Boundary Audit Report

Prepared by Cyfrin

Version 2.2

Lead Auditors

Immeas

BengalCatBalu

January 10, 2026

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Boundary are releasing a new USD stablecoin, USBD, together with staking capabilities. USBD is minted and redeemed against whitelisted ERC-20 collateral (intended to be stablecoins) via an off-chain engine that submits EIP-712 orders to `USBDMinting`. On mint, collateral is transferred directly from the benefactor to approved custodians via a configurable route, and USBD is minted to the beneficiary. On redeem, USBD is burned from the benefactor and collateral is transferred out of the minting contract's redemption buffer to the beneficiary.

Staking is provided via `sUSBD`, an ERC-4626 vault over USBD with linear vesting of rewards transferred into the vault. It can also operate in an optional cooldown-based withdrawal mode using an `USBDSilo`.

Rewards are distributed through `StakingRewardsDistributor`, which holds collateral and uses ERC-1271 to let operators sign mint orders on behalf of the distributor, newly minted USBD is then transferred into `sUSBD` as rewards.

4.1 Actors and Roles

- **Actors:**
  - **Users:** Hold USBD, deposit into `sUSBD` for shares, redeem/withdraw (standard mode) or cooldown + claim (cooldown mode).
  - **Engine:** Off-chain executor; Submits signed mint/redeem orders on-chain; must have `MINTER_ROLE` / `REDEEMER_ROLE` on `USBDMinting`.
  - **Custodians:** Receive collateral on mint via routes and custody funds off-chain (capital efficiency / trading operations).
  - **Operators:** Sign mint orders for `StakingRewardsDistributor` (ERC-1271) and push rewards to stakers.
  - **Admin** / **Operational Admin** / **Guardian:** Configure assets/limits/status, manage roles, and perform emergency revocations.

- **Key Roles**
  - **USBDMinting**
    - `MINTER_ROLE` / `REDEEMER_ROLE`: execute `mint` / `redeem`
    - `OPERATIONAL_ADMIN_ROLE`: assets, custodians, limits, status
    - `BENEFACTOR_MANAGER_ROLE`: manage approved benefactors
    - `GUARDIAN_ROLE`: emergency disable/revoke
  - **sUSBD**
    - `REWARDER_ROLE`: `transferInRewards`
    - `OPERATIONAL_ADMIN_ROLE`: cooldown/vesting/treasury
    - `RESTRICTION_MANAGER_ROLE` / `REDISTRIBUTOR_ROLE`: restrictions + redistribution
  - **StakingRewardsDistributor**
    - `OPERATOR_ROLE`: `transferRewards`, and signatures validated for ERC-1271
    - `OPERATIONAL_ADMIN_ROLE`: minting contract + minting asset approvals

  All core contracts use an AccessControl pattern with a delayed default-admin transfer and enumerable role membership.

4.2 Key Components

- **USBD (ERC-20):** Stablecoin token; `USBDMinting` is granted mint authority in the deployment flow.
- **USBDMinting:** EIP-712 order execution, allowlisted assets/custodians/benefactors, price-deviation checks, and per-block/global limits. Supports EOA signatures, delegated signers, and ERC-1271 for contract wallets.
- **sUSBD (ERC-4626 vault):** USBD staking shares; supports:
  - **Standard mode:** direct `withdraw`/`redeem` when cooldown is disabled
  - **Cooldown mode:** burn shares into `USBDSilo` then `claim` after the timer
  - **Linear vesting:** rewards increase share value gradually; disallows overlapping vesting periods
- **USBDSilo:** Holds USBD during cooldown; only the vault can withdraw from it.
- **StakingRewardsDistributor:** Holds collateral; uses ERC-1271 to validate operator signatures for mint orders, then transfers minted USBD into `sUSBD` as rewards.

4.3 Core Flows

- Mint (collateral -> USBD)
  1. User obtains a quote off-chain, signs an EIP-712 mint order (benefactor/beneficiary/amounts/expiry/nonce).
  2. Engine (with `MINTER_ROLE`) submits `mint(order, route, sig)`; contract verifies signature + nonce + allowlists + deviation + limits.
  3. Collateral is transferred from benefactor directly to custodian destinations per route; USBD is minted to beneficiary.
- Redeem (USBD -> collateral)
  1. User signs redeem order; engine (with `REDEEMER_ROLE`) submits.
  2. USBD is burned from benefactor; collateral is transferred from the minting contract's buffer to beneficiary.
- Stake (USBD -> sUSBD)
  - Users deposit/mint USBD into `sUSBD` to receive shares at the prevailing exchange rate.

