trex/modules/ReserveCapModule.sol:123-128` · `moduleBurnAction`

```
function moduleBurnAction(address, uint256 amount) external onlyCompliance {
    if (amount > issuedUnits) revert BurnExceedsIssued(issuedUnits, amount);
    unchecked { issuedUnits -= amount; }
}
```

A burn is a redemption: the holder surrenders tokens and the vault ships metal. `issuedUnits` drops immediately. `attestedUnits` does **not** — it only changes when the attester next publishes, which by design is periodic (`maxAge` is typically days). In the gap, the module reports headroom for metal that has already left the vault.

Sequence: `attested = 1,000`, `issuedUnits = 1,000`, headroom `0`. 1. Holder redeems 300. `burn(holder, 300)` → `issuedUnits = 700`. Bars 1–3 are released to the holder's carrier. 2. `availableToIssue()` → `1000 - 700 = 300`. The attestation is still fresh (within `maxAge`), so no halt. 3. Agent mints 300 to a new subscriber, backed by bars that are physically in transit to someone else. 4. Attester publishes the true count next cycle: `attested = 700`, `issuedUnits = 1,000`. `availableToIssue()` correctly returns 0 — but **300 units are already outstanding with no backing, and nothing in the system signals it**. There is no `isOverIssued()` view, no event, no auto-pause.

Fix: pair every burn with an attestation decrement, or gate mint on a per-cycle issuance budget that only a fresh attestation replenishes. At minimum, add:

```
function overIssuance() external view returns (uint256) {
    (uint256 attested,) = attestationSource.attestation();
    uint256 outstanding = IERC20(token).totalSupply();
    return outstanding > attested ? outstanding - attested : 0;
}
event OverIssuanceDetected(uint256 outstanding, uint256 attested);
```

and have `ReserveAttestationRegistry.publish` revert (or emit a loud event and auto-pause the token) when the new `attestedUnits` falls below outstanding supply.

Related front-running (same root cause): a downward attestation revision sits in the public mempool. An agent, or an MEV searcher paying an agent, front-runs it with a `mint` that consumes the soon-to-be-invalid headroom. Post-inclusion, `issuedUnits > attested` permanently and irreversibly. Mitigate with a private relay for `publish`, a commit–reveal, or by having `publish` take effect only after a delay during which minting is paused.

H-7 — Module ↔ compliance binding is one-shot, unverified, and mis-binding is a total DoS

`src/trex/DGXModularCompliance.sol:61-69` , `DGXSecurityToken.sol:310-314`

`addModule` stores an arbitrary address. It does **not** verify `IModule(module).compliance() == address(this)`. Every module's `setCompliance` is one-shot (`ReserveCapModule:73` , `MaxHoldersModule:51` , `CountryRestrictModule:141` , `LockupModule:206`).

Failure 1 — total transfer DoS from a two-step deployment. Operator deploys `MaxHoldersModule` and calls `compliance.addModule(m)` but not `m.setCompliance(compliance)`. `moduleCheck` is a `view` and works fine, so `canTransfer` returns `true` and the deployment looks healthy. Then on the first real transfer: `_gatedTransfer` → `_move` succeeds → `compliance.transferred` → `m.moduleTransferAction` → `revert NotCompliance`. **Every transfer, mint and burn reverts.** The instrument is bricked. Recovery requires `removeModule`, which is fine — but see Failure 2.

Failure 2 — the module can never be re-pointed. If `setCompliance` was called against the *old* compliance contract, the module can never be attached to a new one. A compliance migration therefore forces fresh modules, which resets `issuedUnits`, `holderCount` and every `shadowBalance` — the C-1 and M-1 failure modes, arrived at through ordinary operations.

Failure 3 — token ↔ compliance mis-binding. `DGXSecurityToken.setCompliance` does not check `IModularCompliance(c).token() == address(this)`. Pointing the token at a compliance bound to a different token (or to nothing) makes every `transferred` / `created` / `destroyed` `revert NotToken` — again, invisible until the first value movement, because `canTransfer` still works.

Fix:

```
function addModule(address module) external onlyOwner {
    ...
    if (IModule(module).compliance() != address(this)) revert ModuleNotBoundToThis();
}
function setCompliance(address c) external onlyOwner {
    if (IModularCompliance(c).token() != address(this)) revert ComplianceNotBoundToThis();
}
```

Make module `setCompliance` owner-re-pointable (with the old compliance's consent or a timelock) rather than one-shot, and add `compliance()` and `token()` to `IModule` / `IModularCompliance`.

