DATE

FAILSAFE

24 June 2026

MakeBanc

Smart Contract Audit

SECURITY TEST CONDUCTED BY

SMART CONTRACT AUDIT

FailSafe

Table of Contents

Executive Summary

MakeBanc is an on-chain capital connectivity infrastructure platform: Investors deposit USDC into tokenized ERC-4626 + EIP-7540 asynchronous vaults whose capital is traded off-chain on a CEX, with valuation written back on-chain and fees taken per epoch and at redemption. This report covers the on-chain `makebanc-vaults` scope — the five core contracts (`MakebancVault`, `TierManager`, `FeeManager`, `VaultsFactory`, `VaultProxy`), their interfaces, and the deployment scripts — at commit `1156124` on Base. The in-scope deployment scripts provision the production governance topology — the vaults are owned by a 96h `TimelockController` under a 2-of-3 Safe multisig, with a separate operational signer for hot-path actions — detailed in the Trust Model and System & Deployment Architecture sections.

The review combined structured threat modeling, a focused deep-dive that re-explored the code around each candidate finding, and an adversarial verification pass; several findings were reproduced with runnable Foundry proof-of-concepts that assert concrete USDC/share outcomes.

The review identifies **ten findings — two High, three Medium, five Low.** They are overwhelmingly **code-correctness and robustness defects in a pre-production system**, not adversarial exploits: the two High items are both fee-accounting defects in the vault's per-user High-Water-Mark (HWM) machinery — one over-charges a user who exits in multiple requests, and one lets a user avoid the fee by timing a withdrawal around the epoch sweep. The three Medium items are a factory/interface parameter-order mismatch, a dead fee-discount field, and cheaply-bought shares inheriting a stale fee mark. The Low items are a reverting view, a batch-settlement robustness gap, a rounding edge, a management-fee timing nuance, and a tier-renewal bug.

Developer remediations were submitted in **PR #20** and reviewed by FailSafe against the patched source; the statuses below reflect that review — **nine Resolved, one Acknowledged.**

Project Details

Project	Smart Contract Audit
System	MakeBanc Vaults — MakebancVault / TierManager / FeeManager / VaultsFactory / VaultProxy, their interfaces, and the deployment scripts (incl. the TimelockController setup)
Repository	`github.com/protofire/makebanc-vaults` (private)
Branch	`main`
Audit Commit	`115612474fd457e3e9b896f967dc7e9194049559`
Blockchain	Base (EVM)
Framework	Foundry
Language	Solidity 0.8.30 (optimizer runs 200)
Report Date	24 June 2026 (remediation review — PR #20)
Deployment	Production architecture (provisioned by the in-scope deployment scripts) places the vaults under a 96h OpenZeppelin `TimelockController` owned by a 2-of-3 Safe multisig, with a separate 1-of-1 operational Safe for the hot-path roles (see *System & Deployment Architecture*). Deployed on Base mainnet (3 vaults) and Base Sepolia.

Scope

This report audits the **`makebanc-vaults`** repository — the on-chain scope defined in the engagement proposal: **16 files, 1,112 nSLOC.**

Component	Files	nSLOC
Core contracts — `MakebancVault` (542), `TierManager` (117), `FeeManager` (78), `VaultsFactory` (62), `VaultProxy` (5)	5	804
Interfaces — `IMakebancVault`, `ITierManager`, `IVaultsFactory`, `IFeeManager`, `IPausable`	5	122
Deployment scripts — `00_DeployTimelockController`, `01_DeployTierManager`, `02_DeployVaultFactory`, `03_DeployVault`, `BaseScript`, `standalone`	6	186
Total in scope	**16**	**1,112**

All ten findings root-cause in the core contracts and their interfaces; no issues were identified in the deployment scripts.

Out of scope (per the proposal): `makebanc-vaults/upgrade/` and `makebanc-vaults/test/`. The off-chain services (`makebanc-safe-service`, `mb-platform-be`), the in-development NAV oracle, and OpenZeppelin libraries are covered separately and are not part of this smart-contract review.

System & Deployment Architecture

The in-scope deployment scripts were reviewed; they provision the **production** ownership topology described below. No defects were found in the deployment scripts.

- **`00_DeployTimelockController`** deploys an OpenZeppelin `TimelockController` with `minDelay` set from `TIMELOCK_MIN_DELAY` (target **96h**), `PROPOSER_ROLE`/`CANCELLER_ROLE` granted to the **2-of-3 Safe multisig** (`SAFE_MULTISIG_ADMIN`) plus dedicated cancellers, `EXECUTOR_ROLE` restricted to the multisig, and `address(0)` as the timelock admin (self-administered — no external admin role).
- **`01_DeployTierManager`** / **`02_DeployVaultFactory`** / **`03_DeployVault`** (with `BaseScript`) deploy and wire the upgradeable contracts behind proxies.

