vltUSDC Vault — Security Audit

Internal review · Claude (Fable 5) with Solidity Audit Skill · July 2026

Project	vltUSDC — auto-compounding Uniswap V4 VLT/USDC full-range LP vault
Contracts in scope	`contracts/VltUsdcVault.sol`, `contracts/ZapHelper.sol`
Compiler	solc 0.8.26, optimizer 200 runs, `viaIR`, `evmVersion: cancun`
Performed by	Claude (Fable 5) with Solidity Audit Skill
Date	July 8, 2026
Status of code	Pre-deployment (not yet on mainnet); Hardhat workspace, 47 tests passing (61 after the §7d keeperless redesign)

This audit covers source-code analysis, static analysis (Slither, re-run fresh for this report), and review of the project's dynamic test suite (executed for this report). It is not a substitute for the planned independent Shieldify review, formal verification, or an economic simulation of deposit-flow compounding cadence. See **Disclaimer** at the end.

---

1. Prior audit context (VLT token)

Two prior reports cover the **VLT token itself** (`VaultToken`, `0x6b785a0322126826d8226d77e173d75dafb84d11`) — the 18-decimal ERC-20 this vault pairs with USDC. Neither covers the contracts in scope here; they matter because the vault's trust model leans on VLT being a well-behaved token.

- `src/media/VLT_Audit.pdf` (Claude Opus 4.7 w/ Solidity Audit Skill, April 22, 2026): 0 Critical / 0 High / 0 Medium / 3 Low / 8 Informational. All findings cosmetic or pre-deployment-only. Confirms: fixed 1.8M supply, no mint, no pause, no blocklist, no upgrade path, no fee-on-transfer, no rebasing, LP permanently locked at the token contract.
- **Shieldify Security — Bankroll Vault Token (VLT) Security Review** (June 1, 2026, report): 1 Informational (supply-conservation invariant unobservable pre-bootstrap; acknowledged). On-chain verification that 99.998% of Uniswap V2 LP is permanently unwithdrawable — **"rug-proof on liquidity"** determination.

Relevance to this audit: VLT is verified to be a plain, fixed-supply ERC-20 with no transfer hooks, no fee-on-transfer, and no rebasing. The vault's assumptions about its two pool tokens (see §4 trust model and I-06) are therefore sound for the VLT leg. USDC remains an upgradeable, blacklistable third-party contract.

---

2. Executive summary

`VltUsdcVault` is a single-position, full-range Uniswap V4 VLT/USDC LP vault whose ERC-20 shares are denominated in pool **liquidity (L)** rather than dollars. It is **fully ownerless**: no admin, no pause, no upgrade path,

no privileged parameter, no sweep. The two entrypoints (`deposit`, `redeem`) are permissionless and `nonReentrant` — compounding has no entrypoint of its own; it runs inside deposits (§7d); the only external dependency in the critical path is the canonical V4 `PoolManager`, with hooks explicitly disallowed at construction (`VltUsdcVault.sol:283`). `ZapHelper` is a stateless, ownerless periphery contract that converts USDC-only deposits into the balanced pair via one immutable whitelisted router; a fault in it cannot affect vault solvency.

The architecture is deliberately minimal and the hard problems of this design — the first-depositor inflation attack, direct-donation share inflation, and V4's fold-all-fees-into-any-`modifyLiquidity` behavior (the project's "BUG-1") — are all correctly handled (§7). Slither reports **0 results** (75 contracts, 96 detectors, re-run for this audit) and the test suite (including a stateful invariant harness, reentrancy probes, and fee-retention tests against a real `PoolManager`) passes — 42 tests at audit time, 47 after the post-audit remediations and event additions below.

No Critical, High, or Medium severity issues were identified. The findings below are Low-severity refinements and informational notes, several of which are conscious, in-code-documented design choices flagged here for completeness.

Findings summary

Severity	Count	Status
Critical	0	—
High	0	—
Medium	0	—
Low	3	**All 3 fixed** (July 8, 2026 — see per-finding Resolution notes)
Informational	10	Acknowledged / by design (I-10 added by the July 9 re-audit)

July 14, 2026 (§7d): the keeperless auto-compound redesign introduced **and fixed, pre-deployment,** one Medium (M-01 — pre-seed donation DoS via the new deposit trigger) and obsoleted I-01/I-07. All `VltUsdcVault.sol` line citations were re-anchored to the post-redesign source (`main` @ `2006f11`, July 14, 2026); §7d is the authoritative record of the design delta.

Remediation re-verification (July 8, 2026): after the L-01–L-03 fixes, the contracts recompile clean, **Slither again reports 0 results** (75 contracts, 96 detectors), Solhint is clean, and the suite grew to **46 passing tests** — including four new regression tests: an expired-deadline revert for `deposit`, an expired-deadline revert for `zapDeposit` (asserting the external swap leg never ran), a rebalance-cap test (a one-sided donation with no fresh fees is never swapped), and a `zap()` refund-to-caller test. (The §7a event change below adds a fifth regression test, bringing the final post-remediation total to **47**.)

Adversarial re-audit (July 9, 2026): the final contracts + this report were re-reviewed by an independent multi-agent pass — eight parallel dimensions (V4 delta settlement, fee/event accounting, the L-01 rebalance cap, share/inflation math, ZapHelper, reentrancy/access, arithmetic, and report-accuracy), each candidate finding adversarially verified by a separate skeptic, plus a completeness critic. **No new code correctness or security defect was confirmed** in either contract — the two contract-level candidates raised were both refuted as

intended, already-documented BUG-1 behavior. It surfaced one informational contract nuance (I-10, added above) and three accuracy defects in this document itself — stale line-number citations, an incorrect `slither-disable` count, and a self-contradictory test count — all corrected in this revision.

---

3. Scope

File	nSLOC (approx.)	Coverage
`contracts/VltUsdcVault.sol`	~360	Full, line-by-line
`contracts/ZapHelper.sol`	~120	Full, line-by-line
`contracts/test/*` (MockERC20, ReentrantToken, MockSwapRouter, MockPermit2, V4Harness)	~190	Reviewed only as test scaffolding — **not production code**, must never be deployed

Out of scope: `@uniswap/v4-core`, `@uniswap/v4-periphery`, `@openzeppelin/contracts` (audited upstream dependencies), deploy scripts, the frontend, and the VLT token itself (covered by the prior audits above).