- Unstake
  - Standard mode (cooldown disabled): users can `withdraw`/`redeem` directly.
  - Cooldown mode (cooldown enabled):
  1. `cooldownAssets`/`cooldownShares` burns shares and transfers corresponding USBD to `USBDSilo` (timer starts).
  2. After cooldown ends, `claim` withdraws USBD from the silo.
- Rewards distribution (custody yield -> sUSBD)
  1. Custody sends stable collateral to `StakingRewardsDistributor`.
  2. Operator signs a mint order where benefactor/beneficiary is the distributor; engine submits to `USBDMinting`, which calls distributor's ERC-1271 `isValidSignature`.
  3. Distributor calls `transferRewards` to push USBD into `sUSBD` via `transferInRewards`, starting a new vesting period; rewards vest linearly and cannot overlap.
- Restrictions & redistribution (sUSBD)
  - Addresses can be SOFT/HARD/FULL restricted for deposit/withdraw/transfer; FULL restriction enables redistribution (burn-to-all-stakers or transfer-to-treasury).
- Rescue / Excess
  - `USBDMinting`: can transfer excess collateral from its redemption buffer to custodians.
  - `sUSBD` and `StakingRewardsDistributor`: can rescue accidentally sent tokens (per role policy).

4.4 Centralization Risk

The protocol is operationally centralized around privileged roles and off-chain components:

- The off-chain engine controls order submission and last-look execution; the protocol depends on it being correctly permissioned and operated.
- Custodians receive collateral directly on mint and custody funds off-chain, so operational trust and monitoring are critical.
- Admin/operational admin roles can change minting configs (assets, limits, status), staking parameters (cooldown/vesting), and distributor minting approvals; guardians can revoke in emergencies.

We recommend ensuring these privileges are controlled via properly secured multisigs and that robust operational best practices are followed (e.g., key management, access reviews, monitoring, and incident response).

5 Audit Scope

The audit scope was limited to:

```
src/access/AccessControlDefaultAdminRulesEnumerable.sol
src/minting/USBDMinting.sol
src/rewards/StakingRewardsDistributor.sol
src/staking/sUSBD.sol
src/staking/USBDSilo.sol
src/token/USBD.sol
// deployment-related
script/initial/Deploy.s.sol
src/deployers/DeploymentOrchestrator.sol
src/deployers/sUSBDDeployer.sol
src/deployers/USBDDeployer.sol
```

6 Executive Summary

Over the course of 9 days, the Cyfrin team conducted an audit on the Boundary smart contracts provided by Boundary Finance. In this period, a total of 12 issues were found.

During the audit of we identified three low-severity findings, primarily centered on the sUSBD staking vault's reward vesting mechanics and the operational restriction model: (i) unvested rewards becoming effectively unclaimable if all stakers exit during an active vesting period, (ii) `FULL` restriction being bypassable via pre-restriction transfers (an operational/mempool constraint), and (iii) updating `vestingPeriod` retroactively reclassifying previously vested rewards as unvested, which could lead to temporary reverts and unfair reward timing effects.

In addition, we reported nine informational items, including documentation mismatches, integration gotchas (delegated signing vs ERC-1271), and several "behavioral clarity" points (e.g., cooldown disablement instantly unlocking pending cooldown funds, ERC-4626 allowance semantics during cooldown mode, and ownership transfer to a restricted address).

Boundary addressed a subset of the above during remediation (including fixes for the "no active stakers" reward transfer condition and the `vestingPeriod` update behavior), while other items were acknowledged as intentional design choices or operationally mitigated (e.g., private mempool usage for admin restriction actions).

During the audit two additional changes were done:

- Commit `18eff23`: Adding two view functions to `USBDMinting` (`getCustodians` and `getBenefactors`).
- Commits `a88aee4`, `f8dd7f6`, and `e955a3b` changing the name of `sUSBD::unstake` to `claim`

Both of these changes were reviewed and deemed safe.

The Cyfrin team also developed an invariant test suite which was added to the Boundary code base in commits `9c84a1e`, `21194df` and `700253c`.

