

OpenStocks / USDOS Protocol

Scoped, fuzz-driven review of the six trade-module contracts: the inventory minter, the primary market, the merger and acquisition resolver, the pre-IPO conversion registry, the tokenized-stock ERC20, and its factory.

Report title	OpenStocks / USDOS. Trade-Module Security Audit (dev-3)
Assessment type	Manual source review, Foundry fuzzing and stateful invariant testing, proo of concept exploitation, adversarial re-verification
Codebase	openstocks_contracts-dev (dev-3 snapshot), Solidity 0.8.28 (via_ir, optimizer), BSC target
Scope	Strictly contracts/trade: OpenStocksMinter, OpenStockPrimaryMarket, OpenStockMergerAcquisition, OpenStockIPOConversion, OpenStockTokenizer, OpenStockTokenizerFactory (2,210 lines)
Out of scope	All contracts outside contracts/trade, and the external price oracle
Findings	0 Critical, 0 High, 4 Medium, 6 Low, 4 Informational (14 total)
Verification	108 proo of concept tests, all passing; fuzz runs = 512; stateful invariant runs = 256, depth = 32 (roughly 8,192 calls per campaign)
Report date	26 August 2026

1 Detailed Findings

01 Redemption minimum-price guarantee is enforced on gross proceeds, not the net amount paid

Severity	Category	Class	Status
MEDIUM	Logic / user-protection bypass	Privileged (operator), defeats a user guarantee	Open

DESCRIPTION

In OpenStockPrimaryMarket, a redeemer sets minimumPriceUSDC as the lowest acceptable execution price per whole token. On settlement the operator reports gross proceeds and separate execution and acquisition costs, and the requester is paid the net of those. The minimum-price check is computed on gross proceeds, but the receiver is paid net, so the operator-attested costs sit entirely outside the guarantee. The costs are bounded only by being less than gross, so the receiver can be paid as little as one wei while the guard passes.

AFFECTED CODE

contracts/trade/OpenStockPrimaryMarket.sol:299-305 (guard on grossProceedsUSDC), :293-296 and :315 (payout of netProceeds)

```
uint256 netProceeds = grossProceedsUSDC - totalCosts;          // paid to the receiver
uint256 actualPrice = Math.mulDiv(grossProceedsUSDC, 10 ** config.tokenDecimals,
                                liquidatedAmount, Math.Rounding.Floor);
if (actualPrice < request.minimumPriceUSDC) revert MinimumPriceNotMet(); // checks GROSS
...
_pushExact(usdc, request.proceedsReceiver, netProceeds);      // receiver gets NET, not gross
```

MECHANISM

actualPrice is derived from grossProceedsUSDC, so the guard only bounds the gross execution price. executionCostsUSDC and acquisitionCostsUSDC are operator-attested with no upper bound other than being less than gross. An operator can set gross exactly at the requester's minimum, so the guard passes, and set costs equal to gross minus one, so the receiver is paid one wei and the remainder is routed to the treasury as costs. The requester's per-token minimum therefore provides no guarantee on the amount actually received.

STEPS TO REPRODUCE

1. A redeemer requests a large-block redemption with minimumPriceUSDC set to their floor price per token.
2. The operator settles with grossProceedsUSDC equal to exactly minimumPriceUSDC times the liquidated amount, so the gross price guard passes at the boundary.
3. The operator sets executionCostsUSDC to grossProceedsUSDC minus one, which is permitted because it is less than gross.
4. The receiver is paid one wei of net proceeds and the treasury receives the rest, while the request is marked Settled.