H-8 — `AtomicSettlementDvP` performs no settlement

`src/AtomicSettlementDvP.sol:56-64`

```
function executeAtomicSettlement(uint256 orderId) external {
    require(order.status == OrderStatus.Open, "DvP: order not open");
    ...
    order.status = OrderStatus.Settled;
    emit DvPOrderSettled(orderId, block.timestamp);
}
```

There is **no transferFrom**, **no escrow**, **no balance change of any kind** — for either the payment token or the asset token. The contract's NatSpec claims it "atomically exchanges cash/stablecoins against physical gold token title with zero counterparty risk." It exchanges nothing. Either party may unilaterally flip an order to **Settled** and emit an event that an off-chain reconciliation system will treat as proof of settlement.

Additional defects in the same contract: **createOrder** lets a buyer name an arbitrary **seller** who never consents and is never notified; **cancelOrder** is buyer-only, so a seller cannot escape a stale order; expired orders are never reaped; **block.timestamp + validityDurationSecs** reverts on overflow for large inputs.

Fix: either implement real DvP (buyer escrows **paymentAmount** on **createOrder** via **safeTransferFrom**; **executeAtomicSettlement** performs both legs in one transaction and reverts if either leg fails — noting that the asset leg goes through the ERC-3643 gate and can legitimately fail, so the payment leg must roll back), or delete the contract. Shipping it as-is under this name is the single most likely item to be flagged in an allocator's technical diligence.

H-9 — **DGXBullionToken** : unlimited mint per bar, oracle never consulted, sender never whitelist-checked

src/DGXBullionToken.sol:90-98, 109-135, 47-50

- **mintBullion(recipient, amount, barHash)** requires only that the bar is enrolled. **amount** is arbitrary and **unrelated to barRegistry[barHash].fineWeightGrams**, and the same **barHash** may be minted against an unlimited number of times. One 400oz bar backs infinite supply.
- **proofOfReservesOracle** is stored (line 16) but **never read**. **ProofOfReservesOracle.totalVaultedGrams** constrains nothing. The proof of reserves is decorative.
- **transfer** (line 110) requires **isWhitelisted[to]** but **not isWhitelisted[msg.sender]**; **transferFrom** (line 126) likewise does not check **from**. A de-whitelisted (sanctioned) holder can still move their entire position. For a contract whose header claims "ERC-3643 Permissioned Regulatory Identity Rails," the outbound permission check is absent.
- **redeemPhysical** (line 100) is a permissionless burn with no whitelist check, no reserve check and no redemption authorisation.
- **onlyOracle** accepts **msg.sender == governor** (line 48), collapsing the independent-attestor role into the issuer.
- **governor** has no setter — key loss permanently freezes minting and whitelisting; **constructor** does not reject **address(0)**.

Fix: if this contract is still live, it should be migrated to the T-REX stack and deprecated. If it must remain: track `mintedAgainst[barHash]` and cap at `fineWeightGrams`; read and enforce `ProofOfReservesOracle.totalVaultedGrams` against `totalSupply`; check `isWhitelisted[msg.sender]` / `isWhitelisted[from]` on both transfer paths; gate `redeemPhysical`; split `onlyOracle` from `onlyGovernor`; add governor rotation.

Medium

M-1 — `MaxHoldersModule` shadow balances are zero-initialised and can never be reconciled

`src/trex/modules/PolicyModules.sol:22, 62-97`

`shadowBalance` and `holderCount` start empty and there is **no admin sync function**. The module is correct only if it is bound before the very first mint and never detached.

Drift scenario (the realistic one): the token launches with `_modules = [ReserveCapModule]`. 500 investors are onboarded. Legal then requires a Reg D holder cap, so `MaxHoldersModule(cap = 600)` is deployed and added. `holderCount == 0` and every existing holder has `shadowBalance == 0`. - `moduleCheck` treats all 500 existing holders as new. 600 further onboardings are admitted before the cap bites. **Actual holders: 1,100 against a cap of 600.** - Worse in the other direction: an existing holder with a real balance of 1,000 receives 1 unit → `_credit` sets `shadowBalance = 1`, `holderCount = 1`. They then send their full 1,000 → `_debit(who, 1000)` hits the clamp at line 94 (`amount >= bal ? 0 : bal - amount`) → `shadowBalance = 0`, `holderCount -= 1`. The recipient is credited +1. Net: the count tracks a fiction.