In the production topology the roles are **separated**:

- **Governance — 2-of-3 Safe multisig** proposes/executes through the **96h Timelock**, which **owns the Vault** (UUPS upgrades, fee-parameter and config setters) and **owns the Fee Manager**. Time-delayed and m-of-n.
- **Operations — 1-of-1 Safe** (backend-fronted) holds the **value-updater** and **capital-activation- signaler** roles (NAV writes + deposit/redeem settlement, no cooldown) and **owns the Tier Manager**.
- **Pauser** is a separate role (OpenZeppelin `AccessControl`); **`unpause()`** is held by the Timelock.

Because owner-gated actions on the Vault (upgrades, fee-parameter changes, configuration) route through the 96h Timelock under multisig control, their immediacy is bounded by the delay and m-of-n approval rather than a single key.

Live Deployment Configuration

The values below were read **live via** `cast` against Base mainnet (chain 8453); several findings' reachability and impact depend on these values.

Network	Base mainnet (chain 8453) + Base Sepolia testnet
USDC (asset)	`0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913`
FeeManager	`0x202B34c98b79ae59838aFCFc36606846272F00b9`
TierManager	`0x6F25fC2a0436fA684Bb5DbBBef363423DDE5850C`
VaultsFactory	`0x8cB89e16f73e9ab4490c20dF25470222bEDd61E0`
Vault — mbArb	`0x773911c29cD7Ba9e80e9B46419e60F1670a1f6Ed`
Vault — mbDir	`0x3c6187d90d813536D05971B2c5D229e7a4BdAFb1`
Vault — mbMulti	`0x68f7D4F1fb3d34aF8229928d6fc722A421493b0e`
Owner (all contracts)	`0xde5A…6ed5` — a 1-of-1 Gnosis Safe (threshold 1)
Fees	`performanceFeeBps = 3000` (30%), `managementFeeBps = 300` (3%/yr) on all vaults
Tiers	`defaultFeeCutBps = 2000` (unused by fee math); `tiers[1]`/`tiers[2]` free (`tierFeeCutBps = 0`, `priceUsdc = 0`); all users on free Tier 1
Minimums	`minDepositAmountAssets = 0`, `minRedeemAmountShares = 0`
Operational roles	`valueUpdater = capitalActivationSignaler = feeCollector = address(0)` (owner-fallback runs hot-path ops)
Live share prices	≈ mbArb 0.18, mbDir 1606, mbMulti empty

Trust Model

Actor	Trust	Notes
Governance — 2-of-3 Safe multisig + 96h Timelock	TRUSTED	The multisig proposes/executes through the Timelock, which owns the Vault (UUPS upgrades, fee-parameter and config setters) and the Fee Manager; holds `unpause()`. Time-delayed (96h) and m-of-n.
Operations — 1-of-1 Safe (backend-fronted)	TRUSTED	Holds value-updater + capital-activation-signaler (NAV writes, deposit/redeem settlement, no cooldown) and owns the Tier Manager. Hot/operational.
Pauser (OZ AccessControl)	TRUSTED	Can pause the Vault; `unpause()` is held by the Timelock (governance).
Users	UNTRUSTED	`requestDeposit` / `requestRedeem` only; KYC + active-tier gated.
Operators	UNTRUSTED	`setOperator`-approved; submit requests on a controller's behalf; cannot settle.
Ceffu / Binance / NAV feed	OUT OF SCOPE	Off-chain custody, trading, and valuation source.

The model above is the **production** topology provisioned by the in-scope deployment scripts and is the trust model this audit assesses.

Structure & Organization of The Security Report

Issues are tagged “Open”, “Acknowledged”, “Partially Resolved”, “Resolved”, or “Closed”. The statuses below reflect FailSafe’s review of the developer remediations in **PR #20**: **nine Resolved and one Acknowledged.**

- **Open**: reported and awaiting remediation from the developer team.
- **Acknowledged**: reviewed and accepted, but the team has decided not to fix it.
- **Partially Resolved**: mitigations applied, yet some risk remains.
- **Resolved**: fully addressed; no further work necessary.
- **Closed**: no longer pertinent or actionable.

Severity	Definition
Critical	The issue affects the platform in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the platform to compile or operate in a significant way.
Medium	The issue affects the ability of the platform to operate in a way that doesn’t significantly hinder its behavior.
Low	The issue has minimal impact on the platform’s ability to operate.
Info	Informational; no direct risk to the platform’s operation.

Summary of Findings

Severity	Total	Open	Ack.	Partial	Resolved	Closed
● Critical	0	0	–	–	–	–
● High	2	0	–	–	2	–
● Medium	3	0	1	–	2	–
● Low	5	0	–	–	5	–
Total	**10**	**0**	**1**	–	**9**	–

#	Severity	Finding	Status
1	High	Principal overcharge when a position is withdrawn in multiple requests	Resolved
2	High	Performance-fee evasion by escrowing shares before the epoch sweep	Resolved
3	Medium	Vault factory sets fee caps in the wrong order versus its interface	Resolved
4	Medium	Tier fee-cut discount is recorded but never applied	Acknowledged
5	Medium	Cheap top-up shares inherit a stale high-water mark and escape fees	Resolved
6	Low	User-valuation view can revert on an unguarded subtraction	Resolved
7	Low	A USDC-blacklisted recipient bricks the whole batch redemption	Resolved
8	Low	Performance-fee shares can round to zero on high-price vaults	Resolved
9	Low	Management fee applies a rate change retroactively / clock reset on zero-share mint	Resolved
10	Low	Tier renewal overwrites the remaining duration	Resolved