---

4. Architecture, actors, and trust model

Actors

Actor	Capabilities	Notes
Any user	`deposit`, `redeem`, ERC-20 share ops	No privileged role exists anywhere
Triggering depositor	Any deposit at ≥$100 claimable runs the internal compound leg first	No reward — 100% of the harvest reinvests (§7d)
ZapHelper (or any zapper)	Ordinary caller of `deposit`	Vault has no knowledge of it
Uniswap V4 PoolManager	Sole external dependency; calls back `unlockCallback`	Callback gated to `msg.sender == poolManager` (`VltUsdcVault.sol:785-786`)
Deployer	Constructor parameters only	Zero post-deploy privileges

Mutable state on the otherwise-immutable vault is limited to: OZ ERC-20 share accounting, `poolKey` (constructor write-once), `inceptionTime` (write-once on first deposit), the informational 35-slot L/share snapshot ring, and the informational lifetime fee counters `totalFeesVlt` / `totalFeesUsdc` (July 16, 2026 — see §7d; incremented at every fee-collection point so they always equal Σ `Compound` + Σ `FeesRetained`, asserted by test). There is nothing an admin key could touch because there is no admin key.

Core invariants (all verified in code and exercised by `test/vault.invariants.test.js`):

1. **Solvency by construction** — shares are a pro-rata claim on position liquidity; `redeem` removes exactly `floor(L·shares/supply)` and returns both tokens in-kind (`VltUsdcVault.sol:671`). The vault never owes more L than the position holds.

2. **Share supply changes only in `deposit` / `redeem`** — the compound leg mints no shares, so L/share is monotonically non-decreasing across compounds (`VltUsdcVault.sol:942–943`).
3. **Uncompounded pool fees always accrue to all holders** — every `modifyLiquidity` path splits `feesAccrued` out of `callerDelta` and retains it at the vault (`VltUsdcVault.sol:817–850, 884–907`); no path pays fees out — the compound leg converts them to position liquidity (§7d). See I-10 for the one bounded exception this creates for a joining depositor.
4. **Every unlock nets its deltas to zero** — otherwise the PoolManager reverts `CurrencyNotSettled`; all passing flow tests prove this transitively.
5. `MINIMUM_LIQUIDITY` **(1,000 L) is locked at** `0xdead` **forever** on first deposit (`VltUsdcVault.sol:549–550`).

Token assumptions: both pool tokens must be non-hooked, non-fee-on-transfer, non-rebasing ERC-20s. Verified true for VLT (§1). USDC satisfies this today but is an upgradeable proxy with a blacklist (see I-06); the blacklist case is handled and documented in-code (`VltUsdcVault.sol:645–648`).

---

5. Findings — Low

L-01 — The compound leg values and rebalances at manipulable spot price; "never loses" is a bounded-loss claim, not an absolute one

Location: `VltUsdcVault.sol:352–417` (`compoundClaimable`), `1015–1050` (`_rebalance`), `116–126` (price-limit constants)

The compound path gates on a USDC valuation of claimable assets taken from the pool's current `sqrtPriceX96`, and `_rebalance` sizes and executes its internal swap against that same spot price, bounded to a ≤5% *further* move by `sqrtPriceLimitX96`. The 5% band bounds the price impact **of the vault's own swap**, relative to whatever the pool price already is — it does not bound a *prior* manipulation. An attacker can push the pool price, then trigger the compound leg (originally by calling `compound()`; today via a deposit), causing the vault to swap ~half its claimable imbalance at the skewed price and add liquidity at that skewed ratio; the attacker's restoring swap then extracts part of that value.

Why this is Low, not Medium: the exploit is structurally unprofitable in this design. (1) The at-risk notional is only the reinvestable balance (fees since the last compound plus retained dust — typically small; the trigger floor is now $100, §7d). (2) Manipulating a full-range 1%-fee pool requires a large swap whose 1% fee accrues predominantly **to the vault itself** (it is the dominant LP), so the attacker funds the holders they are attacking, twice (push and restore). (3) Liquidity added at a bad price is bounded dust relative to the manipulation cost. The in-code claim "a manipulated price simply defers, never loses" (`VltUsdcVault.sol:116–126`) is slightly stronger than what holds: an unfilled remainder defers, but the filled portion executes at the prevailing (possibly skewed) price. The suite's NAV/share-monotonic-under-price-push test exercises exactly this and passes because the fee income exceeds the skew loss.

Recommendation: No code change required given the economics. If extra belt-and-braces is wanted before mainnet, cap `amountIn` in `_rebalance` to a small multiple of this harvest's `fee0` / `fee1` (bounding worst-case skewed-swap notional even when large retained balances have accumulated), and soften the comment at line 119 to "defers, and bounds any loss to the filled portion at the prevailing price."

Resolution (fixed, July 8, 2026): both recommendations applied. A new constant `REBALANCE_CAP_MULT = 4` caps the rebalance swap input to 4× the value of the current call's freshly harvested pool fees, expressed in the input currency at spot (`_rebalance` now takes `fee0` / `fee1`); with zero fresh fees the cap is zero and the vault never trades held balances at spot. Capped-out imbalance folds forward across compounds. The overstated "never loses" comment was rewritten to state the actual guarantee (the 5% band bounds the vault's own price impact; the cap bounds the notional exposed to a prior manipulation). Regression test: `vault.flows.test.js` — "rebalance swap is capped by fresh fees: a one-sided donation alone is never swapped."

Note (July 14, 2026 — §7d): `compound()` no longer exists as an entrypoint; the same logic is the internal `_compound()` leg triggered by deposits.