The `_debit` clamp is the root enabler: it **silently absorbs any inconsistency** instead of reverting, so drift is permanent and undetectable on-chain. There is no view that compares `shadowBalance[w]` to `token.balanceOf(w)`.

Fix: read `token.balanceOf()` directly in `moduleCheck` (`if (IERC20(token).balanceOf(to) != 0) return true;`) and maintain `holderCount` from the token's own state, eliminating shadow balances entirely — the same structural fix as C-1. If shadow state is kept, replace the clamp with an explicit revert, add an owner `seedHolders(address[] calldata, uint256[] calldata)` migration function, and add a `reconcile(address)` view.

Test gap: the suite adds all modules before any mint, so this path is never exercised.

M-2 — `recoveryAddress`: unfrozen state window, reentrancy exposure, and a silently dropped sanctions flag

`src/trex/DGXSecurityToken.sol:267-296`

```

frozenTokens[lostWallet] = 0;           // direct write – no TokensUnfrozen event
frozen[lostWallet] = false;             // direct write – no AddressFrozen event
forcedTransfer(lostWallet, newWallet, bal); // ← external call to compliance.transferred
if (lockedTokens > 0) freezePartialTokens(newWallet, lockedTokens);
if (wasFrozen) setAddressFrozen(newWallet, true);

```

(a) Reentrancy window. During `compliance.transferred` inside `forcedTransfer`, `newWallet` holds the entire recovered position, `frozenTokens[newWallet]` has not yet been set, and `frozen[newWallet]` is still `false`. A bound module reached in that loop can call `token.transferFrom(newWallet, attacker, bal)` if it holds an allowance from `newWallet`. The value escapes a freeze that, after the transaction, the chain state claims was preserved. This requires a hostile/compromised bound module plus a pre-existing allowance — narrow, but the compliance owner is a distinct principal from the token owner and `addModule` takes arbitrary code, so the trust boundary is real.

(b) The re-freeze is unfaithful in one direction. `setAddressFrozen(newWallet, true)` only fires if `(wasFrozen)`. If `newWallet` was *already* frozen and `lostWallet` was not, the freeze correctly persists. But `frozen[lostWallet] = false` is never restored — on success, the old wallet is left **unfrozen with no event**. `AddressFrozen` and `TokensUnfrozen` are the events a sanctions-monitoring pipeline subscribes to, and this path bypasses both emitters. If the wallet is later re-registered (nothing prevents it), it is silently clean.

(c) No `lostWallet != newWallet` guard. If they are equal: `forcedTransfer(w, w, bal)` is a no-op self-move, `freezePartialTokens` restores the lock, and then `deleteIdentity(lostWallet)` **deletes the identity of the wallet that was just registered**. The holder keeps their balance but is permanently unverified and can never transfer. `RecoverySuccess` still emits.

Fix: inline the move rather than calling `forcedTransfer`, set `frozen[newWallet]` and `frozenTokens[newWallet]` **before** the `compliance.transferred` call, add `nonReentrant`, emit `AddressFrozen(lostWallet, false, msg.sender)` and `TokensUnfrozen(lostWallet, lockedTokens)` for the clearing writes, and add `if (lostWallet == newWallet) revert SameWallet();`.

M-3 — `registerIdentity` silently overwrites; `AlreadyRegistered` is dead code

`src/trex/DGXIdentityRegistry.sol:37, 67-72`

`error AlreadyRegistered(address wallet);` is declared and **never used**. `registerIdentity` overwrites `_identity[wallet]` and `_country[wallet]` with no prior-state check, emitting only the generic `IdentityRegistered`. Consequences: - Any identity agent can re-point an existing holder's ONCHAINID and country. `CountryRestrictModule` reads `investorCountry` live, so a holder's jurisdiction — and therefore whether they can receive units — changes with one call and no distinguishable event. - `recoveryAddress` inherits this: if `newWallet` already belongs to a *different* investor, recovery silently rebinds that investor's wallet to the lost investor's ONCHAINID. - No key-purpose proof at registration (see C-3).