PROOF OF CONCEPT

test-foundry/trade/round3/primary/PM_MinPriceNetGap.t.sol::test_receiverStarved_whilePriceGuardPasses (PASS); rounding sub-hypothesis disproven by ::testFuzz_priceGuard_lowerBoundsGrossPrice (512 runs)

```
uint256 gross = (MIN_PRICE * AMOUNT) / 1e18;                // actualPrice == MIN_PRICE exactly
market.settleRedemption(id, AMOUNT, gross, gross - 1, 0, ref); // execution cost = gross - 1
assertEq(bobGain, 1, "receiver got 1 wei net");
```

```
assertEq(treGain, gross - 1, "treasury took the rest");
assertLt((bobGain * 1e18) / AMOUNT, MIN_PRICE, "net price far below stated minimum");
// negative control: executionCosts = 0 -> receiver paid the full gross at exactly MIN_PRICE
```

IMPACT

The user-facing minimum-price protection is unenforceable on realised proceeds. A rogue or misconfigured operator can route nearly all proceeds to the treasury while passing the guard, and even an honest operator with legitimately high costs will pay the requester below the minimum they set. Trigger requires the operator role, but the defect is that the contract does not enforce the guarantee it advertises.

RECOMMENDED FIX

Enforce the minimum against net proceeds, for example require $\text{floor}(\text{netProceeds} \times 10^{\text{decimals}} / \text{liquidatedAmount})$ to be at least `minimumPriceUSDC`. Additionally cap the sum of execution and acquisition costs as a fraction of gross, or let the requester bound the acceptable costs at request time.

02 Legacy redemption migration reserves tokens the contract does not hold

Severity	Category	Class	Status
MEDIUM	Accounting / stuck funds	Privileged, upgrade-scoped (one-time owner action)	Open

DESCRIPTION

When a proxy is upgraded from the pre-reservation implementation, the owner runs `migrateLegacyRedemptions` to copy each legacy record's amount into the new `reservedOpenStocks` accounting. The migration adds to the reservation without checking that the contract actually holds enough tokens to back it. The legacy implementation is precisely the version that lacked reservation accounting and could sell or sweep escrowed tokens, so a shortfall at upgrade time is realistic. If any legacy escrow was already gone, the reservation exceeds the balance and the ring-fence is broken from the moment accounting is activated.

AFFECTED CODE

```
contracts/trade/OpenStocksMinter.sol:628 (unchecked reserve increment), :641-647
(finalizeLegacyRedemptionMigration)
```

```
for (uint256 redemptionId = legacyRedemptionsMigrated + 1; redemptionId <= end; ++redemptionId) {
    PendingRedemption storage redemption = pendingRedemptions[redemptionId];
    ...
    redemption.status = RedemptionStatus.Pending;
    reservedOpenStocks[redemption.openstockToken] += redemption.openstockAmount; // no backing
check
}
// finalizeLegacyRedemptionMigration() then sets redemptionAccountingReady = true unconditionally
```

MECHANISM

Migration blindly sums the legacy amounts into `reservedOpenStocks`. `finalize` activates normal operation without verifying that `reservedOpenStocks[token]` is backed by the contract's balance. Once reserved exceeds balance, the invariant that guarantees every redeemer can be made whole is violated: early cancellers drain the available balance, and later legacy users' `cancelRedemption` reverts inside `escrowTransfer` on an insufficient

balance. Because reserved is greater than balance, emergencyWithdraw and emergencyWithdrawAll also revert, so the owner cannot even top the contract up through the normal recovery path.

STEPS TO REPRODUCE

1. A proxy is upgraded from the pre-reservation implementation with two legacy redemptions totalling 200 tokens, but only 150 tokens remain escrowed on-chain.
2. The owner calls migrateLegacyRedemptions and finalizeLegacyRedemptionMigration.
3. reservedOpenStocks is now 200 while the balance is 150, so the reservation exceeds the backing.
4. The first legacy user cancels and recovers 100, leaving 50 on-chain against a remaining reservation of 100. The second user's cancelRedemption reverts and their tokens are permanently stuck.

PROOF OF CONCEPT

```
test-foundry/trade/round3/minter/MinterLegacyMigration.t.sol::test_F2_migrateOverReservesUnbackedTokens (PASS); negative control ::test_F2_negControl_fullyBackedMigrationIsSafe
```

```
_fundInventory(150e18); // only 150 of 200 legacy escrow remains
minter.migrateLegacyRedemptions(2); minter.finalizeLegacyRedemptionMigration();
assertGt(minter.reservedOpenStocks(stock), stock.balanceOf(minter)); // reserved 200 > balance 150
minter.cancelRedemption(1); // user1 drains 100, leaving 50
vm.expectRevert(); // ERC20InsufficientBalance
minter.cancelRedemption(2); // user2 permanently stuck
```

IMPACT

Some legacy redeemers cannot recover their escrowed tokens, and the owner cannot sweep to make them whole, so the migration itself introduces a stuck-funds condition. The scope is limited to an upgrade from the legacy implementation with a pre-existing escrow shortfall, and the trigger is the owner, but the outcome is a broken ring-fence with no on-chain remedy.