Remediation revisited (July 14, 2026 — cap REMOVED): `REBALANCE_CAP_MULT` was deleted under the keeperless design, deliberately. Two reasons. (1) It was ineffective as a defense: it bounded only the swap leg while `_addLiquidity()` always deployed the full loose balance at spot — the independent Codex review's I-02 probe extracted value through the add leg with the cap active, so the cap bounded half of one exposure. (2) It caused real harm: the trigger fires on total claimable value, so one-sided value with no fresh fees re-triggered a paid no-op compound on every deposit (a $101 one-sided donation = standing gas tax) while the value sat idle outside redeemable principal. The operative bound is structural and is the one this finding's economics already relied on: the $100 deposit trigger keeps loose balances at trigger scale under any deposit flow, push/restore manipulation pays 1% pool fees on a far larger notional mostly to the vault itself, and the ≤5% `sqrtPriceLimitX96` bounds the vault's own impact (unfilled remainder folds forward). The donation regression test was inverted accordingly ("a one-sided donation is rebalanced and reinvested by the next trigger").

L-02 — `zap()` sends the unspent-input refund to `recipient`, not to the caller

Location: `ZapHelper.sol:285–303` (`zap`); pre-fix the refund `_sweep` sent unspent `tokenIn` to `recipient` (now `msg.sender`, line 302)

`zap(tokenIn, tokenOut, amountIn, minOut, recipient, swapData)` pulls `amountIn` from `msg.sender`, but if the route under-consumes its input, the leftover `tokenIn` is swept to `recipient` along with the output. When `recipient` is a third party (a common pattern: "swap and send to my cold wallet / a contract"), the caller's refund is silently misdirected to that third party. The NatSpec ("refund any unspent `tokenIn`") does not say to whom, and the natural reading is "to the payer."

Impact: loss limited to route under-spend dust, only when `recipient != msg.sender`; no protocol funds at risk. (`zapDeposit` is unaffected — it sweeps to `msg.sender`.)

Recommendation: `_sweep(tokenIn, msg.sender)` in `zap()`, or document explicitly that all residuals follow `recipient`.

Resolution (fixed, July 8, 2026): `zap()` now sweeps unspent `tokenIn` to `msg.sender` (the payer); `recipient` receives the output only, and the NatSpec states this explicitly. Regression test: `zaphelper.permit2.test.js` — "refunds unspent input to the CALLER, not the recipient."

L-03 — No deadline parameter on `deposit` / `zapDeposit`

Location: `VltUsdcVault.sol:486–502` (`deposit`), `ZapHelper.sol:134–146` (`zapDeposit`)

Neither the vault's `deposit` nor the helper's `zapDeposit` accepts a `deadline`. A transaction that lingers in the mempool (low gas, congestion) can execute much later under market conditions the user no longer intends. The existing bounds (`minShares`, `minVltOut`) do limit the damage — `minShares` is denominated in L, so a stale execution still cannot mint fewer shares than promised — but L-denominated bounds do not pin the *dollar* terms of the entry, and Uniswap periphery convention is to include a deadline. `redeem` and `compound` genuinely do not need one (in-kind / self-bounded respectively).

Recommendation: add a `uint256 deadline` + `require(block.timestamp <= deadline, "expired")` to `deposit` and `zapDeposit` (or accept, documented, that `minShares` / `minVltOut` are the only staleness guards).

Resolution (fixed, July 8, 2026): `deposit(vltAmount, usdcAmount, minShares, deadline)` and `zapDeposit(usdcAmount, swapUsdcToVlt, minVltOut, minShares, deadline, swapData)` both revert `"expired"` past the deadline. The helper checks **before** its external swap leg and passes the same deadline through to the vault (defense in depth). All callers updated: test helpers, seed/simulate scripts (the seed script uses now + 30 min), the frontend ABI, and `vltUSDC.js` (sends now + 30 min). Regression tests: `vault.edge.test.js` — "deposit past its deadline reverts"; `scripts.quoter.test.js` — "zapDeposit past its deadline reverts before the swap leg runs."

---

6. Findings — Informational

I-01 — (superseded — §7d M-01) Pre-first-deposit donation makes the compound path revert

`compoundClaimable()` counts vault-held balances (`VltUsdcVault.sol:369–370`), so a ≥$1 donation made **before any deposit exists** passes the gate, and `_onCompound`'s fee-collect poke (`modifyLiquidity` with `liquidityDelta == 0`, `VltUsdcVault.sol:918–925`) then reverts inside V4 (`CannotUpdateEmptyPosition` — a poke requires an existing position). Nothing is stuck or lost — `compound()` simply reverts until the first deposit creates the position, after which the donation folds into the next compound as designed. Self-healing; noted for completeness.

Superseded (July 14, 2026 — §7d): under the keeperless redesign this behavior briefly escalated to a vault-bricking DoS (M-01) and was fixed by gating the deposit trigger on `positionLiquidity() > 0`; a pre-seed donation now sits retained until the position exists.

I-02 — Permit2 branch of `_execRoute` does not reset the residual ERC-20 approval

`_execRoute` resets the router allowance to 0 after the route in the plain-allowance branch (`ZapHelper.sol:339`) but the Permit2 branch leaves the helper→Permit2 ERC-20 allowance at whatever the route did not consume (`ZapHelper.sol:302`). This is benign in the current design: the helper holds no balances between calls (everything is swept), and the Permit2 sub-approval (router as spender) expires after 60 seconds. Asymmetric hygiene only; reset both branches for uniformity if touched again.

I-03 — Any tokens resting in `ZapHelper` are claimable by the first caller (by design)

`zap()` / `zapDeposit()` measure and sweep the helper's **entire** balance of the involved tokens (`ZapHelper.sol:149, 168–169, 302, 343–346`). Tokens accidentally transferred directly to the helper are therefore collectable by anyone who calls `zap` next. This is the intended "stateless, no custody" model and cannot touch user or vault funds, but it is worth stating in user-facing docs: **never send tokens directly to the ZapHelper address.**

I-04 — Vault shares report `decimals() == 0`

`VltUsdcVault.sol:329–331`. Intentional (shares are raw integer L units), and correct in that mint/burn/transfer semantics are unchanged. Expect cosmetic quirks in integrations that assume 18 decimals (some DEX UIs, portfolio trackers, `permit`-less approvals with "unlimited" shortcuts). Purely a display/integration consideration.

I-05 — `Snapshot.timestamp` is `uint32` (wraps in 2106)

`VltUsdcVault.sol:173`. The APR ring's timestamps overflow in February 2106. The ring is informational-only and touches no accounting; acceptable, noted for the record.