Fix: revert `AlreadyRegistered` when `_identity[wallet] != address(0)`; add a separate `updateIdentity` that emits `IdentityUpdated(wallet, oldId, newId)`; require `keyHasPurpose(keccak256(abi.encode(wallet)), 1)` on the identity.

M-4 — `CountryRestrictModule` pins its own identity registry with no setter

`src/trex/modules/PolicyModules.sol:110, 132-136, 158`

`identityRegistry` is set in the constructor and has **no setter**. `DGXSecurityToken.setIdentityRegistry` (line 304) can re-point the token at a new registry at any time. After such a migration the token verifies against registry B while the country module reads jurisdictions from registry A. A wallet registered only in B has `investorCountry == 0` in A; `allowedCountry[0]` is `false` by default, so the holder is silently blocked from receiving with a `ComplianceRejected` that names no cause. If country `0` was ever opened for any reason, the inverse happens and the allow-list is fully bypassed. The module cannot be fixed — only redeployed, which re-triggers H-7.

Fix: add `setIdentityRegistry (owner)` to the module, or have it read the registry through `IToken(compliance.token()).identityRegistry()` so the two can never diverge. Also treat country `0` as an explicit sentinel and reject it in `setCountryAllowed` and `registerIdentity`.

M-5 — `pause` does not stop privileged value movement, and any agent can unpause

`src/trex/DGXSecurityToken.sol:194, 228-236, 243, 267`

`burn`, `forcedTransfer` and `recoveryAddress` all lack `whenNotPaused`. During an emergency pause — the thing you reach for after a key compromise — an attacker holding a single agent key can still burn any holder's unfrozen balance and force-transfer any holder's entire position to any verified wallet. `unpause` is `onlyAgent`, not `onlyOwner`, so a compromised agent simply turns the pause off; the owner has no unilateral kill switch it can hold against its own agents.

Fix: `whenNotPaused` on `burn`, `forcedTransfer` and `recoveryAddress` (with an explicit, owner-held `emergencyForcedTransfer` if court orders must execute during a pause). Make `pause` agent-callable but `unpause` `onlyOwner`.

M-6 — Config changes that determine who is verified emit no events

`src/trex/DGXIdentityRegistry.sol:127-135`, `DGXModularCompliance.sol:118-121`,
`MineOperatorRegistry.sol:172-175`

`setTopicsRegistry` and `setTrustedIssuers` emit **nothing**. Swapping the topics registry is the single highest-impact configuration change in the system — it redefines verification for every holder at once — and it is invisible to any log-based monitor. `DGXModularCompliance.transferOwnership` and `MineOperatorRegistry.setAgent` are likewise silent (the latter despite `DGXSecurityToken.setAgent` emitting `AgentSet`, so the inconsistency will fool an indexer built against the token).

Fix: emit `TopicsRegistrySet(address old, address new)`, `TrustedIssuersSet(address old, address new)`, `OwnershipTransferred`, `AgentSet` respectively. Include the previous value — several existing events (`IdentityRegistrySet`, `ComplianceSet`) also omit it, which makes reorg-safe indexing harder than it needs to be.

M-7 — `mint` does not check `frozen[to]`

`src/trex/DGXSecurityToken.sol:183-192`

`_gatedTransfer` checks `frozen[from]` and `frozen[to]` (lines 158-159). `mint` checks neither. An agent can issue new units into a wallet that has been frozen for sanctions reasons. The units are stuck there (outbound transfers are blocked) so this is not direct value escape, but it breaks the "frozen means no activity" property that a sanctions programme relies on, and it produces a `Transfer` into a blocked address that an off-chain screening system will flag as a control failure.

Fix: `if (frozen[to]) revert WalletFrozen(to);` in `mint`. Consider the same for `forcedTransfer`'s recipient, with a documented exception for a court-ordered move into a custodial escrow.