RECOMMENDED FIX

Before finalize succeeds, require reservedOpenStocks[token] to be at most the contract's live balance of that token for every migrated token, and revert or require a top-up otherwise. Consider migrating per token and checking backing incrementally.

03 Escrow-role revocation permanently strands pending redemptions

Severity	Category	Class	Status
MEDIUM	Cross-contract availability / stuck funds	Cross-contract (token owner action)	Open

DESCRIPTION

In OpenStockPrimaryMarket every path that returns escrowed tokens to a requester (a partial settlement, an expired cancellation, and an operator rejection) uses escrowTransfer on the token, which requires the market to hold the token's ESCROW_ROLE. requestRedemption verifies that role, but none of the return paths re-verify it. If the token owner revokes the market's escrow role after a request is recorded, the return paths revert and the requester's escrowed tokens are frozen with no rescue function.

AFFECTED CODE

```
contracts/trade/OpenStockPrimaryMarket.sol:314 (partial-settlement return), :354
(_cancelAndReturn); depends on OpenStocksTokenizer.sol:163 (escrowTransfer is
onlyRole(ESCROW_ROLE))
```

```
// every return path is hard-wired to escrowTransfer, which needs ESCROW_ROLE on the token:
if (returnedAmount != 0) IOpenStockTokenizer(request.token).escrowTransfer(request.requester,
returnedAmount);
...
function _cancelAndReturn(...) internal {
    IOpenStockTokenizer(request.token).escrowTransfer(request.requester, request.requestedAmount);
}
```

MECHANISM

escrowTransfer is gated by ESCROW_ROLE on the token contract. requestRedemption checks the role through _requireTokenActive, but settleRedemption, cancelExpiredRedemption and rejectRedemption never re-check it and have no alternative return mechanism. A revocation of the market's escrow role, which is a realistic migration or incident-response action on the token, makes cancelExpiredRedemption and rejectRedemption revert, and makes any partial settlement revert as well. The requester has no self recovery path.

STEPS TO REPRODUCE

1. A KYC-approved user requests a redemption and their tokens are escrowed in the market.
2. The token owner revokes the market's ESCROW_ROLE on the token, for example during a migration or incident response.
3. The redemption deadline passes and the user calls cancelExpiredRedemption, which reverts with an access-control error.
4. The escrowed tokens remain frozen in the market with no on-chain path to return them.

PROOF OF CONCEPT

```
test-foundry/trade/round3/primary/PM_EscrowRoleDoS.t.sol::test_cancelExpired_reverts_afterEscrowRoleR
evoked (PASS); negative control ::test_cancelExpired_succeeds_whenEscrowRoleIntact; scope isolator
::test_fullSettle_succeeds_withoutEscrowRole
```

```
token.revokeEscrowRole(address(market)); // token owner revokes the role
vm.warp(deadline + 1);
vm.expectRevert(); // AccessControlUnauthorizedAccount(market,
ESCROW_ROLE)
market.cancelExpiredRedemption(id);
assertEq(token.balanceOf(address(market)), AMOUNT); // frozen
assertEq(token.balanceOf(alice), 0); // user cannot recover
```

IMPACT

In-flight redemptions become unrecoverable during any event that removes the market's escrow permission on a token. User principal is frozen indefinitely until the role is re-granted. No funds are stolen, and full settlement without a return still works, which isolates the failure to the escrow-dependent return path.

RECOMMENDED FIX

Prefer a plain transfer when the token is not paused and fall back to escrowTransfer only when necessary, add an owner rescue path for pending redemptions that does not route through escrowTransfer, or operationally gate escrow-role revocation on there being no outstanding pending requests.