I-06 — External-token trust assumptions (USDC upgradeability & blacklist)

The vault hardcodes correct behavior for today's USDC (6 decimals, blacklist handled at `VltUsdcVault.sol:645–648`: a blacklisted redeemer's whole `redeem` reverts, harming only themselves). Residual risks outside the vault's control: USDC is an upgradeable proxy (Circle could in principle add hooks/fees, violating the vault's token assumptions), and a **blacklisted vault address** — however unlikely — would freeze the USDC leg of every redeem. Neither has a code remediation in an ownerless design; they are inherent to pairing with USDC and should be listed in user-facing risk docs. The VLT leg carries no equivalent risk (immutable token, §1).

I-07 — (obsolete — §7d removed the finder fee) Finder-fee base excluded retained balances

Conscious design of the original keeper model (then documented in-code and in the README): the 1% bounty was computed only on **freshly harvested pool fees**, while retained deposit/redeem-harvested fees and dust reinvested with no bounty. Consequence: the $1 gate could be satisfied almost entirely by retained balances while the bounty was near zero, so no rational keeper would call the then-public `compound()` and the retained value sat unreinvested until fresh pool fees accrued. Pure opportunity cost to holders, no loss.

Obsolete (July 14, 2026 — §7d): the finder fee was removed entirely — 100% of every harvest reinvests for holders and no keeper path exists, so this incentive-lag concern no longer has a subject.

I-08 — `ZapHelper` emits no events

`zap` / `zapDeposit` emit nothing. Vault events cover deposits, but off-chain accounting of zap routing, under-spends, and sweeps would benefit from a `ZapExecuted`-style event. (Consistent with the project's no-event-log-dependency stance for *reads*; emitting is still cheap and useful for indexers.) *Update (July 10, 2026):* the §7b change moved wallet attribution into the vault's own `Deposit` event (`recipient`), so the remaining benefit here is routing/under-spend observability only; the accepted decision (no helper events) stands.

I-09 — Dependency versions are caret ranges, not exact pins

`package.json` declares `^5.1.0` / `^1.0.2` for OZ and v4-core/periphery. `package-lock.json` pins the actual install, but for an audited artifact the convention is exact versions in `package.json` too, so the audited bytecode is reproducible from the manifest alone.

I-10 — A joining depositor's share price ignores retained/pending fees (bounded JIT-yield capture)

Surfaced by the July 9 adversarial re-audit. Shares are priced purely on position liquidity: `shares = supply * liquidityAdded / liqBefore` (`VltUsdcVault.sol:557`), with `liqBefore` snapshotted before the unlock (`:425`). The vault's **off-position** value — prior compound dust, redeem-retained fees, and the position's pending pool fees that `_onDeposit` harvests into the retained balance *during this very deposit* (`:636-669`) — is deliberately excluded from that price. This exclusion is exactly what defeats donation inflation (pricing on token balances would reopen it, see §7), so it is a **deliberate trade-off, not a coding error**. The side effect: a new depositor pays nothing for the retained/pending value yet, by becoming a holder, gains a pro-rata claim on it; when a later `compound()` folds that value into L, the joiner captures their fraction of value that accrued entirely before they joined, diluting prior holders by the same amount. `redeem()` is asymmetric in the holder-favoring direction (a leaver forfeits pending fees, `:703-726`), so the only extractable direction is deposit-then-hold-through-a-compound — classic JIT yield capture, not an atomic round-trip.

Why informational, not a vulnerability: the transferable amount is bounded by the *uncompounded* fee/dust balance at deposit time — there is no atomic profit and no solvency or theft vector, and capturing a meaningful fraction requires depositing capital comparable to the entire existing position for a dust-sized gain. Invariant #3 and the §7 donation analysis were tightened to cross-reference this.

Recommendation: accept and document as bounded JIT-yield capture — the current, and correct, choice. The theoretical alternative, crystallizing pending+retained fees into L at the top of `deposit()` (invoking the harvest-and-reinvest path before snapshotting `liqBefore`), is **not recommended**: it grafts a full compound onto every deposit, imposing three concrete costs to close a near-zero gap. (1) *Gas tax on every depositor* — the retained balance is almost always one-sided, so folding it in requires the rebalance `swap()` + `_addLiquidity`, i.e. the whole `compound()` body, on each deposit. (2) *Keeper cannibalization* — deposits would drain the fresh-fee pool the 1% finder bounty is computed on, so standalone `compound()` rarely clears gas and genuine keeper participation dries up, leaving a quiet market with no deposits uncompounded. (3) *Breaks the swap-free deposit invariant* — `deposit()` performs no swap today (only `compound()` swaps, in the vault's own pool); crystallizing drags a pool swap into the deposit path, giving deposits price-manipulation exposure (the L-01 cap would now have to guard deposits too) and making their gas/success depend on live pool state — eroding the deterministic, oracle-free deposit the design leans on. The gap it would close is already self-solving: I-10's magnitude is bounded by the *uncompounded* fee balance at deposit time, which under an active keeper is ~zero. The cheap, sufficient mitigation is a user-facing note that a keeper `compound()` immediately before a large deposit erases even the bounded case. **Do not** instead add vault token balances to the share price — that reopens the donation-inflation vector this design exists to prevent. Flagged for the Shieldify review so the trade-off is explicitly re-confirmed.

Superseded in part (July 14, 2026 — §7d): the redesign deliberately adopted a threshold-gated form of the "not recommended" alternative above: a deposit at ≥$100 claimable crystallizes pending value into L *before* its

shares are priced. Of the three costs cited, (1) the gas tax now falls only on threshold-crossing deposits, (2) keeper cannibalization is moot (no keeper or bounty exists), and (3) the swap-free-deposit invariant was consciously traded away — the L-01 cap and ≤5% price bound guard the leg's swap wherever it runs. I-10's residual window is therefore only the sub-$100 claimable balance.

---

7. Attack-surface analysis (verified safe)

Each of the classic vault attack classes was checked explicitly:

First-depositor share inflation — mitigated. First deposit requires `liquidityAdded > 1_000` and locks 1,000 shares at `0xdead` (`VltUsdcVault.sol:547–553`). Because shares are L-denominated and L/share can only be inflated by *donating value that compounds to all holders*, the attacker's inflation spend is recouped only pro-rata to their own holding; quantitatively, making a victim's worst-case rounding loss reach even $100 requires donating ≥ $100 × totalSupply-in-shares — astronomically unprofitable — and `minShares` independently reverts a rounded-to-zero mint.