M-8 — `MineOperatorRegistry`: one agent walks the whole state machine; `Rejected` is a dead end

`src/trex/MineOperatorRegistry.sol:99-170`

- Every transition — `submitApplication`, `executeNda`, `openDiligence`, `accept`, `suspend`, `reinstate`, `reject` — is guarded by the `same onlyAgent` role. The contract's premise ("the applicant never self-accepts", "there is no path from Submitted straight to Accepted, so an asset cannot be onboarded without a diligence step") holds only against *sequence*, not against *authority*: a single agent key executes `submitApplication` → `executeNda` → `openDiligence` → `accept` in four calls inside one block. The diligence gate is a formality with no maker-checker split and no minimum dwell time.
- `reject` is documented as "terminal unless re-applied" (line 164) but `submitApplication` requires `status == None` (line 105) and nothing ever resets status. **A rejected operator can never re-apply under the same `operatorId`** — behaviour that directly contradicts the comment an allocator will read.
- `reinstate` re-emits `OperatorAccepted` (line 160) and does not update `decidedAt`, so an indexer cannot distinguish a first acceptance from a reinstatement after suspension-for-cause.

Fix: split roles (`INTAKE_AGENT` for submit/NDA, `DILIGENCE_AGENT` for openDiligence, `COMMITTEE` — ideally a multisig — for accept/suspend/reject); enforce `block.timestamp >= o.submittedAt + minDiligencePeriod` in `accept`; add an owner-gated `reopen(operatorId)` from `Rejected`, or make the docstring match; emit a distinct `OperatorReinstated`.

M-9 — `mayIssue` is never consumed by the issuance layer

`src/trex/MineOperatorRegistry.sol:89-93`

The NatSpec says `mayIssue` "is what the issuance layer should check before configuring a token for an operator's asset."
No contract in `src/` reads it. `DGXSecurityToken.mint` does not; `ReserveCapModule` does not; `DGXModularCompliance` has no operator module. The operator state machine is an off-chain-enforced record with an on-chain audit trail, not a gate. The invariant suite confirms `mayIssue` returns the right boolean but never asserts that anything acts on it.

Fix: either add an `OperatorGateModule` implementing `IModule` that vetoes minting when the configured `operatorId` is not `Accepted` (making it a genuine on-chain gate), or amend the NatSpec and the offering documents to state plainly that the registry is an evidentiary record and the gate is procedural. The current wording will not survive diligence.

M-10 — `setMaxAge(0)` is a silent kill switch for the staleness halt; `setLockup` has no upper bound

`src/trex/modules/ReserveCapModule.sol:84-87, 92, 99` and `PolicyModules.sol:210-213, 240`

- `availableToIssue` and `attestationIsFresh` both short-circuit on `maxAge == 0`. `setMaxAge(0)` — one call, held by a non-transferable EOA (H-5) — permanently disables the halt-on-stale property while `MaxAgeSet(0)` scrolls past unremarked. The constructor also accepts `maxAge_ = 0`.
- `setLockup(type(uint256).max)` makes `block.timestamp + lockupSeconds` overflow in `_startLockup` (line 240), so **every transfer reverts**: a one-call total DoS from the lockup module's owner.

Fix: enforce `MIN_MAX_AGE <= maxAge_ <= MAX_MAX_AGE` (reject `0`); enforce `lockupSeconds_ <= 10 * 365 days`; put both setters behind a timelock.

M-11 — `enrollBarPassport` hash collision via `abi.encodePacked` on three dynamic strings

`src/DGXBullionToken.sol:71`

```
bytes32 barHash = keccak256(abi.encodePacked(serialNumber, refiner, vaultEnclave));
```

`encodePacked` does not delimit dynamic types. `("AB", "C", "Zurich")` and `("A", "BC", "Zurich")` produce an identical `barHash`. Two physically distinct bars collapse to one passport; the second `enrollBarPassport` reverts "Bar already enrolled" (so the second bar can never be registered), or, depending on ordering, one bar's provenance record is the one that survives while the other is minted against. For a serial-number-based bullion passport this is a direct chain-of-custody defect.

Fix: `keccak256(abi.encode(serialNumber, refiner, vaultEnclave))`.

M-12 — `MiningStreamingForge`: completed concessions can be silently overwritten; deliveries are uncapped

`src/MiningStreamingForge.sol:55, 83-88`

`enrollConcession` guards on `!concessions[hash].isActive`. `logRefineryDelivery` sets `isActive = false` once the commitment is met (line 87). A **completed** concession therefore passes the enrolment guard and is overwritten wholesale, destroying `totalOzDelivered` and the `technicalReportFilingHash` that ties it to the NI 43-101 filing. `totalOzDelivered += ozDelivered` is never capped at `totalOzCommitted`, so over-delivery is silently recorded. `provenReservesOz >= totalOzCommitted * 2` (line 52) reverts on overflow rather than on a business-rule violation for large inputs. `cashPaid` is computed and emitted but no value moves — like H-8, this is a logging contract presented as a settlement contract.