Summary

Project Name	Boundary
Repository	boundary-protocol-ethereum
Commit	99d752a795c0...
Fix Commit	b6fe3116aa1d...
Audit Timeline	Dec 29th - Jan 8th, 2026
Methods	Manual Review, Invariant testing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	3
Informational	9
Gas Optimizations	0
Total Issues	12

Summary of Findings

[L-1] Unvested amount is lost if all users withdraw	Resolved
[L-2] `FULL` restriction can be bypassed	Acknowledged
[L-3] Updating `vestingPeriod` can retroactively reclassify vested rewards as unvested	Resolved
[I-1] Add minimum deposit to mitigate vault manipulation edge cases	Acknowledged
[I-2] Incorrect information in redeem-documentation	Resolved
[I-3] Delegate cannot remove delegation issued by a benefactor	Resolved
[I-4] Smart-account benefactors cannot use ERC-1271 signature validation and delegated signers at the same time	Resolved
[I-5] Residual permissions after benefactor removal	Resolved
[I-6] Cooldown Deactivation Instantly Unlocks Pending Cooldown Funds	Resolved
[I-7] ERC-7702 Benefactors Are Not Supported by Signature Verification Logic	Resolved
[I-8] Cooldown Withdrawals Do Not Support ERC4626 Allowance-Based Owner Flow	Acknowledged
[I-9] Ownership can be transferred to a restricted address	Acknowledged

7 Findings

7.1 Low Risk

7.1.1 Unvested amount is lost if all users withdraw

Description: `sUSBD` permanently locks an initial "burnt" share balance (minted to the vault itself for donation-attack protection). During a vesting period, if all user shares are redeemed/burned such that only the locked shares remain, the remaining unvested rewards continue vesting into `totalAssets()` and are effectively attributed to the locked shares. As a result, rewards that were intended for stakers become permanently unclaimable.

Impact: A portion (up to all) of unvested rewards can be irreversibly "lost" if users fully exit during an active vesting period. This will strand reward funds inside the vault, accruing to the locked share balance that cannot be redeemed.

Proof of Concept: Add the following test to `test/unit/staking/sUSDB.Withdraw.t.sol`:

```
function test_VestedAmountIsLostIfAllUsersWithraw() public {
    // First set cooldown to 0 to enable direct withdraw/redeem
    vm.prank(operationalAdmin);
    isusbdMgmt.setCooldownDuration(0);

    vm.prank(rewarder);
    isusbd.transferInRewards(100e18);

    // user withdraws after half the vesting period
    vm.warp(block.timestamp + 7 days);

    vm.prank(user1);
    uint256 assets = isusbd.redeem(500e18, user1, user1);

    assertApproxEqAbs(assets, 550e18, 1e18);

    // rest of the vesting period goes
    vm.warp(block.timestamp + 7 days);

    uint256 burntAssets = isusbd.convertToAssets(isusbd.balanceOf(address(isusbd)));

    // rest of the vested amount is vested to the initial burnt shares
    assertApproxEqAbs(burntAssets, 51e18, 1e18);
}
```

Recommended Mitigation: Consider one of the following options:

- Maintain a significant protocol deposit at all time (significantly above the locked shares), which will ensure there's always at least one user.
- Add a "sweep" mechanism to sweep the locked shares to the treasury (`if(totalSupply() == 1e18) sweepAssets = convertToAssets(1e18) - 1e18)`)

Boundary: Resolved in PR#157 added a check for active stakers in transferInRewards and lowered the initial burnt deposit to 1e12. Additionally, we will initiate and maintain a protocol deposit significantly above the locked shares to ensure there is always at least one staker.

Cyfrin: Verified. `transferInRewards` reverts if no stakers.

7.1.2 `FULL` restriction can be bypassed

Description: `sUSBD` enforces `RestrictedStatus.FULL` by blocking *transfers* in `_update()` only when both `from != address(0)` and `to != address(0)`. However, restriction is applied per-address and is not "sticky" to the shares themselves. If an account anticipates being set to `FULL`, it can front-run the `setRestrictedStatus(account, FULL)` transaction by transferring its sUSBD shares to a fresh address that is not restricted. Once the restriction

update lands, the original address is frozen, but the shares have already been moved and remain fully usable from the new address.

Impact: `FULL` restriction can be bypassed.