04 Escrow-role revocation bricks cancelRedemption and strands escrowed tokens

Severity	Category	Class	Status
MEDIUM	Cross-contract availability / stuck funds	Cross-contract (token owner action)	Open

DESCRIPTION

OpenStocksMinter.cancelRedemption returns a user's escrowed tokens through escrowTransfer, which requires the minter to hold the token's ESCROW_ROLE. If the token owner revokes that role after a redemption is recorded, cancelRedemption reverts permanently and the user has no self recovery path. This is the same cross-contract dependency as 03, in the minter rather than the primary market.

AFFECTED CODE

contracts/trade/OpenStocksMinter.sol:587 (cancelRedemption return via escrowTransfer); depends on OpenStocksTokenizer.sol:163

```
function cancelRedemption(uint256 redemptionId) external nonReentrant redemptionAccountingIsReady {  
    ...  
    reservedOpenStocks[redemption.openstockToken] -= redemption.openstockAmount;  
    IOpenStockTokenizer(redemption.openstockToken).escrowTransfer(redemption.user,  
redemption.openstockAmount);  
} // reverts for the whole call if the minter's ESCROW_ROLE was revoked
```

MECHANISM

The only user-initiated recovery for an unpaid redemption is cancelRedemption, and it depends on escrowTransfer. The settlement path stays available but is controlled by the settler role, not the user. Once the escrow role is revoked, a user whose chain payment never arrives cannot reclaim their tokens.

STEPS TO REPRODUCE

1. A user requests a redemption and their tokens are escrowed and reserved in the minter.
2. The token owner revokes the minter's ESCROW_ROLE on the token.
3. The user waits out the cancellation delay and calls cancelRedemption, which reverts.
4. The tokens remain reserved and stuck in the minter with no user-side recovery.

PROOF OF CONCEPT

test-foundry/trade/round3/minter/MinterFindings.t.sol::test_F5_escrowRevoke_locksPendingRedemption (PASS); negative control ::test_F5_negControl_cancelWorksWithEscrowIntact; state-machine soundness ::test_F5_stateMachineTransitions

```
stock.revokeEscrowRole(address(minter));  
vm.warp(cancellableAt);  
vm.expectRevert(); // escrowTransfer reverts, cancel bricked  
vm.prank(user); minter.cancelRedemption(id);  
assertEq(minter.reservedOpenStocks(stock), 100e18); // still reserved and stuck
```

IMPACT

A user's escrowed tokens can be frozen by a role change on a separate contract, with no self recovery. Availability defect, no theft. Severity matches 03 because the root cause and the stuck-funds outcome are the same.

RECOMMENDED FIX

Apply the same remediation as 03: a non-escrow return path or an owner rescue for pending redemptions, and operational discipline around revoking the escrow role while redemptions are outstanding.

05 mint has no zero-output guard; a rounding buyer pays in full for zero tokens

Severity	Category	Class	Status
LOW	Missing input validation	Unprivileged (caller self harm)	Open

DESCRIPTION

OpenStocksMinter.mint quotes the output amount from the oracle, then transfers the full payment to the treasury and delivers the output. There is no check that the output is non-zero. If the oracle rounds the output to zero for a small payment and the caller passes a minimum-output of zero, the buyer pays the full amount to the treasury and receives nothing.

AFFECTED CODE

contracts/trade/OpenStocksMinter.sol:437-456 (no openstockAmount == 0 check between the quote and the transfers)

```
(uint256 openstockAmount, uint256 feeAmount) = calculateMintAmount(openstockToken, paymentToken, paymentAmount);
if (openstockAmount < minOpenStockAmount) revert InsufficientPayment(); // passes when both are 0
...
IERC20(paymentToken).safeTransferFrom(msg.sender, treasury, paymentAmount); // full payment leaves
IERC20(openstockToken).safeTransfer(msg.sender, openstockAmount); // delivers 0
```

MECHANISM

The slippage floor minOpenStockAmount fully mitigates this when it is set to a non-zero value, so the issue only bites a caller who submits with a zero floor and a payment small enough to round the oracle output to zero. It cannot harm any party other than the caller.