1. Principal Overcharge When a Position Is Withdrawn in Multiple Requests

Severity: HIGH **Status:** RESOLVED (PR #20)

Description

Withdrawing a position in more than one redemption request causes the protocol to overcharge the user: every withdrawal after the first is taxed on its full principal as if it were profit. A user with no gains at all loses the full performance-fee rate on each later tranche. This is a self-inflicted footgun, not an attack: only the user's own funds are affected, the overage accrues to the protocol's fee collector, and one user cannot do this to another (`requestRedeem` only ever moves the caller's or their operator's own shares).

`requestRedeem` escrows shares into the vault immediately, so once a full position is queued the user's balance is zero. When `_settleRedeem` finishes a tranche it runs `delete users[controller]` on a zero balance, which erases the user's high-water mark; the next tranche then reads a high-water mark of zero, treats the entire payout as profit, and applies the performance fee to principal. The earlier remediation that re-initialized the high-water mark on the deposit path did not cover this redeem-side deletion.

On-chain: `minRedeemAmountShares = 0` and `performanceFeeBps = 3000` on all three vault proxies (mbArb `0x7739…f6Ed`, mbDir `0x3c61…AFb1`, mbMulti `0x68f7…3b0e`), so a position can be split into multiple requests freely and the overcharge is 30% of each tranche after the first.

Source

The defect spans three hops: shares are escrowed at request time, the per-account mark is consumed at settle, and the account record is deleted in between.

`src/MakebancVault.sol:330` — `requestRedeem` escrows the shares, so the user's balance drops immediately:

```
1  _transfer(_owner, address(this), shares);
```

`src/MakebancVault.sol:362-367` — `_settleRedeem` computes the fee against the per-account mark (the consumer):

```
1  uint256 sellPartValueAtHighestSharePriceFeePaid =
2      Math.mulDiv(shares, users[controller].highestSharePriceFeePaidX18, 1e18);
3
4  uint256 netProfit = assets > sellPartValueAtHighestSharePriceFeePaid
5      ? assets - sellPartValueAtHighestSharePriceFeePaid
6      : 0;
```

`src/MakebancVault.sol:386-390` — the root cause: once the balance is zero, the whole user record (including the mark) is deleted, so any *later* tranche reads `highestSharePriceFeePaidX18 == 0`:

```
if (balanceOf(controller) == 0) {
    delete users[controller];
    emit UserLeftVault(controller, address(this));
}
```

Impact

- A user who exits in multiple requests is overcharged the performance-fee rate on principal — real value loss, not a fee on actual profit.
- Whether it occurs in normal operation depends on whether a full exit is ever expressed as two or more simultaneously-pending requests per vault; either way, a user can also trigger it on themselves.

Remediation

Do not destroy a user's fee state while other redeem requests for them are still pending.

```
// Option A — guard the delete with a count of unfilled redeem requests.
// add state: mapping(address => uint256) public openRedeemRequests;  (++ in requestRedeem, -- in _settleRedeem)
if (balanceOf(controller) == 0 && openRedeemRequests[controller] == 0) {
    delete users[controller];
    emit UserLeftVault(controller, address(this));
}

// Option B (preferred — also fixes finding 2): snapshot the mark into the request and settle against it.
// in requestRedeem:  redeemRequests[controller][requestId].hwmX18 = users[controller].highestSharePriceFeePaidX18;
// in _settleRedeem:  uint256 hwmX18 = redeemRequests[controller][requestId].hwmX18;
//                    uint256 sellPart = Math.mulDiv(shares, hwmX18, 1e18);
```

Proof of Concept

`pocs/test/PoC01_PrincipalOvercharge.t.sol` — with **zero profit**, `testBug_splitRedeemConfiscatesPrincipal` shows the same position split into two requests returns **1,700 USDC (300 of pure principal taken)**, and asserts the mark is wiped to 0 after the first settle; `testFix_remediationReturnsFullPrincipal` shows a redeem that preserves the mark returns the full 2,000 USDC and that the second tranche owes 0 fee against the preserved mark.

Resolution

`requestRedeem` now snapshots the high-water mark into `redeemRequests[c][id].hwmX18` and `_settleRedeem` taxes against that snapshot, so a sibling request's `delete users[c]` can no longer zero a later tranche's mark. Verified against the patched source: a split redeem at zero profit returns the full principal (no confiscation).

2. Performance-Fee Evasion by Escrowing Shares Before the Epoch Sweep

Severity: HIGH **Status:** RESOLVED (PR #20)

Description

A user can sidestep almost the entire performance fee on their gains by moving their shares into a pending withdrawal just before the protocol's periodic fee sweep. The sweep skips shares already queued for withdrawal yet still advances the user's fee mark, so the queued shares later cash out fee-free. This is a fee-accounting defect — the loser is the protocol's fee revenue — not an attack on other users.