Proof of Concept:

1. `victim` holds `N` sUSBD shares.
2. `RESTRICTION_MANAGER_ROLE` submits `setRestrictedStatus(victim, FULL)` to the mempool.
3. `victim` observes the pending tx and submits `transfer(newAddr, N)` with higher priority (front-runs).
4. `transfer` succeeds because `victim` is not yet `FULL` (and `newAddr` is unrestricted).
5. `setRestrictedStatus(victim, FULL)` executes afterward; `victim` is frozen but now holds `0` shares. The shares are controlled by `newAddr` and can be withdrawn/transferred normally.

Recommended Mitigation: Full mitigation of this issue requires significant reworking. For example, a short lock could be introduced during transfer shares. Another possible solution is to ensure that any transactions blocking users is done through flashbots / private mempool and have this properly documented.

Boundary: Acknowledged. We recognize this as an operational challenge. Administrative transactions will be submitted through private mempools to prevent front-running.

7.1.3 Updating `vestingPeriod` can retroactively reclassify vested rewards as unvested

Description: `sUSBD` tracks an active vesting "epoch" using `vestingAmount` and `lastRewardTimestamp`, while `sUSBD::getUnvestedAmount` computes remaining unvested rewards using the current `vestingPeriod`. If `vestingPeriod` is updated after rewards have been transferred in, the vault can retroactively change how much of that epoch is considered "unvested" (even after it was previously fully vested). Since `totalAssets()` is computed as `balanceOf(this) - getUnvestedAmount()`, this can (i) cause `totalAssets()` to underflow and revert when `getUnvestedAmount()` exceeds the vault's current balance, and/or (ii) temporarily suppress `totalAssets()` without reverting, allowing early redeemers to receive less while later redeemers benefit as the "resurrected" unvested amount decays over time.

Impact: * **DoS / broken ERC4626 accounting:** If `getUnvestedAmount() > vaultBalance`, calls depending on `totalAssets()` (and conversions/withdrawals) can revert (underflow) until sufficient time passes.

- **Unfair reward distribution:** If `0 < totalAssets() < vaultBalance`, the share price is temporarily reduced, so redeemers during that window receive fewer assets, while remaining holders later receive more as the same epoch "re-vests," effectively redistributing rewards based on timing rather than share ownership during the reward period.

Proof of Concept: Add the following test to `test/unit/staking/main/sUSBD.GetUnvestedAmount.t.sol`, it shows how changing the vesting period can re-activate vesting and cause DoS when `totalAssets` is called:

```
function test_GetUnvestedAmount_CanResurrectAfterVestingPeriodIncrease_AndMakeTotalAssetsUnderflow()
    public
{
    // Enable direct withdraw/redeem path (so user can exit fully)
    vm.prank(operationalAdmin);
    isusbdMgmt.setCooldownDuration(0);

    vm.prank(operationalAdmin);
    isusbdMgmt.setVestingPeriod(1);

    // Start a vesting epoch
    vm.prank(rewarder);
    isusbd.transferInRewards(REWARD_AMOUNT);

    // Let rewards fully vest under the current vestingPeriod
    vm.warp(block.timestamp + _vault.vestingPeriod());
    assertEq(isusbd.getUnvestedAmount(), 0);
```

```
    // User exits, pulling out vested rewards (leaving only the permanently locked shares behind)
    vm.startPrank(user1);
    isusbd.redeem(isusbd.balanceOf(user1), user1, user1);
    vm.stopPrank();
    // Increase vesting period AFTER vesting completed. Because getUnvestedAmount() uses the *current*
    // vestingPeriod with the same vesting epoch state, this can "resurrect" a non-zero unvested amount.
    vm.prank(operationalAdmin);
    isusbdMgmt.setVestingPeriod(uint24(14 days));

    uint256 resurrectedUnvested = isusbd.getUnvestedAmount();
    uint256 vaultBal = IERC20(_vault.asset()).balanceOf(address(isusbd));

    // The problematic condition: unvested becomes larger than the vault's actual balance
    assertGt(resurrectedUnvested, vaultBal);

    // totalAssets() does (balance - unvested) -> underflow panic (0x11)
    vm.expectRevert(stdError.arithmeticError);
    isusbd.totalAssets();
}
```

Together with these two imports:

```
+ import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
+ import { stdError } from "forge-std/StdError.sol";
```