STEPS TO REPRODUCE

1. A caller submits mint with a payment amount small enough that the oracle output rounds to zero and with minOpenStockAmount set to zero.
2. The quote returns a zero output and the zero floor check passes.
3. The full payment is transferred to the treasury and the caller receives zero tokens.

PROOF OF CONCEPT

test-foundry/trade/round3/minter/MinterFindings.t.sol::test_F1_mintRoundsToZero_userReceivesNothingButPaysFull (PASS); negative control ::test_F1_negControl_minOutFloorReverts

```
// oracle rate 1/1000, paymentAmount = 999 -> quote rounds to 0 stock
vm.prank(buyer); minter.mint(stock, pay, 999, 0);
assertEq(treasuryGain, 999); assertEq(stockToBuyer, 0); assertEq(minter.totalMinted(stock), 0);
// negative control: minOpenStockAmount = 1 reverts InsufficientPayment
```

IMPACT

Loss of the dust payment for a caller who does not set a slippage floor. No impact on other users or on protocol accounting. Low because the standard slippage parameter prevents it.

RECOMMENDED FIX

Add a revert when the quoted output is zero, for example if (openstockAmount == 0) revert ZeroAmount(); after the quote.

06 transferOwnership moves admin but retains the mint role on the old owner

Severity	Category	Class	Status
LOW	Access control / handover gap	Privileged handover	Open

DESCRIPTION

OpenStockTokenizer overrides transferOwnership to move the default admin role to the new owner, but it does not move or revoke the issuer role. Because the initializer grants the deployer the issuer role, an operator who hands over the token with plain transferOwnership leaves the previous owner able to mint unlimited supply. The atomic transferOwnershipAndIssuerRole does the full handover, which makes the asymmetry a footgun.

AFFECTED CODE

contracts/trade/OpenStocksTokenizer.sol:190-196 (override moves DEFAULT_ADMIN_ROLE only), :70 (issuer granted at init), :172-181 (atomic alternative)

```
function transferOwnership(address newOwner) public override onlyOwner {
    _grantRole(DEFAULT_ADMIN_ROLE, newOwner);
    _revokeRole(DEFAULT_ADMIN_ROLE, previousOwner); // admin handed over ...
    super.transferOwnership(newOwner);
} // ... but ISSUER_ROLE on previousOwner is untouched -> old owner can still mint()
```

MECHANISM

The override deliberately re-aligns the default admin role, which leads a reader to assume a full handover, yet the issuer role granted to the deployer at initialization is left in place. The old key retains the ability to mint the stock token until the new owner explicitly revokes it.

STEPS TO REPRODUCE

1. The token is initialized, which grants the deployer both the default admin and issuer roles.
2. The operator hands the token over with plain transferOwnership.
3. The new owner holds the default admin role, but the old owner still holds the issuer role and can mint supply to any address until it is revoked.

PROOF OF CONCEPT

test-foundry/trade/round3/base/TokenizerRound3.t.sol::test_transferOwnership_retainsStaleIssuerRole (PASS); negative control ::test_transferOwnershipAndIssuerRole_clean

```
token.transferOwnership(BOB);
assertEq(token.owner(), BOB);
assertFalse(token.hasRole(DEFAULT_ADMIN_ROLE, OWNER));
assertTrue(token.hasRole(ISSUER_ROLE, OWNER)); // stale mint capability
vm.prank(OWNER); token.mint(attacker, 1_000_000e18); // old owner still mints
```

IMPACT

A prior key keeps supply-inflation capability on a tokenized security after an apparent ownership handover, until it is explicitly revoked. Fully recoverable by the new owner, so Low, but material for a security token given the surprising asymmetry.

RECOMMENDED FIX

Revoke the old owner's issuer role inside transferOwnership, or route token administration handovers exclusively through transferOwnershipAndIssuerRole and disable or restrict the plain override.

07 Owner can pause a funded cash resolution for the whole window, then reclaim it

Severity	Category	Class	Status
LOW	Centralization / failure to perform	Privileged (owner), trusted key	Open

DESCRIPTION

In OpenStockMergerAcquisition, pausing a resolution blocks resolve for its whole duration but has no effect on the deadline clock. An owner can pause a fully funded cash resolution for the entire window so that no holder can redeem, then call withdrawExpiredCash after the deadline and recover the entire funded pool. Holders keep now-worthless old tokens and receive nothing.