`requestRedeem` moves shares into escrow, so they leave `balanceOf`. The epoch sweep `_clearUserAccount` sizes the fee from the *current* balance (and skips an account entirely once its balance is zero) but still sets the per-account high-water mark to the epoch close price for the whole account. The escrowed shares then settle against that bumped mark, so their gain goes untaxed.

On-chain: `performanceFeeBps = 3000` and `minRedeemAmountShares = 0` on each vault proxy, so a user can escrow an arbitrary fraction right before the sweep. Realism is moderate because the (trusted) backend controls the sweep/settle cadence and could neutralize it by ordering.

Source

The bug spans four hops: escrow removes shares from the balance, the sweep both skips on a zero balance and sizes the fee from the live balance, yet it advances the mark for the whole account.

`src/MakebancVault.sol:330` — escrow removes shares from `balanceOf`:

```
1  _transfer(_owner, address(this), shares);
```

`src/MakebancVault.sol:475` — `_clearUserAccount` skips an account whose balance is zero:

```
1  if (userLastEpochProcessed == lastEpoch || balanceOf(user) == 0) return;
```

`src/MakebancVault.sol:482-483` — and sizes the fee from the *current* balance (escrowed shares excluded):

```
1  uint256 userCostBasisHighestPriceFeePaid = _getUserCostBasis(user, userHighestSharePriceFeePaidX18);
2  uint256 userCostBasisLastEpochEndPrice = _getUserCostBasis(user, lastEpochEndPriceX18);
```

`src/MakebancVault.sol:487` — yet it advances the mark for the whole account, so the escrowed shares inherit it and `_settleRedeem` (`:362-363`) later settles them fee-free:

```
1  userData.highestSharePriceFeePaidX18 = lastEpochEndPriceX18;
```

Impact

- Loss of protocol fee revenue, roughly the full performance-fee rate on the escrowed gain, repeatable each epoch.
- The same root also lets an un-crystallized epoch peak go untaxed if the price reverses before settlement.

Remediation

Account for escrowed (pending-redeem) shares in the fee crystallization — the cleanest path is the per-request snapshot (Option B of finding 1). Alternatively, track escrowed shares and include them in the sweep's cost basis:

```
// add state: mapping(address => uint256) public pendingRedeemShares;  (+= in requestRedeem, -= in _settleRedeem)

function _getUserCostBasis(address user, uint256 priceX18) internal view returns (uint256) {
    return Math.mulDiv(balanceOf(user) + pendingRedeemShares[user], priceX18, 1e18);
}

// and clear an account that still has escrowed shares:
if (userLastEpochProcessed == lastEpoch || (balanceOf(user) + pendingRedeemShares[user]) == 0) return;
```

Proof of Concept

`pocs/test/PoC02_EscrowFeeEvasion.t.sol` — `testFix_fullClear_paysFee` charges a non-escrowing control **150,000,000 fee-shares (≈300 USDC)** on a 1,000 USDC gain; `testBug_escrowBeforeSweep_avoidsFee` charges the same user escrowing just before the sweep **150,000 fee-shares (≈0.3 USDC)** — a ~1000× reduction.

Resolution

the same per-request `hwmX18` snapshot (taken at request time, before the sweep bumps the per-account mark) makes the escrowed shares settle against the pre-sweep mark. Verified: escrow-before-sweep now pays the full performance fee (~300 USDC on a 1,000 USDC gain) instead of dust.

3. Vault Factory Sets Fee Caps in the Wrong Order Versus Its Interface

Severity: MEDIUM **Status:** RESOLVED (PR #20)

Description

The vault factory accepts its two fee-ceiling settings in the opposite order from its own published interface, so anyone deploying a vault through that interface sets the performance-fee cap and management-fee cap backwards — permanently, since the caps cannot be changed after deployment. The fix is simply for the interface and the implementation to agree; this is a correctness defect, not an attack.

Because Solidity binds arguments by position, a caller coding to the interface's parameter names sets the two immutable caps swapped.

Source

`src/VaultsFactory.sol:59–60` — the concrete function declares the caps `(performance, management)`:

```
uint256 _performanceFeeCapBps,
uint256 _managementFeeCapBps
```

`src/VaultsFactory.sol:68` — and forwards them positionally to the vault constructor:

```
MakebancVault vaultImplementation = new MakebancVault(_performanceFeeCapBps, _managementFeeCapBps);
```

`src/MakebancVault.sol:135–137` — the constructor is consistent with the concrete factory:

```
constructor(uint256 _performanceFeeCapBps, uint256 _managementFeeCapBps) {
    MANAGEMENT_FEE_CAP_BPS = _managementFeeCapBps;
    PERFORMANCE_FEE_CAP_BPS = _performanceFeeCapBps;
```

`src/interfaces/IVaultsFactory.sol:11–12` — but the published interface declares the same two params **reversed**:

```
uint256 managementFeeCapBps,
uint256 performanceFeeCapBps
```

Impact

- A vault deployed through the interface gets its performance-fee and management-fee ceilings swapped, and the caps are immutable — recoverable only by redeploying.