Fix: guard on a separate `exists` flag rather than `isActive`; cap deliveries at the commitment; make the "streaming" semantics explicit in the NatSpec.

M-13 — `IoTEsgOffsetRegistry` permits the same carbon credit to be retired repeatedly

`src/IoTEsgOffsetRegistry.sol:32-47`

`retireCarbon` pushes a record with no uniqueness check on `vcsCreditSerial`. The same VCS serial can be retired an unlimited number of times, and `totalRetirementsCount` will report each as a distinct retirement. Double-counting offsets is the canonical carbon-credit fraud pattern and will be the first thing an ESG assurance provider tests. `registryAdmin` is immutable with no rotation, and the constructor does not reject `address(0)`.

Fix: `mapping(bytes32 => bool) public retiredSerial; keyed on keccak256(bytes(vcsCreditSerial))`, reverting on a repeat; add admin rotation; validate the constructor argument.

Low

- **L-1 — `solc 0.8.20` with no `evm_version` pin.** `foundry.toml` sets `solc_version = "0.8.20"` and omits `evm_version`. 0.8.20 defaults to the Shanghai target and emits `PUSH0`, which several L2s and sidechains did not (or do not) support. Deployment will fail or, worse, succeed with subtly different bytecode across chains. Pin `evm_version = "paris"` (or the lowest target among intended chains) and lock the pragma to `pragma solidity 0.8.20;` rather than `^0.8.20`.
- **L-2 — `burn` is missing `whenNotPaused` and does not check `frozen[from]`.** See M-5. Separately, burning from a frozen wallet is probably intentional (redemption under a court order) but should be a distinct, separately-evented function rather than an unstated property of the ordinary burn path.
- **L-3 — `transferFrom` does not check `frozen[msg.sender]`.** A frozen spender holding a pre-existing allowance can still move an unfrozen holder's tokens (`DGXSecurityToken.sol:138-146`). Add `if (frozen[msg.sender]) revert WalletFrozen(msg.sender);`.
- **L-4 — `setAgent(address(0), true)` is accepted** on `DGXSecurityToken` (line 299) and `DGXIdentityRegistry` (line 122), while `MineOperatorRegistry.setAgent` (line 172) rejects it. Harmless in practice but the inconsistency signals the roles were not reviewed together. Add the check everywhere.

- **L-5 — Constructor collapses owner and agent.** `DGXSecurityToken` (lines 98-99) and `DGXIdentityRegistry` (lines 52-53) both set the deployer as owner *and* agent, and nothing ever requires that to be undone. The token's own `NatSpec` forbids exactly this. Take the intended owner and first agent as constructor arguments, and do not self-assign.
- **L-6 — `MaxHoldersModule(0)` and `addClaimTopic(0)` are deployment footguns.** `maxHolders = 0` makes `holderCount + 1 <= 0` false forever, so no wallet can ever be onboarded until `setMaxHolders` is called. Adding claim topic `0` makes `getClaimIdsByTopic(0)` return empty on any honest `ONCHAINID`, un verifying every holder instantly. Reject both.
- **L-7 — `transferOwnership` is single-step everywhere it exists.** `DGXSecurityToken:316`, `DGXModularCompliance:118`. A typo in the address is unrecoverable. Use two-step accept (and see H-5 for the nine contracts that have no transfer at all).
- **L-8 — `approve` has no allowance-race mitigation and `transferFrom` emits no `Approval` on decrement.** Standard ERC-20 behaviour; noted because institutional integrators frequently check for `increaseAllowance / decreaseAllowance`.