Direct-donation inflation — mitigated. Shares are minted pro-rata to **ΔL reported by the PoolManager** (`VltUsdcVault.sol:557`), never to token balances; donated tokens are snapshotted as `retain0/1` *before* the deposit transfer (`VltUsdcVault.sol:523–524`) and excluded from the depositor's add, folding forward to all holders. (This ΔL-only pricing has one bounded side effect — a joining depositor's pro-rata claim on *pre-existing* retained/ pending fees — analyzed as informational finding I-10.)

V4 fee-sweep ("BUG-1") — correctly fixed in all three paths. V4 folds a position's full accrued fees into `callerDelta` on *any* `modifyLiquidity`. `_onDeposit` harvests fees to the vault before the depositor's add (`VltUsdcVault.sol:817–850`); `_onRedeem` pays the redeemer `callerDelta − feesAccrued` and takes the fees to the vault (`VltUsdcVault.sol:884–907`); `_onCompound` pays nothing out — 100% of the harvest reinvests (§7d). `test/vault.fees.test.js` covers all three.

Delta settlement — nets to zero on every path. Adds settle negative deltas (`sync → transfer → settle`, `VltUsdcVault.sol:1082–1090`), removes/harvests `take` positive deltas in full, and the swap settles only the input actually consumed under a partial fill (`VltUsdcVault.sol:1056–1079`). Any residual would revert the whole unlock (`CurrencyNotSettled`), which the passing flow tests exclude.

Reentrancy — guarded and structurally limited. Both entrypoints are `nonReentrant`; the unlock callback accepts only the PoolManager (`VltUsdcVault.sol:785–786`); the pool tokens have no transfer hooks (§1); the compound leg makes no external transfer to any account (§7d). The suite's `ReentrantToken` probes (reentrant `redeem` and reentrant `deposit` from a token hook) assert reentry reverts with `ReentrancyGuardReentrantCall`.

Redeem sandwiching — not value-extractable. Redemption is strictly in-kind and pro-rata; a manipulated price only shifts the token *split*, and by AM-GM the bundle at a displaced price is worth more, not less, at the true price. The absence of min-out bounds on `redeem` is therefore sound (and keeps exit ungrieffable).

Rounding directions — consistently favor the vault/holders. Deposit shares floor (`:453`), redeem liquidity floors (`:494`), V4 rounds removal amounts down, the `uint128` downcast in `redeem` is bounded by the position's own `uint128` liquidity, the sqrt-space 5% limit constants round inward ($9747^2 = 0.95004$, $10246^2 =$

1.04980), and `_liquidityForAmounts` takes the min of the two single-sided computations so an add can never demand more than the vault holds.

Access control — nothing to protect. There are no privileged functions; the constructor enforces `hooks == address(0)` and canonical token ordering (`VltUsdcVault.sol:278–284`), eliminating the hook-based attack surface entirely.

7a. Post-audit change (July 8, 2026): fee-accounting events for off-chain adapters

Reviewed as part of this engagement. Motivated by DefiLlama fees-adapter / TVL support: the original events could not account for realized pool fees from logs alone, because the BUG-1 fix makes `deposit()` / `redeem()` harvest the position's accrued fees to the vault **silently**, and `Compound` emitted only the finder's 1% cut — an events-only fee adapter would systematically under-report whenever the vault was actively used.

Changes (`VltUsdcVault.sol`, events section and callback handlers):

- `Compound` extended: `Compound(finder, fee0, fee1, finder0, finder1, liquidityAdded)` — now carries the FULL freshly harvested pool fees, with the finder amounts alongside.
- New `FeesRetained(uint256 fee0, uint256 fee1)`, emitted by `_onDeposit` and `_onRedeem` whenever a fee harvest is retained at the vault (only when non-zero).

Adapter contract: realized pool fees = Σ `Compound.fee0/fee1` + Σ `FeesRetained.fee0/fee1`; keeper revenue = Σ `finder0/finder1`; SupplySideRevenue = the difference; ProtocolRevenue = 0 (no protocol take exists). TVL needs no events: `previewRedeem(totalSupply())` plus the vault's token balances at a block.

Security review of the change: accounting-neutral — no storage, no control flow, no transfer logic touched; emits occur inside the unlock callback after calls into the trusted PoolManager only, under `nonReentrant` entrypoints (the pre-existing `reentrancy-events` justification now also covers `_onDeposit` / `_onRedeem`, suppressed inline with the same rationale as `_onCompound`). Event args honestly label their basis: `Compound.fee0/fee1` are the fresh harvest only (retained balances reinvested in the same call are reported by the earlier `FeesRetained` emissions that created them — no double count). Verified by a new regression test (`vault.fees.test.js` — "emits complete fee-accounting events…") asserting `FeesRetained` matches computed gross fees on both paths and `finder = fee / 100` exactly; suite at 47 passing, Slither 0, Solhint clean after the change. Frontend ABI (`src/js/vault-abi.js`) updated to match.

(Naming note: §7c later renamed these event fields to token-named `vltFees` / `usdcFees` / `vltFinder` / `usdcFinder` — same layout and topic0 hashes at that time. §7d then removed the finder fields entirely — `Compound(vltFees, usdcFees, liquidityAdded)`, new topic0 — and the identity simplified to supply-side revenue == realized fees.)

7b. Post-audit change (July 10, 2026): recipient/receiver params + consumed-amount deposit accounting

Motivated by per-wallet PnL tracking from logs alone: the original `Deposit` event attributed zapped entries to the ZapHelper (`user = msg.sender`) and reported the GROSS amounts pulled (pre-refund), so individual cost basis required joining raw ERC-20 `Transfer` logs.