Recommended Mitigation: Consider setting `vestingAmount = 0` when updating the vesting period:

```
    function setVestingPeriod(
        uint24 period
    ) external onlyRole(OPERATIONAL_ADMIN_ROLE) {
        if (period > MAX_VESTING_PERIOD) revert ExceedsMax();
        if (period == 0 && cooldownDuration == 0) revert ZeroCooldownAndPeriod();
        if (getUnvestedAmount() != 0) revert VestingInProgress();

        uint24 oldPeriod = vestingPeriod;
        vestingPeriod = period;
+       vestingAmount = 0;

        emit VestingPeriodUpdated(msg.sender, oldPeriod, period);
    }
```

Boundary: Resolved. Fixed in PR#181 as recommended by resetting `vestingAmount` to 0 when updating the vesting period.

Cyfrin: Verified. Recommended fix implemented.

7.2 Informational

7.2.1 Add minimum deposit to mitigate vault manipulation edge cases

Description: Consider adding basic input/rounding validations to reduce exposure to common ERC4626 vault-manipulation edge cases. In particular: enforce a small minimum deposit (e.g., `>= 1e18` USBD). This check help prevent dust/rounding-based griefing outcomes that can be leveraged in manipulation attempts or to create confusing user-facing behavior.

Boundary: Acknowledged. The vault already burns `1e12` shares to itself at deployment, which permanently protects against inflation/donation attacks and ensures dust deposits cannot round to zero shares. Combined with existing `ZeroAssets` and `ZeroShares` checks in `_deposit` and `_withdraw`, minimum deposit enforcement is redundant.

7.2.2 Incorrect information in redeem-documentation

Description: Currently, the protocol documentation file `03-Fund-Flows` section `B.Redeeming` states that in order to execute `redeem`, `benefactor` must send USDB to the `USDBMinting` contract. However, this is not implemented in the code. The code calls the `USDB::burnFrom` function during the `redeem` process, which means that USDB must be held by the `benefactor`, but an `allowance` must be issued to the `USDBMinting` address.

Boundary: Resolved. Documentation corrected in PR#162.

Cyfrin: Verified.

7.2.3 Delegate cannot remove delegation issued by a benefactor

Description: A delegate cannot remove delegation issued by a benefactor. Once delegation has been accepted by calling the `acceptDelegatedSigner` function, it can only be revoked by the benefactor by calling the `removeDelegatedSigner` function.

This contradicts the documentation, which states in the `05-Mint-and-Redeem` file that both roles can revoke delegation. Thus, this is a lack of functionality for the delegate.

```
Delegated Signing - Both EOA and contract benefactors can delegate signing to an approved EOA.
↪  Delegation requires two steps: benefactor calls initiateDelegatedSigner, delegate calls
↪  acceptDelegatedSigner. Either party can remove the delegation.
```

Recommended Mitigation: Add functionality to remove delegation for delegate

Boundary: Resolved. Documentation corrected in PR#165 - contract behavior is correct, benefactors manage their delegates.

Cyfrin: Verified.

7.2.4 Smart-account benefactors cannot use ERC-1271 signature validation and delegated signers at the same time

Description: In `USBDMinting::_verifySignature`, a smart-account benefactor cannot use its own ERC-1271 signature logic and delegated signers simultaneously. As soon as any delegate is added to `_delegatesPerBenefactor[benefactor]`, signature verification for that benefactor switches to the ECDSA path and ERC-1271 validation is no longer used. This makes ERC-1271–based validation mutually exclusive with delegated signing for smart-account benefactors.

Consider allowing smart-account benefactors to keep ERC-1271 validation active even when delegates exist, or explicitly document this limitation as an intentional design constraint.

Boundary: Resolved. Intentional - contract benefactors choose one signature mode. Documentation clarified in PR#167.

Cyfrin: Verified.

7.2.5 Residual permissions after benefactor removal

Description: When a benefactor is removed via `removeBenefactor`, previously configured beneficiaries and delegated signers for that benefactor are not cleared. If the same address is later re-added as a benefactor, all previously approved beneficiaries and delegates automatically become active again. This behavior may be unexpected, as removing a benefactor does not fully revoke prior trust relationships.

Consider clearing all beneficiaries and delegated signer data when a benefactor is removed, or explicitly document that removing a benefactor only disables mint/redeem temporarily and that prior approvals persist if the benefactor is re-added.

Boundary: Resolved. Intentional design - benefactors manage their own configuration which persists independently of approval status. Documentation clarified in PR#169.