Informational

- **I-1 — Mocks live in `src/`, not `test/`.** `src/trex/mocks/MockIdentity.sol` contains `MockOnchainId` (whose `addKeyPurpose` and `addClaim` are **unpermissioned**) and `MockClaimIssuer`. They compile into the production artifact set and a deploy script that globs `src/**` will ship them. `MockOnchainId` deployed alongside a live instrument is a verified-identity factory for anyone. Move to `test/` and exclude from the deploy manifest.
- **I-2 — Nothing is upgradeable and the migration paths are blocked.** No proxies, no `delegatecall`. Combined with H-7's one-shot binders and H-5's frozen ownership, there is no route to fix a bug in `DGXSecurityToken` short of deploying a new token and re-issuing to every holder. State the migration plan explicitly in the offering documents, or adopt a minimal, timelocked proxy for the token.
- **I-3 — No `canMint` view.** `canTransfer` (line 123) dry-runs the transfer gate but there is no equivalent for the mint gate, which has a different check set (no `frozen`, no `unfrozenBalanceOf`, `from == address(0)`). Operators cannot pre-flight an issuance, and the reserve-cap refusal surfaces only as a generic `ComplianceRejected`. Note that `IssuanceRefused` is declared in `ReserveCapModule` (line 44) and **never emitted** — `moduleCheck` is a `view` and cannot emit, so the event is dead code that implies an audit trail which does not exist.
- **I-4 — No claim expiry.** `_topicSatisfied` accepts any claim the issuer still calls valid. Nothing reads the claim's `data` field for a `validUntil`. "Verified" is permanent until the issuer revokes or is untrusted, which for annual KYC refresh cycles means a stale attestation with no on-chain aging signal.
- **I-5 — `operatorId` is unbound caller-chosen `bytes32`.** `MineOperatorRegistry.submitApplication` accepts any id with no derivation from the applicant's address or the `assetRef`. Agents can squat ids, and two records for the same legal entity under different ids are indistinguishable on-chain. Derive it:
`keccak256(abi.encode(applicantAddress, assetRef))`.
- **I-6 — `ProofOfReservesOracle` has a single immutable auditor and no staleness enforcement.** `primaryAuditor` has no setter (line 9) — key loss permanently ends reserve updates. `updateReserves` permits arbitrary increases and decreases with no monotonic timestamp, no evidence hash, and no consumer. The newer

`ReserveAttestationRegistry` is strictly better; retire this one rather than running both, since two contradicting reserve sources is itself a diligence finding.

Checked and found clean

These are the issues I actively looked for and did **not** find, with the reason:

1. **Classic balance reentrancy on `transfer` / `_move` — not present.** `_gatedTransfer` completes `_move` (balances written, `Transfer` emitted) *before* `compliance.transferred`. A reentrant call re-runs the entire gate against already-updated balances and cannot double-spend. Checks-effects-interactions is respected for the token's own accounting. The reentrancy problem in H-3 is confined to the *module's* accounting, which updates inside the interaction.
2. **Modules cannot mutate state during `canTransfer` — structurally prevented.** `IModule.moduleCheck` is declared `view` in `ITREX.sol:44`, so the compiler emits `STATICCALL`. Even a module implemented with a state-writing `moduleCheck` reverts rather than mutating. `DGXModularCompliance.canTransfer` is correspondingly `view`.
3. **`frozenTokens[w] <= balanceOf(w)` holds on every path.** Verified for `_gatedTransfer` (spendable check, line 161), `burn` (line 195), `forcedTransfer` (explicit release, lines 249-255), `freezePartialTokens` (line 214), `unfreezePartialTokens` (line 221) and `recoveryAddress` (zeroed then re-added against a larger balance). `unfrozenBalanceOf` additionally clamps at line 119. No path can push locked above balance.
4. **All five unchecked blocks are correctly guarded.** `transferFrom` allowance (line 141, guarded by `allowed < amount`), `_move` (line 173, guarded by the spendable check in `_gatedTransfer` and by `amount > _balances[from]` in `forcedTransfer`), `burn` (line 197, `amount <= unfrozenBalanceOf <= balance <= totalSupply`), `unfreezePartialTokens` (line 222, guarded by `amount > locked`), `ReserveCapModule.moduleBurnAction` (line 125, guarded by `amount > issuedUnits`). No underflow is reachable.
5. **`frozenTokens[from] = locked - release` in `forcedTransfer` (line 252) cannot underflow.** `release = amount - free` and `free = balance - locked`, so `release <= locked` given `amount <= balance` (checked at line 246).
6. **Module and topic loops are bounded.** `MAX_MODULES = 12` enforced in `addModule` (line 65); claim topics capped at 8 (`DGXClaimTopicsRegistry:171`). `canTransfer`, `transferred`, `created` and `destroyed` are all $O(12)$ external calls — bounded and affordable. The unbounded loop is the *inner* claim loop only (H-4).
7. **A trusted issuer that is an EOA fails closed.** Solidity ≥ 0.8 inserts an `extcodesize` check before a call with a declared return value; calling an EOA reverts, the `catch` at line 116 swallows it, and the loop continues. `trustIssuer` accepting a non-contract is therefore a config error, not an exploit.
8. **`removeModule`'s swap-and-pop is correct.** Bounds are right (`_modules[i] = _modules[n-1]`; `_modules.pop()`; `break`), `isBound` is cleared, and removing the last element is handled. Only evaluation *order* changes — which matters for H-3's exploitability, not for correctness.