Changes (`VltUsdcVault.sol` entrypoints/events/`_addLiquidity`, `ZapHelper.sol`):

- `deposit(vltAmount, usdcAmount, minShares, deadline, recipient)` — shares mint to `recipient` (`require != 0`). Tokens are still pulled from — and any imbalanced excess refunded to — `msg.sender` (the payer).
- `Deposit(address indexed sender, address indexed recipient, uint256 vltUsed, uint256 usdcUsed, uint256 sharesOut, uint128 liquidityAdded)` — `vltUsed`/`usdcUsed` are the amounts the pool actually CONSUMED (decoded from the unlock callback's settle amounts, not balance reads), so the event alone is exact cost basis.
- `redeem(shares, receiver)` — burns the CALLER's shares (no owner/operator model), pays both tokens to `receiver` (`require != 0`). `Redeem(address indexed owner, address indexed receiver, uint256 sharesIn, uint256 vltOut, uint256 usdcOut)` (token-named per §7c).
- `_addLiquidity` now returns `(liquidity, paid0, paid1)`; the compound path discards the paid amounts (its accounting is already carried by the `Compound` fee fields).
- `ZapHelper.zapDeposit(..., recipient, swapData)` passes the recipient through; the helper's share-forwarding transfer is removed (the helper never holds shares at any point). Dust sweeps still go to the caller, who paid.

Security review of the change: `recipient`/`receiver` are caller-chosen output destinations — the same trust class as `zap()`'s existing `recipient` (L-02): the caller can only spend their own tokens and burn their own shares, so misdirection is self-inflicted, never theft. Refunds deliberately stay with the payer so `deposit` gains no USDC-blacklist revert path on a third-party address; `redeem`'s blacklist note moves from caller to receiver (caller-chosen, and the caller can simply redeem to another receiver). No storage layout, share math, or settlement logic changed. Regression tests: `vault.flows.test.js` — "mints shares to `recipient`… CONSUMED amounts" (conservation: `used == pulled - refunded`) and "pays a designated receiver…"; `vault.edge.test.js` — zero-address reverts; `scripts.quoter.test.js` — "zapDeposit mints shares DIRECTLY to `recipient`…". Suite at 52 passing, Solhint clean. Dune queries/README (`periphery/dune`) updated to the new topic0 hashes and exact-volume semantics.

7c. Post-audit change (July 10, 2026): token-named external surface (no currency0/1 leakage)

Convention decision following §7b: the external surface previously mixed token-named fields (`Deposit.vltUsed/usdcUsed`) with pool-ordered ones (`Redeem.amount0Out/1Out`, `Compound.fee0/fee1`, `previewRedeem` returns), forcing every consumer to carry the `usdcIsCurrency0` mapping for half the fields. Since the vault is pair-specific by construction (immutable `vlt`/`usdc`, single pool), currency ordering is an

address-sort accident with no semantic content — so ALL events and external view returns are now token-named (`vlt*` 18d / `usdc*` 6d), and the currency0/1 ordering stays strictly internal (callbacks, `BalanceDelta` plumbing, `_rebalance`).

Changes (`VltUsdcVault.sol`):

- New internal `_toVltUsdc(amount0, amount1)` — the single mapping boundary, applied at every emit/return.
- `Redeem(owner, receiver, sharesIn, vltOut, usdcOut)`; `Compound(finder, vltFees, usdcFees, vltFinder, usdcFinder, liquidityAdded)`; `FeesRetained(vltFees, usdcFees)`.
- `redeem` returns `(vltOut, usdcOut)`; `previewRedeem` returns `(vltAmount, usdcAmount)`; `compoundClaimable` returns `(vltAmount, usdcAmount, valueUsdc, feesValueUsdc)`.
- Renames only — no types changed, so all four topic0 hashes are UNCHANGED (topic0 hashes arg types, not names); no storage, math, or control-flow changes.

Verification note: a currency-order mapping like `_toVltUsdc` is the kind of code that fails invisibly — when VLT sorts as currency0 the mapping is the identity, so a missed or inverted call site produces correct output in one ordering and silently swapped 18d/6d amounts in the other. The coverage is therefore pinned under both orderings: the test fixture accepts `forceUsdcIsCurrency0` (mock-token ordering is otherwise deploy-nonce dependent) and a dedicated suite (`vault.flows.test.js` — "token-named events/views hold under BOTH currency orderings") asserts every event/view field against per-token wallet deltas under BOTH orderings. Consumers updated in the same pass: Dune queries/README, DefiLlama fees + TVL adapters and fork harness, frontend ABI + test client. Suite at 54 passing (plus the 2 mainnet-fork tests), Solhint clean, Slither 0 findings (96 detectors, re-run July 10 post-§7b/§7c). *(§7d later reshaped `Compound` and removed the compound entrypoint + finder fields.)*

7d. Post-audit change (July 14, 2026): keeperless auto-compound redesign

The keeper model was removed end-to-end (branch `experiment/auto-compound-gas`, since merged to `main`). This section is the authoritative record of the delta; line citations throughout the document are re-anchored to the post-redesign source (`main` @ `2006f11`).

Design delta. - **No compound entrypoint of any kind.** The vault's external write surface is now `deposit` and `redeem`, full stop. *(Superseded by §7e: `donate` joined the surface on July 18 — still no compound entrypoint.)* The former `compound()` logic is the internal `_compound()` leg, reached only from `deposit()`: once `compoundClaimable().valueUsdc` reaches `AUTO_COMPOUND_MIN_USDC` **(100e6, a public ungoverned constant)** *and the position exists*, the deposit harvests + reinvests before its own liquidity is measured. `MIN_COMPOUND_VALUE_USDC` ($1 keeper gate) is removed. A small deposit is the manual way to force a compound in a quiet market. - **No fee of any kind.** `FINDER_FEE_BPS` and the finder payout are gone; 100% of every harvest reinvests for holders. The compound leg makes **no external transfer to any account** — its only token movements are settlements with the `PoolManager` (this deletes the CEI concern around paying an arbitrary caller). - **Event change (off-chain breaking):** `Compound(uint256 vltFees, uint256 usdcFees, uint128 liquidityAdded)` — topic0

`0x9bc6dc46ca3a56321f65a8e41f17e5a16077d04974dd396b6af9d0bb91c62094`. The §7a accounting identity simplifies: supply-side revenue == realized fees (no keeper term). DefiLlama adapters, the fork harness, and the Dune queries were updated in the same pass (the keeper-economics query was deleted). - **Reentrancy structure:** guards live only on the external entrypoints (`deposit`, `redeem`); `_compound()` executes under the calling deposit's lock. Hostile-token probes re-aimed accordingly: reentrant `redeem()` and reentrant `deposit()` from a token hook mid-deposit both revert with `ReentrancyGuardReentrantCall` (asserted byte-exact).