AFFECTED CODE

contracts/trade/OpenStockMergerAcquisition.sol:205-210 (pauseResolution), :225 (resolve pause gate), :310-322 (withdrawExpiredCash)

```
function resolve(...) { ... if (resolution.paused) revert ResolutionPaused(); ... }
// pause does not extend the deadline; after deadline the owner can reclaim everything:
function withdrawExpiredCash(address oldToken, address receiver) external onlyOwner nonReentrant {
    if (block.timestamp <= resolution.deadline) revert ResolutionWindowClosed();
    uint256 amount = resolution.cashRemaining; ... _safeExactTransfer(usdc, receiver, amount);
}
```

MECHANISM

resolve reverts while the resolution is paused, and the pause does not extend the deadline, so an owner can starve holders for the full period and then reclaim the unspent cash as if it had expired unclaimed. Holders never deposit into the contract before resolving, so no user-deposited funds are stolen; this is a trusted-operator failure to perform rather than theft.

STEPS TO REPRODUCE

1. The owner configures and fully funds a cash resolution.
2. The owner pauses the resolution before any holder resolves and keeps it paused for the whole window.
3. After the deadline the owner calls withdrawExpiredCash and recovers the entire funded pool, leaving holders unpaid.

PROOF OF CONCEPT

test-foundry/trade/round3/merger/MergerCashUnit.t.sol::test_OwnerPauseThenReclaim_StrandsHolders (PASS); negative control ::test_OwnerPause_NegControl_NoPausePaysHolders

```
vm.prank(owner); mna.pauseResolution(oldToken); // paused for the whole window
vm.warp(deadline + 1);
```

```
vm.prank(owner); mna.withdrawExpiredCash(oldToken, owner);
assertEq(reclaimed, TCP); // owner takes back 100% of funded cash
assertEq(usdc.balanceOf(alice), 0); // holder paid nothing, keeps old tokens
```

IMPACT

The owner can unilaterally deny holders their owed consideration and take back the funded cash. Bounded because no user funds are held before resolution, so it is a trusted-key concern, not a theft of deposits.

RECOMMENDED FIX

Bound the cumulative pause duration, extend the deadline by the paused time, or block `withdrawExpiredCash` when the resolution spent most of its life paused. Governing the owner key with a timelocked multisig further reduces the risk.

08 Conversion liveness depends on the pre-IPO token's mutable burn and pause flags

Severity	Category	Class	Status
LOW	Cross-contract availability	Cross-contract (pre-IPO token owner)	Open

DESCRIPTION

`OpenStockIPOConversion.convert` pulls the pre-IPO token into the registry and then burns it. Both steps depend on the pre-IPO token's configuration: the pull reverts if that token is paused, and the burn reverts if burning is disabled on it. Whoever controls the pre-IPO token can therefore brick every conversion for that token, and the frozen-ratio design means the registry cannot re-point the configuration after the IPO to recover.

AFFECTED CODE

`contracts/trade/OpenStockIPOConversion.sol:199` (`safeTransferFrom`, blocked when paused), `:204` (`burn`, blocked when `burnEnabled` is false); depends on `OpenStocksTokenizer.sol:90-95` and `:184-187`

```
preIPOTokenERC20.safeTransferFrom(msg.sender, address(this), amount); // reverts if pre-IPO token
paused
...
IOpenStockTokenizer(preIPOToken).burn(amount); // reverts if burnEnabled ==
false
```

MECHANISM

`convert` requires both a successful holder-to-holder transfer of the pre-IPO token and a successful burn. The pre-IPO token owner can disable burning or pause transfers on that token, and either toggle DoSes all conversions of it. No funds are lost because the reverts are atomic, but a permissionless user right becomes unavailable.

STEPS TO REPRODUCE

1. A conversion is configured for a pre-IPO token and its IPO has started.
2. The pre-IPO token owner disables burning or pauses transfers on that token.
3. Every `convert` call for that token reverts, and the registry cannot re-point the frozen configuration to restore service until the flag is reversed.