9. **Every value-moving path does call the compliance state hooks.** `transfer` / `transferFrom` → `transferred`; `mint` → `created`; `burn` → `destroyed`; `forcedTransfer` → `transferred`; `recoveryAddress` → `transferred` via `forcedTransfer`. There is no path that moves balances without notifying modules. (The shadow-balance problem in M-1 is initialisation and migration, not a missed hook.)
10. **ReserveAttestationRegistry correctly rejects replayed and backdated attestations.** `asOf <= _asOf` reverts. The monotonicity gap is entirely on the *upper* side (C-2).
11. **ReserveCapModule.setCompliance being one-shot does prevent cross-instrument accounting drift**, as the NatSpec claims. It is the same property that creates the migration trap in H-7 — the mechanism works, the lifecycle around it does not.
12. **Self-transfers and zero-amount transfers do not corrupt MaxHoldersModule.** `moduleTransferAction(X, X, n)` credits then debits the same slot, leaving `shadowBalance` and `holderCount` unchanged; `_credit / _debit` both early-return on `amount == 0` (lines 86, 92), so `holderCount` cannot be inflated by dust. The zero-amount hole is in `LockupModule` only (H-1).
13. **mint to address(0) is blocked** (line 184), and `isVerified(address(0))` returns `false` anyway since `_identity[0] == 0`. `_gatedTransfer` and `forcedTransfer` both reject `to == address(0)`.
14. **No ETH handling, no delegatecall, no assembly, no selfdestruct, no call{value:}** anywhere in `src/`. Confirmed by `grep`. There is no storage-collision, proxy-initialisation or forced-ether surface.
15. **canTransfer and _gatedTransfer agree.** Both check pause, both freezes, unfrozen balance, both verifications and compliance, in a consistent order. The dry-run is a faithful predictor (modulo the gas-exhaustion case in H-4).
16. **DGXModularCompliance.bindToken and the onlyToken modifier correctly prevent a third party from forging state-hook calls.** Only the bound token can move module state.

Recommended remediation order

1. **C-1** — make `ReserveCapModule` read `token.totalSupply()`. This single change also closes **H-3** and most of **H-6**, and is the change an allocator's engineer will look for first.
2. **C-3** — one line (`if (claimTopic != topic) continue;`) plus key-purpose proof in `registerIdentity`.
3. **C-2** — bound `asOf`, add k-of-n attestors, add an owner emergency reset.
4. **H-5** — add `Ownable2Step` to all nine contracts and hand ownership to a Safe + timelock at deployment. Nothing else matters if the deploy key permanently holds the perimeter.
5. **H-1, H-2, H-7** — zero-amount guard, `canTransfer` in `forcedTransfer`, binding assertions in `addModule / setCompliance`.
6. **H-8 / H-9** — decide whether `AtomicSettlementDvP` and `DGXBullionToken` ship at all. As written, their NatSpec claims materially exceed their behaviour, which is a disclosure problem before it is a security one.

Suggested additions to the invariant suite, mapped to the findings the current 40 checks miss: module remove/re-add and compliance swap (C-1); `asOf > block.timestamp` (C-2); a non-compliant ONCHAINID returning cross-topic claims (C-3); `transfer(victim, 0)` before/after `unlockAt` (H-1); `forcedTransfer` at the holder cap and into a closed

country (H-2); a reentrant mock module (H-3); an issuer with an unbounded loop rather than a `require(false)` (H-4); `addModule` without `setCompliance` (H-7); adding `MaxHoldersModule` after minting (M-1); `recoveryAddress` with `lostWallet == newWallet` and with a pre-frozen source (M-2).