M-01 (Medium, introduced and FIXED during this redesign — never deployed): pre-seed donation permanently bricks the vault. Composing three individually-reasoned choices — (1) the auto-compound trigger inside `deposit()`, (2) shared fate (direct internal call, no try/catch), and (3) V4's revert on a zero-delta poke of a nonexistent position (I-01) — produced a griefing vector: a ≥$100 donation to a **virgin** vault would make every first-deposit attempt run a reverting compound, and since nothing can lower claimable value, the vault would be permanently unusable for a $100 outlay (no fund loss; redeploy possible). **Fix:** the trigger is gated on `positionLiquidity() > 0` (a load-bearing gate, documented in-code); pre-seed donations sit retained and fold into the first post-seed compound. **Regression test:** `vault.edge.test.js` — "a >= $100 donation to a virgin vault does not make the first deposit revert."

Accepted risk — shared fate (flagged for Shieldify). With no try/catch isolation, a reverting `_compound()` reverts every threshold-crossing deposit, and claimable value can only fall via a successful compound — a latent revert in the leg would therefore block deposits above the trigger permanently (redemption is unaffected; an orderly exit and redeploy remains possible). Accepted deliberately for simplicity: the leg is argument-free, its swap is double-bounded (L-01 cap + ≤5% price limit), and it is exercised by every threshold-crossing deposit in the suite and the 60-day fork simulation. **Audit focus request: revert-freedom of `_onCompound` / `_rebalance` / `_addLiquidity` across pool states.**

Property change — donation socialization (supersedes the §7 donation-inertness claim). Deposits now reinvest pending value before pricing shares, so a donation is socialized to pre-existing holders at the next triggering deposit instead of sitting inert. The first-deposit inflation analysis was re-derived: the depositor still receives exact ΔL-formula shares at the post-compound price (value-fair, `minShares`-guarded), donations mint nobody shares, and `MINIMUM_LIQUIDITY` keeps inflation griefing a ~1000:1 money-loser. The former exact-issuance test was replaced by a socialization-semantics test. This also narrows I-10's JIT-capture window: above the $100 trigger, pending value is crystallized into L *before* the joiner's shares are priced.

Lifetime fee counters (July 16, 2026): public `totalFeesVlt` / `totalFeesUsdc` added — informational, incremented via an internal `_recordFees` at all three fee-collection points (`_onDeposit` / `_onRedeem` retention pokes + the compound leg's fresh harvest), replacing the `_toVltUsdc` call each event site already made, so counters and events cannot drift (the identity `totals == Σ Compound + Σ FeesRetained` is asserted exactly in `vault.fees.test.js`). Compound-only recording was deliberately rejected: it would under-count every redeem and below-threshold-deposit harvest — the same under-reporting §7a fixed for events. The writes occur after trusted-PoolManager calls inside the guarded unlock (three `reentrancy-benign` suppressions extended with justification; Slither re-run: 0). Gas: ~+44k on the first triggering deposit (two cold SSTOREs), ~+10k warm; quiet deposits unchanged.

Cap removal (same day): `REBALANCE_CAP_MULT` and the fresh-fee sizing of the rebalance swap were removed — see the L-01 "Remediation revisited" note for the full rationale (ineffective: the add leg was never capped, per Codex I-02; harmful: one-sided claimable value re-triggered paid no-op compounds and idled outside redeemable principal). The rebalance now works the whole loose balance, bounded by the ≤5% price limit; the trigger and the leg share the same value base by construction. Slither/Solhint re-run post-removal: 0 results / clean; gas within noise of the table in `GASNOTES.md`.

Verification. 61 tests passing (fee-less compound asserted exactly: trigger depositor's wallet delta == deposit refund; Compound event == full gross harvest; NAV/share monotonic under pre-trigger price pushes; below-threshold deposits provably skip the leg). Gas: quiet deposit ~237.9k (+16.4k claimable view check vs. main), triggering deposit ~481.5k (vs. main's 418k keeper tx + 222k deposit as two transactions). See `GASNOTES.md`. **Slither re-run July 14 on the final keeperless code: 0 results (75 contracts, 96 detectors); Solhint clean.**

7e. Post-final-report additions (July 18–19, 2026): donate + modern vault/ERC-20 surface

Shieldify's final report (July 19: 11 Informational, 0 Medium/Low; fixes verified at `ce44c31`) closed the original round. Immediately after the fixes hash, five entrypoints were added — the vault was missing modern vault/ERC-20 features it can never gain post-deploy (immutable, ownerless). **Shieldify approved these additions on 2026-07-22; the remaining formality is their confirmation of the FINAL DEPLOYED contracts on mainnet (addresses below).** Code: `6a938e3` (donate/zapDonate), `2006f11` (permit/previewDeposit/zapRedeem). This section is the authoritative record.

- `donate(vltAmount, usdcAmount, donor, deadline)` — the sanctioned no-mint gift path: pulls the pair from `msg.sender`, adds max balanced liquidity at the pool price (reuses the deposit unlock callback verbatim — it never mints; share math lives in `deposit()`), refunds the short leg to the payer, mints no shares → L/share rises for all holders atomically. Guards: `nonReentrant`, deadline, both legs > 0, `donor != 0`, `totalSupply() > 0` (a pre-supply donation would be claimed by the first depositor), and the same $100 fold-first trigger as `deposit()`. Event `Donate(sender ix, donor ix, vltUsed, usdcUsed, liquidityAdded)` — sender=payer vs donor=attributed giver (zapped gifts distinguishable); the fee-accounting identity (fees = Σ Compound + Σ FeesRetained) is untouched. Closes the substance of final-report I-04: value routing uses `vault.donate()`, never `PoolManager.donate()` (binding constraint, pitch doc). **Documented JIT caveat:** L/share jumps atomically, so a large one-shot donation is front-runnable pro-rata (characterization-tested: 90% supply captures 90%); mitigation is operational — small tranches / private submission.
- `ZapHelper.zapDonate(usdcAmount, swapUsdcToVlt, minVltOut, deadline, donor, swapData)` — zapDeposit minus shares: USDC-only gift via the whitelisted route, credits `donor`, custody-free, `nonReentrant`.
- `ERC20Permit` **on the share token** (EIP-2612; domain "Bankroll VLT-USDC LP" / "1") — gasless share approvals; unaddable post-deploy.
- `previewDeposit(vltAmount, usdcAmount) → (shares, vltUsed, usdcUsed)` — entry-side quote symmetric with `previewRedeem`; exact-match tested against executed deposits (shares equal; consumed

amounts round UP as the pool charges, ±1 wei); pre-trigger state, `minShares` binding, never reverts.