PROOF OF CONCEPT

```
test-foundry/trade/round3/base/IPConversionRound3.t.sol::test_DoS_burnDisabledOnPreIPO (PASS),  
::test_DoS_preIPOPaused (PASS); negative controls re-enable burn / unpause and convert succeeds
```

```
preIPO.setBurnEnabled(false);  
vm.expectRevert(); conv.convert(address(preIPO), amount, receiver); // BurningDisabled  
// negative control: setBurnEnabled(true) -> convert succeeds and mints the expected public amount
```

IMPACT

Availability of the permissionless pre-IPO to public conversion depends on the continued cooperative configuration of a separate token contract. No theft; reverts are atomic and holders keep their tokens.

RECOMMENDED FIX

Document the trust coupling, ensure the same trusted admin or timelock governs both the pre-IPO token and the conversion registry, and consider a design in which the registry holds the burn authority so liveness does not depend on a mutable flag on an external token.

09 Non-refundable processing fee enables instant-reject grieving

Severity	Category	Class	Status
LOW	Griefing / fee handling	Privileged (operator)	Open

DESCRIPTION

In OpenStockPrimaryMarket the requester pays a non-refundable processing fee to the treasury at request time. rejectRedemption has no time gate and does not refund the fee, so an operator can reject a request immediately, before any service is performed, and the requester loses the fee while getting their tokens back.

AFFECTED CODE

contracts/trade/OpenStockPrimaryMarket.sol:249-252 (fee pulled to treasury at request), :339-345 (rejectRedemption, no refund, no time gate)

```
if (config.processingFeeUSDC != 0) {  
    _pullExact(usdc, msg.sender, config.processingFeeUSDC);  
    _pushExact(usdc, treasury, config.processingFeeUSDC); // non-refundable, taken up front  
}  
// rejectRedemption returns the tokens but never refunds the fee, and has no time gate
```

MECHANISM

The fee is taken at request and is never returned. A rogue or careless operator can reject immediately, so the requester pays for a service that is never attempted. The fee goes to the treasury and not to the operator, so it is grieving rather than direct operator profit.

STEPS TO REPRODUCE

1. A requester submits a redemption and pays the processing fee to the treasury.
2. An operator calls rejectRedemption immediately, before any liquidation attempt.
3. The requester's tokens are returned but the fee is retained by the treasury with no service rendered.

PROOF OF CONCEPT

```
test-foundry/trade/round3/primary/PM_FeeGriefKyc.t.sol::test_immediateReject_userLosesFee (PASS);  
negative control ::test_negativeControl_zeroFee_noLoss
```

```
vm.prank(operator); market.rejectRedemption(id, reason);
assertEq(feeCharged, config.processingFeeUSDC); // debited from the requester
assertEq(treasuryKeepsFee, config.processingFeeUSDC); // retained despite no service
```

IMPACT

A trusted operator can burn a requester's fee without performing any work. Bounded by the operator role and by the fee size, and consistent with the fee being documented as non-refundable, but it removes any protection for a requester against a rejection before service.

RECOMMENDED FIX

Refund the fee on an operator-initiated rejection, or escrow the fee and only forward it to the treasury once a genuine settlement attempt has been made.

10 KYC is enforced only at request time, not on the settlement payout path

Severity	Category	Class	Status
LOW	Compliance gap	Privileged (operator), compliance	Open

DESCRIPTION

requestRedemption checks that both the requester and the proceeds receiver are KYC approved, but settleRedemption performs no KYC re-check. If compliance revokes a receiver's KYC after a request, settlement still pays net USDC to the now unapproved receiver. The operator can choose to reject instead, but nothing on the paying path enforces a re-check.

AFFECTED CODE

contracts/trade/OpenStockPrimaryMarket.sol:232 (KYC checked at request), :280-328
(settleRedemption performs no KYC re-check)

```
// request time:
if (!kycApproved[msg.sender] || !kycApproved[proceedsReceiver]) revert KYCRequired();
// settle time: no kycApproved check before paying proceedsReceiver
```

MECHANISM

KYC is a point-in-time check at request. Settlement, which is when USDC actually moves to the receiver, does not re-validate approval, so a receiver de-approved between request and settlement is still paid unless the operator manually rejects.