- The in-repo deploy script uses the concrete `VaultsFactory` type, so the live vaults are likely unaffected; the risk is to any external integrator coding against the published interface.

Remediation

Align the interface to the implementation — reorder the trailing parameters so position matches meaning:

```
// src/interfaces/IVaultsFactory.sol
function deployVault(
    address asset, address csw, uint256 minDepositAmountAssets,
    string memory name, string memory symbol,
    uint256 performanceFeeCapBps, uint256 managementFeeCapBps   // was (management, performance)
) external returns (address);
```

Add a deploy-time test asserting the resulting `PERFORMANCE_FEE_CAP_BPS` / `MANAGEMENT_FEE_CAP_BPS`.

Proof of Concept

`pocs/test/PoC03_FeeCapParamOrder.t.sol` — both callers intend perf-cap 4000 / mgmt-cap 1000; `testBug_interfaceOrderSwapsCaps` shows the interface-typed caller gets `(1000, 4000)` — swapped — while `testFix_alignedOrderMatchesIntent` shows the implementation order lands the same intent on the correct caps.

Resolution

`IVaultsFactory.deployVault` now declares its trailing parameters as `(performanceFeeCapBps, managementFeeCapBps)`, matching `VaultsFactory.deployVault` and the `MakebancVault` constructor. Verified: an interface-following caller now lands the caps as intended.

4. Tier Fee-Cut Discount Is Recorded but Never Applied

Severity: MEDIUM **Status:** ACKNOWLEDGED (PR #20)

Description

Each withdrawal records the user's tier fee discount, but the fee calculation never reads it — every user is charged the flat performance fee regardless of tier. Today this is harmless dead code (all configured tiers are free), but it silently mis-charges every tiered user the moment a paid tier with a real discount is configured.

`requestRedeem` writes `redeemRequests[...].tierFeeCutBps`, yet none of the four fee computations consume it; they all use the flat `performanceFeeBps`. The `TierManager` fee-cut API is therefore disconnected from actual fee accrual.

On-chain: on the TierManager `0x6F25…850C`, `tiers[1]` and `tiers[2]` both have `tierFeeCutBps = 0` and all users are on free Tier 1, while each vault charges `performanceFeeBps = 3000` — so if the tier rate were meant to apply, every current user would be over-charged; as configured today, nothing is mis-charged yet.

Source

`src/MakebancVault.sol:333-334` — the per-request discount is written:

```
redeemRequests[controller][requestId].tierFeeCutBps =
    ITierManager(tierManager).getApplicableFeeCutBps(controller);
```

`src/MakebancVault.sol:370` — but every fee computation (also `:427`, `:441`, `:542`) uses the flat global rate and never reads the snapshot:

```
perfFeeAssets = netProfit * performanceFeeBps / 10_000;
```

`src/TierManager.sol:146-150` — the discount that is computed and then dropped:

```
function getApplicableFeeCutBps(address user) public view returns (uint256) {
    if (userTier[user].endTime <= block.timestamp) return defaultFeeCutBps;
    return tiers[userTier[user].tierId].tierFeeCutBps;
}
```

Impact

- Dead state and a misleading on-chain field (`getRedeemRequest` returns a discount that integrators may trust but that never takes effect).
- A latent fee-correctness defect: the day a paid, non-zero-cut tier is configured, tiered users are charged the flat rate instead of their tier rate.

Remediation

Resolve the intent, then either remove the dead path or wire the snapshot into the fee math.

```
// Option A — delete the dead path: remove the write at :333–334, drop tierFeeCutBps from the Request struct,
//            and remove the unused TierManager fee-cut API (getApplicableFeeCutBps / applyFeeCut / defaultFeeCutBps).

// Option B — apply the per-request snapshot in _settleRedeem:
uint256 rateBps = redeemRequests[controller][requestId].tierFeeCutBps;
perfFeeAssets = netProfit * rateBps / 10_000;
// (the epoch path _clearUserAccount has no snapshot; it would call getApplicableFeeCutBps(user) live)
```

Proof of Concept

`pocs/test/PoC04_TierFeeCutIgnored.t.sol` — a user is placed on a tier whose `tierFeeCutBps` is a non-zero **10%** (1,000 bps), distinct from the flat 30%. `testBug_tierFeeCutSnapshotNeverApplied` asserts the redeem request *does* snapshot the 10% cut (`redeemRequests[...].tierFeeCutBps == 1000`), yet the user is still charged the flat 30% on the 1,000 USDC gain (net **1,700 USDC**) — the snapshot is never read. `testFix_applyFeeCutChangesTheCharge` routes the flat 300 USDC fee through the protocol's own `TierManager.applyFeeCut`, which returns **30 USDC** — a different, lower figure — showing the snapshot would change the charge if it were ever consumed; as configured, it never is.

This also confirms by source inspection that `tierFeeCutBps` has exactly one write (`:333–334`) and zero reads in the vault's fee math (the four sites at `:370`, `:427`, `:441`, `:542` all use `performanceFeeBps`).