Cyfrin: Verified.

7.2.6 Cooldown Deactivation Instantly Unlocks Pending Cooldown Funds

Description: The documentation does not clearly specify the behavior of funds that are already in the cooldown state when `cooldownDuration` is set to 0. In the current implementation, disabling cooldown allows users with pending cooldown balances in the silo to immediately withdraw their funds via `unstake()`, even if their original `cooldownEnd` timestamp has not been reached. This behavior may be non-obvious to integrators and users relying on the documentation, as it effectively cancels all ongoing cooldowns.

Recommended Mitigation: Explicitly document that setting cooldownDuration to 0 immediately unlocks all pending cooldown balances and allows instant withdrawal through unstake().

Bounded: Resolved. Documentation clarified in PR#170.

Cyfrin: Verified.

7.2.7 ERC-7702 Benefactors Are Not Supported by Signature Verification Logic

Description: The signature verification logic in `USBDMinting` relies on `address.code.length` to distinguish EOAs from contract accounts. If `benefactor.code.length == 0`, `ECDSA` signatures are accepted; otherwise, the contract falls back to ERC-1271 validation unless delegated signers are configured.

With `ERC-7702`, an EOA can have non-empty code (delegation indicator) while still behaving as a transaction-initiating account. In this case, a benefactor using `ERC-7702` without protocol-level delegated signers will be treated as a contract account and forced through `ERC-1271` validation. If the delegated logic does not explicitly implement `ERC-1271` signature validation, otherwise valid `ECDSA` signatures from the benefactor will be rejected.

As a result, ERC-7702-based benefactors are effectively unsupported unless their delegated contract explicitly handles signature validation in a compatible way. This behavior is implicit and not documented.

Recommended Mitigation: Avoid relying on address.code.length as an EOA/contract discriminator for signature handling, or explicitly document that ERC-7702 benefactors must implement ERC-1271-compatible signature validation (or use delegated signers) to interact with the protocol.

Boudary: Resolved. We align with OpenZeppelin's stance on this matter (see Issue 5707). When a user delegates their EOA via ERC-7702, they explicitly opt into that contract's behavior. Bypassing their delegation choice to force ECDSA verification would override their intent. Documentation clarified in PR#178.

Cyfrin: Verified.

7.2.8 Cooldown Withdrawals Do Not Support ERC4626 Allowance-Based Owner Flow

Description: `ERC-4626` specifies that withdraw and redeem MUST support a flow where shares are burned from an owner address when `msg.sender` has sufficient ERC-20 allowance over the owner's shares. In this implementation, the standard `withdraw` and `redeem` functions correctly implement this requirement when `cooldownDuration == 0`. However, when cooldown mode is enabled, these functions are explicitly disabled and users must instead use `cooldownAssets` or `cooldownShares`.

The cooldown withdrawal functions do not provide an owner parameter and always burn shares from `msg.sender`, making allowance-based withdrawals impossible during cooldown mode. As a result, while the contract technically implements the ERC-4626 requirement in normal operation, this behavior is not preserved under cooldown conditions.

Recommended Mitigation: Either extend the cooldown withdrawal logic to support an explicit owner parameter with allowance checks, or clearly document that cooldown mode intentionally restricts withdrawals to be initiated only by the share owner.

Boundary: Acknowledged. The owner parameter is omitted to prevent a griefing attack vector.

7.2.9 Ownership can be transferred to a restricted address

Description: `sUSBD` prevents restricting the current `owner()` and `treasury` in `sUSBD::setRestrictedStatus`, and it also prevents setting `treasury` to an address with a non-`NONE` restricted status in `sUSBD::setTreasury`.

However, there is no equivalent guard on ownership changes: the owner can transfer ownership to an address that is already restricted (e.g., `SOFT/HARD/FULL`). If that happens, the restriction on the new owner cannot be lifted via `setRestrictedStatus`, because `setRestrictedStatus` explicitly rejects updates when `account == owner()`. As a result, the only way to "unrestrict" that owner address would be to transfer ownership again to a different (non-restricted) address.

Consider overriding/guarding the ownership transfer flow (e.g., `transferOwnership`) to require `restrictedStatuses[newOwner] == RestrictedStatus.NONE`, aligning ownership changes with the protocol's restriction policy.

Boundary: Acknowledged. Restriction status does not affect administrative operations, so additional guards are unnecessary.