STEPS TO REPRODUCE

1. A KYC-approved user requests a redemption naming a KYC-approved proceeds receiver.
2. Compliance revokes the receiver's KYC approval after the request.
3. The operator settles the request and net USDC is paid to the now unapproved receiver.

PROOF OF CONCEPT

test-foundry/trade/round3/primary/PM_FeeGriefKyc.t.sol::test_deKycdReceiver_stillPaidOnSettle (PASS);
negative control ::test_negativeControl_kycEnforcedAtRequest

```
market.setKYCStatus(receiver, false); // compliance revokes after request
```

```
vm.prank(operator); market.settleRedemption(id, ...); // still pays the de-KYC'd receiver
assertGt(usdc.balanceOf(receiver), 0);
```

IMPACT

A compliance control can be bypassed by timing a revocation after a request is recorded. Mitigable because the operator can reject, but the paying path does not enforce the control automatically.

RECOMMENDED FIX

Re-check KYC approval for the proceeds receiver, and optionally the requester, inside `settleRedemption`, or explicitly document that settlement intentionally bypasses KYC and rely on the operator to reject.

2 Informational Observations

The following observations were confirmed with passing proof of concept tests but carry no direct security impact. They are recorded for completeness.

ID	Contract	Observation and proof of concept
11	Minter	settleRedemption marks a redemption Settled and frees its reservation with no on-chain proof of the of chain payment, and once Settled the user can no longer cancel. This is the intended of chain-settlement model; it concentrates trust in the settler role and warrants monitoring. The stored payout quote is fixed at request time and is unaffected by later oracle moves, which is also intended. POC: MinterFindings.t.sol::test_F3_settleRemovesRecourse_noOnChainProof, ::test_F3_quoteFrozenAtRequestTime .
12	MergerAcquisition	A holder whose balance is below eligibleSupply divided by totalCashProceeds floors to a zero cash payout and cannot resolve for cash. With an 18-decimal token and 6-decimal USDC the threshold is negligible, but such holders forfeit and cannot exit through this contract. POC: MergerEligibleSupply.t.sol::test_SubThreshold_ZeroOutput_Reverts, ::test_AtThreshold_NegControl_One Unit.
13	IPOConversion	convert burns the full input amount but mints a floored public amount, so the fractional remainder is destroyed with no refund. The loss grows with the ratio denominator and affects callers who pass non-aligned amounts. POC: IPOConversionRound3.t.sol::test_RoundingDust_valueLostNoRefund, characterized by ::testFuzz_conversionMathConservation.
14	Factory	setTokenImplementation checks only that the candidate has code and

		returns the standard UUPS proxiable slot. Any UUPS contract passes, including one that is not a tokenizer and lacks a compatible initialize, which would silently brick future deploys. POC: FactoryRound3.t.sol::test_setTokenImplementation_acceptsForeignUUPS_thenDeployBricks, negative control ::test_setTokenImplementation_rejectsEOAandNonUUPS.
--	--	---

3 Conclusion

The six trade-module contracts are, on the whole, carefully written. The central accounting invariants hold under fuzzing and stateful invariant testing, and the serious exploitation hypotheses were disproven with passing tests rather than left as open questions. No critical or high-severity issue was found.

The four Medium findings are worth prompt attention. 01 is a correctness defect in a user-facing guarantee: the redemption minimum price is enforced on gross proceeds while the user is paid net. 02 is a one-time migration that can break the minter's reservation ring-fence and strand legacy redeemers. 03 and 04 are the same cross-contract pattern in two contracts, where revoking an escrow role on the token permanently freezes escrowed user tokens with no rescue path. The Low and Informational items are trusted-role and by-design behaviours, reported with their trust boundary stated so the client can decide what to harden.

Verification. Every finding is backed by a deterministic Foundry proof OF concept with measured assertions and a one-variable negative control. All 108 tests reproduce together in a single clean run. I re-ran and inspected each one and confirmed every mechanism against source.