Resolution

MakeBanc team: *"TierFeeCut is not a discount; it is the legacy per-user fee model, superseded by the per-vault fee model. The code is intentionally retained as we may use it in the future for other vaults."*

FailSafe note: accepted as a design decision. The `redeemRequests[...].tierFeeCutBps` field and the TierManager fee-cut API remain inert (never read by the fee math); harmless while no paid, non-zero-cut tier is configured.

5. Cheap Top-Up Shares Inherit a Stale High-Water Mark and Escape Fees

Severity: MEDIUM **Status:** RESOLVED (PR #20)

Description

A user who has already locked in a high fee mark can buy more shares cheaply after a price dip and have those cheap shares inherit the old high mark, so when the price recovers their real gains escape the performance fee. To be precise about what is *not* a defect: a plain top-up while merely in profit is fair — the blended mark is exact — so the issue is specifically the inheritance of a *stale-high* mark by later, cheaper deposits.

The vault keeps one high-water mark per account. On a top-up, `_settleDeposit` sets it to `Math.max(currentMark, newAverageCost)`. When the price has dropped below the user's existing mark, the `Math.max` keeps the old high mark and applies it to the newly-bought cheap shares too; on recovery, their gain back up to that mark is untaxed.

On-chain: `minDepositAmountAssets = 0` and `performanceFeeBps = 3000` on each vault proxy (mbArb `0x7739…f6Ed`, mbDir `0x3c61…AFb1`, mbMulti `0x68f7…3b0e`); the dip-then-recovery price regime is plausible for a CEX-strategy NAV.

Source

`src/MakebancVault.sol:275–277` — a top-up first blends the average acquisition cost:

```
users[controller].averageShareAcquisitionCostX18 = (
    users[controller].averageShareAcquisitionCostX18 * previousShares + sharePriceX18 * shares
) / (previousShares + shares);
```

`src/MakebancVault.sol:279–286` — it then ratchets the mark *up* with `Math.max`, so a stale-high mark survives a cheap deposit instead of blending down toward the new shares' price:

```
if (users[controller].highestSharePriceFeePaidX18 == 0) {
    users[controller].highestSharePriceFeePaidX18 = sharePriceX18;
} else {
    users[controller].highestSharePriceFeePaidX18 = Math.max(
        users[controller].highestSharePriceFeePaidX18, users[controller].averageShareAcquisitionCostX18
    );
}
```

`src/MakebancVault.sol:362–363` — at redemption the cheap shares are taxed only above that inherited mark (the consumer):

```
uint256 sellPartValueAtHighestSharePriceFeePaid =
    Math.mulDiv(shares, users[controller].highestSharePriceFeePaidX18, 1e18);
```

Impact

- Loss of protocol fee revenue equal to the full performance-fee rate on the cheap tranche's recovered gain.

Remediation

Do not let a stale-high mark apply to newly-deposited shares — blend it down to the share-weighted average on a top-up (drop the `Math.max`):

```
// in _settleDeposit's else branch, replace the Math.max with a share-weighted blend:
users[controller].highestSharePriceFeePaidX18 = (
    users[controller].highestSharePriceFeePaidX18 * previousShares + sharePriceX18 * shares
) / (previousShares + shares);
```

A structural alternative that also resolves findings 1 and 2 is to track the mark and cost basis per deposit tranche rather than as one per-account scalar.

Proof of Concept

`pocs/test/PoC05_StaleHwmInheritance.t.sol` — `testBug_staleHighMarkInheritance_evadesFee`: after building a mark of 2.0, a user buys a cheap tranche at 1.0; on recovery to 2.0 they pay **0 fee-shares**, while a control buying the identical tranche fresh pays **~300 USDC**. `testFix_blendedMark_surfacesTheGain` shows a volume-weighted blended mark (1.5) surfaces the recovered gain to tax that the stale-high mark hides.

Resolution

`_settleDeposit` now blends the high-water mark by share volume on a top-up instead of `Math.max`, so a cheap top-up pulls the mark down and the freshly-bought tranche's recovered gain is taxed. Verified against the patched source: the cheap tranche's gain is now charged, and — checked explicitly — the blend does **not** over-charge shares whose gains were already paid (it nets within one mark, so the charge is fair-or-under, never above). The durable fix remains per-tranche fee accounting if many tranches per account become common.

6. User-Valuation View Can Revert on an Unguarded Subtraction

Severity: LOW **Status:** RESOLVED (PR #20)

Description

A read-only valuation helper can revert instead of returning a number: when a user's accrued fee debt exceeds their current position value, an unguarded subtraction underflows. A view function should never revert; this can break any dashboard or integration that reads it.

No on-chain function consumes this view — only external integrators — so the impact is integration robustness, not protocol behavior.