- `ZapHelper.zapRedeem(shares, minUsdcOut, deadline, receiver, swapData)` + `zapRedeemWithPermit(..., v, r, s, ...)` — USDC-only exit: pull shares (tolerant try/catch permit in the WithPermit variant), redeem in-kind to the helper, sell the whole VLT leg via the route, deliver under an AGGREGATE `minUsdcOut`. The vault's `redeem` stays swap-free — exit purity holds; the sell leg exists only in the replaceable periphery. A zapped exit shows `owner = helper ≠ receiver` in the Redeem event (zapDeposit convention).

Verification: 85 tests (24 new across `vault.donate.test.js` and `vault.permit-preview-zapredeem.test.js`, incl. donate-path reentrancy probe and the JIT characterization), Slither 0, Solhint clean; every flow additionally executed against a mainnet fork through the browser client.

8. Static & dynamic analysis (re-run for this audit)

- **Slither** (local venv, `slither.config.json`, test contracts and `node_modules` filtered): **0 results** over 75 contracts with 96 detectors. The ~31 inline `slither-disable` directives in the two production contracts (26 `-next-line` + one `-start/-end` region in `VltUsdcVault.sol`, 4 in `ZapHelper.sol`, spanning ~10 distinct detector categories) were each re-reviewed and all are justified (intentional tick-spacing truncation, `== 0` guards, partial-tuple reads, wrapping fee-growth subtraction, trusted-callee event ordering under `nonReentrant`, including the `reentrancy-events` disables added for the §7a `FeesRetained` emits).
- **Solhint** security gate: wired (`npm run lint:sol`), clean per workspace config.
- **Hardhat suite: 42 passing, 2 pending** at audit time; **47 passing** after the L-01–L-03 remediations and the §7a event change (the pending pair are the opt-in `FORK=1` mainnet-fork tests). Coverage includes deposit/redeem/compound flows against a real deployed `PoolManager`, the fee-retention (BUG-1) tests, first-deposit inflation, reentrancy, USDC-blacklist behavior, the price-push-before-compound NAV-monotonicity test, the APR ring, and a 3-seed stateful invariant harness.
- **Suggested pre-mainnet additions** (echoing the README's own caveats): a Foundry `invariant_*` suite for thousands-of-runs fuzzing depth, and a fork test through the real Universal Router once the VLT mainnet route exists.

---

9. Security posture checklist

Item	Status
No owner / admin / privileged role	Pass
No upgradeability, `delegatecall`, or `selfdestruct`	Pass
No pause (exit always open)	Pass
Reentrancy guards on all entrypoints; callback caller-gated	Pass
Checks-effects-interactions (burn-before-unlock; compound leg makes no external transfers)	Pass
First-depositor inflation mitigated (dead shares + min first ΔL)	Pass

Item	Status
Donation-inflation mitigated (ΔL-based shares, retain snapshot)	Pass
V4 `feesAccrued` /principal split complete on every `modifyLiquidity` path	Pass
Oracle-free (no external price dependency in vault)	Pass
Hooks disallowed at construction	Pass
Slippage bounds where value-extractable (`minShares` , `minVltOut`)	Pass
SafeERC20 everywhere; no unchecked external-call returns	Pass
Slither clean / Solhint clean	Pass
Deadline parameters on time-sensitive entrypoints	Pass (fixed — L-03)
Refund routing in periphery `zap()`	Pass (fixed — L-02)
Rebalance exposure bounded (≤5% price limit; loose balances structurally trigger-scale — L-01 revisited, §7d)	Pass
Pre-seed donation cannot brick the first deposit	Pass (fixed — §7d M-01)
Exact dependency pinning in manifest	Fail (Info — I-09)
Independent third-party audit	Pending (Shieldify)

10. Conclusion

The vltUSDC vault is an unusually disciplined design: ownerless, oracle-free, single external dependency, shares denominated in a manipulation-resistant unit (L), in-kind exit that cannot be gated or sandwiched for value, and a periphery split that keeps the entire routing/MEV surface out of the solvency-critical core. The historically dangerous parts of this pattern — first-depositor inflation, donation inflation, and V4's fee-folding semantics — are all handled correctly and covered by targeted tests.

No Critical, High, or Medium issues were found in the July 8 review. (The July 14 keeperless redesign later introduced **and fixed pre-deployment** one Medium — M-01, §7d.) The three Low findings — a fresh-fee-scaled cap on the compound rebalance notional (later removed as ineffective — see L-01's "Remediation revisited"), refund routing in `zap()` , and deadlines on `deposit` / `zapDeposit` — **have all been fixed and regression-tested** (see the Resolution notes in §5; Slither 0, 47 tests passing post-fix). The informational items are predominantly conscious design choices that should simply be re-confirmed during the independent review.

Recommended before mainnet, in order: (1) ~~apply/accept L-01–L-03 explicitly~~ done; (2) complete the planned **Shieldify audit** with the README's flagged scope (V4 settlement paths, the fee split, the §7d keeperless auto-compound — shared-fate coupling and compound-leg revert-freedom — and the ZapHelper external-sourcing pattern); (3) re-run Slither + the full suite (and ideally a Foundry invariant pass) on the final commit; and (4) publish the audited commit hash and deployed addresses so users can verify source against bytecode.

Disclaimer

This review reflects analysis of the source code in `src/contracts/contracts/` as of the date above, the referenced prior reports, and the project's own test/static-analysis tooling executed locally. It does not guarantee the absence of vulnerabilities, does not cover deployed bytecode verification (nothing is deployed yet), and does not constitute financial advice. Modifications after this review — including fixes for the findings above — may introduce new issues and warrant re-review.