Source

`src/MakebancVault.sol:567-569` — the unguarded subtraction:

```
function getUserValuation(address user) public view returns (uint256) {
    uint256 userAssets = convertToAssets(balanceOf(user));
    return userAssets - _getUserRealizedDebt(user);   // reverts (Panic 0x11) if debt > userAssets
}
```

`src/MakebancVault.sol:574-596` — `_getUserRealizedDebt` is where the debt that can exceed `userAssets` comes from (it is priced off the epoch close, while `userAssets` is the current mark-to-market):

```
if (lastEpochEndPriceX18 > userHighestSharePriceFeePaidX18) {
    uint256 userCostBasisHighestPriceFeePaid = _getUserCostBasis(user, userHighestSharePriceFeePaidX18);
    uint256 userCostBasisLastEpochEndPrice = _getUserCostBasis(user, lastEpochEndPriceX18);
    performanceFee = _getPerformanceFee(user, userCostBasisLastEpochEndPrice, userCostBasisHighestPriceFeePaid);
}
```

Impact

- A standard read-only call can revert for some users, breaking off-chain reporting or any integrator that aggregates valuations. No funds or on-chain state are affected; `getUserDebt` (no subtraction) stays callable.

Remediation

Clamp the subtraction so the view always returns a value:

```
function getUserValuation(address user) public view returns (uint256) {
    uint256 userAssets = convertToAssets(balanceOf(user));
    uint256 debt = _getUserRealizedDebt(user);
    return userAssets > debt ? userAssets - debt : 0;
}
```

Resolution

`getUserValuation` now clamps the subtraction (`userAssets > debt ? userAssets - debt : 0`). Verified: no remaining unguarded subtraction on the valuation/debt path.

7. A USDC-Blacklisted Recipient Bricks the Whole Batch Redemption

Severity: LOW **Status:** RESOLVED (PR #20)

Description

If a single recipient in a batch redemption has been blacklisted by USDC, the entire batch reverts and nobody in it is paid until an operator rebuilds the batch without that address. This is a robustness gap: the batch should isolate per-recipient failures rather than fail as a whole.

On-chain: the asset on all three vaults is canonical Base USDC `0x8335…2913`, which is blacklist-capable. The single `settleRedeem` entry point lets an operator settle the unaffected users individually, so the impact is recoverable.

Source

`src/MakebancVault.sol:377` — `_settleRedeem` pushes USDC directly to the controller:

```
IERC20(asset()).safeTransfer(controller, assets - perfFeeAssets);
```

`src/MakebancVault.sol:520-522` — and `batchSettleRedeem` calls it in an atomic loop with no per-item isolation, so one reverting transfer reverts the whole batch:

```
for (uint256 i = 0; i < requestIds.length; i++) {
    _settleRedeem(requestIds[i], controllers[i]);
}
```

Impact

- One blacklisted recipient blocks settlement for every co-batched user until the batch is rebuilt; no funds are lost and recovery is available via single `settleRedeem`.
- **Detection burden:** on-chain there is no isolation and no breadcrumbs — diagnosing *which* recipient caused the revert depends entirely on off-chain screening (`USDC.isBlacklisted`) or transaction tracing.

Remediation

Isolate per-recipient failures so one blacklisted payee cannot brick the batch, and make the failure explicit:

```
event SettlementSkipped(address indexed controller, uint256 requestId);

// in batchSettleRedeem's loop, replace the direct call with:
try this.settleRedeem(requestIds[i], controllers[i]) {}
catch { emit SettlementSkipped(controllers[i], requestIds[i]); }
```

Alternatively adopt pull-payments: credit `withdrawable[controller] += (assets - perfFeeAssets)` at settle and add a `withdraw()` the controller calls separately.

Resolution

`batchSettleRedeem` now settles each request through a self-only `settleRedeemSelf` wrapper inside a `try/catch`, so a failing (e.g. blacklisted) recipient reverts only its own settlement — the rest of the batch proceeds and the failure is surfaced via `RedeemSettleFailed`. Verified: auth/pause are enforced on the public entry points and the failed request's state rolls back for retry.

8. Performance-Fee Shares Can Round to Zero on High-Price Vaults

Severity: LOW **Status:** RESOLVED (PR #20)

Description

On a high-priced vault, a small performance fee can round down to zero shares while the user is still charged the fee in assets, leaving a sub-cent amount unbacked. It's a rounding/accounting defect, not exploitable for profit (the redeemer loses the dust; it accrues to remaining holders).

This only triggers where the share price exceeds 1 (e.g. mbDir at ≈1606), not on low-price vaults (mbArb at ≈0.18).

Source

`src/MakebancVault.sol:372` — the fee-share conversion floors to zero for a small fee:

```
perfFeeShares = convertToShares(perfFeeAssets);   // floors to 0 for small perfFeeAssets on a high-price vault
```

`src/MakebancVault.sol:377-384` — the asymmetry: the user is always charged the fee in assets and all shares are burned, but the transfer to the fee manager is gated on `perfFeeShares > 0`:

```
IERC20(asset()).safeTransfer(controller, assets - perfFeeAssets);   // user pays the fee...
_burn(address(this), shares - perfFeeShares);
if (perfFeeShares > 0) {                                            // ...but nothing is collected if 0
    _approve(address(this), feeManager, perfFeeShares);
    IFeeManager(feeManager).collectPerformanceFeeInShares(address(this), perfFeeShares);
}
```

`src/MakebancVault.sol:394` — and `totalAssetsValue` keeps `perfFeeAssets` with no backing share:

```
totalAssetsValue -= (assets - perfFeeAssets);
```

Impact

- Sub-cent value left unbacked per dust redemption on high-price vaults; cumulative but rounding-scale, and not attacker-profitable.

Remediation

Waive the fee when it rounds to zero shares, so payout and accounting stay consistent:

```
uint256 perfFeeShares = convertToShares(perfFeeAssets);
if (perfFeeAssets > 0 && perfFeeShares == 0) {
    perfFeeAssets = 0;   // (or round perfFeeShares up)
}
```

Resolution

the performance fee is now waived when it rounds to zero shares (when `perfFeeShares == 0`, `perfFeeAssets` is set to 0), applied in `_settleRedeem` and mirrored in the redeemable-assets view. Verified consistent across both paths.

9. Management Fee Applies a Rate Change Retroactively / Clock Reset on Zero-Share Mint

Severity: LOW **Status:** RESOLVED (PR #20)

Description

The management fee applies the current rate across the whole elapsed period, and a fee mint that computes to zero shares still resets the accrual clock — forfeiting the unminted fee. Both paths are operated by the owner/signaler only, so this is a defensive-hardening item rather than a user-facing risk.

Source

`src/MakebancVault.sol:770–777` — the fee is computed with the *current* rate over the entire window since the last mint:

```
function getManagementFeeShares() public view returns (uint256) {
    uint256 timeSinceLastMint = block.timestamp - lastMintSharesForVaultManagementFee;
    uint256 managementFeeAssets = (totalAssets() * managementFeeBps * timeSinceLastMint) / 365 days / 10_000;
    return convertToShares(managementFeeAssets);
}
```

`src/MakebancVault.sol:758–765` — and the clock is advanced unconditionally, even when zero shares are minted:

```
function mintSharesForVaultManagementFee() external onlyCapitalActivationSignaler whenNotPaused {
    uint256 managementFeeShares = getManagementFeeShares();
    _mint(address(this), managementFeeShares);
    _approve(address(this), feeManager, managementFeeShares);
    IFeeManager(feeManager).collectManagementFee(address(this), managementFeeShares);
    lastMintSharesForVaultManagementFee = block.timestamp;   // advanced even when managementFeeShares == 0
}
```

Impact

- A rate increase applies retroactively to the un-minted period (bounded by the immutable cap); a zero-share mint forfeits the protocol's own accrued fee. No user funds are at risk and there is no untrusted path.

Remediation

Skip the mint (and clock reset) when no shares are due, and settle accrual at the old rate before a rate change:

```
function mintSharesForVaultManagementFee() external onlyCapitalActivationSignaler whenNotPaused {
    uint256 managementFeeShares = getManagementFeeShares();
    if (managementFeeShares == 0) return;          // don't reset the clock on a 0-share mint
    _mint(address(this), managementFeeShares);
    _approve(address(this), feeManager, managementFeeShares);
    IFeeManager(feeManager).collectManagementFee(address(this), managementFeeShares);
    lastMintSharesForVaultManagementFee = block.timestamp;
}
// and in setManagementFeeBps: mint/accrue at the OLD rate before updating managementFeeBps.
```

Resolution

`_mintManagementFee()` returns early without resetting the accrual clock when zero shares are due, and `setManagementFeeBps` crystallizes the fee at the old rate before applying the new one. Verified: a rate change no longer re-prices the elapsed period and a zero-share mint no longer forfeits accrual.

10. Tier Renewal Overwrites the Remaining Duration

Severity: LOW **Status:** RESOLVED (PR #20)

Description

Renewing a tier overwrites its remaining time instead of extending it, so a user who renews early loses the days they already had. It's a straightforward bug; impact is zero today because tiers are free, but it would cost users paid time once paid tiers ship.

Source

`src/TierManager.sol:104-110` — `_setUserTier` sets the end time from `block.timestamp`, replacing any unexpired duration rather than adding to it:

```
function _setUserTier(address user, uint256 tierId) internal {
    uint256 endtime = block.timestamp + tiers[tierId].duration;   // overwrites, does not extend
    userTier[user] = UserTier(tierId, endtime);
    userHasTier[user] = true;
    emit UserTierSet(user, tierId);
}
```

Impact

- An early renewal forfeits the user's unexpired tier time. Zero monetary impact while tiers are free (`priceUsdc = 0`); it would matter once paid tiers with non-zero prices are configured.

Remediation

Extend from the later of the existing end time and now, so renewing never discards remaining time:

```
function _setUserTier(address user, uint256 tierId) internal {
    uint256 base = userTier[user].endTime > block.timestamp ? userTier[user].endTime : block.timestamp;
    uint256 endtime = base + tiers[tierId].duration;
    userTier[user] = UserTier(tierId, endtime);
    userHasTier[user] = true;
    emit UserTierSet(user, tierId);
}
```

Resolution

`_setUserTier` now stacks the new duration onto the remaining time when renewing the same active tier.