SECURITY

SECURITY REVIEW · PREPARED FOR

Bucket Launchpad

A bonding-curve token launchpad on Robinhood Chain

DATE	COMMIT	REMEDIATION
23.09.2026	95dbc5f2	41f4c42e

01 ABOUT / DISCLAIMER / RISK CLASSIFICATION

Phase & process

1. About SBSecurity

SBSecurity is a team of skilled smart contract security researchers. Based on the audits conducted and numerous vulnerabilities reported, we strive to provide the absolute best security service and client satisfaction. While it's understood that 100% security and bug-free code cannot be guaranteed by anyone, we are committed to giving our utmost to provide the best possible outcome for you and your product.

Book a Security Review with us at sbsecurity.net or reach out on Telegram @sbsecurity_audits.

2. Disclaimer

A smart contract security review can only show the presence of vulnerabilities **but not their absence**. Audits are a time, resource, and expertise-bound effort where skilled technicians evaluate the codebase and their dependencies using various techniques to find as many flaws as possible and suggest security related improvements. We as a company stand behind our brand and the level of service that is provided but also recommend subsequent security reviews, on-chain monitoring, and high whitehat incentivization.

3. Risk classification

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

3.1 IMPACT

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol's functionality.
- **Low** - funds are not at risk.

3.2 LIKELIHOOD

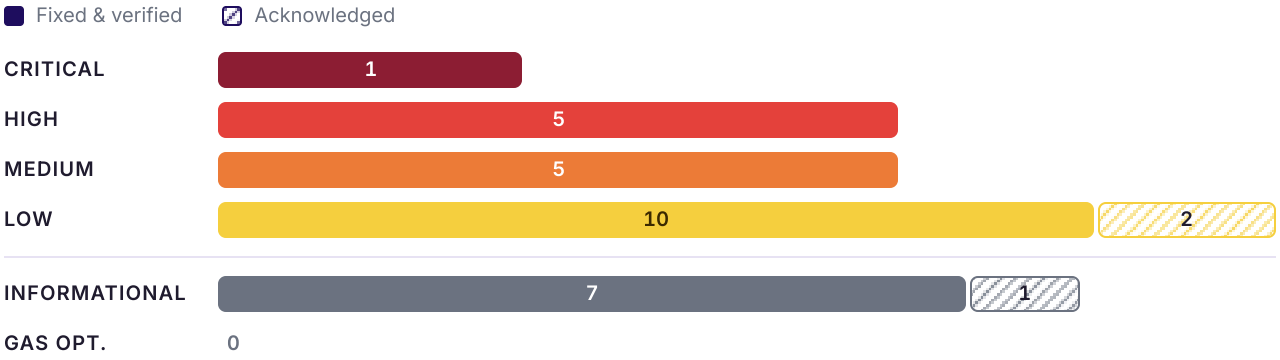
- **High** - almost certain to happen, easy to perform, or highly incentivized.
- **Medium** - only conditionally possible, but still relatively likely.
- **Low** - requires specific state or little-to-no incentive.

02 EXECUTIVE SUMMARY

Key findings

TOTAL FINDINGS	RESOLVED	HIGH SEVERITY RESOLVED
31	28 of 31 · 90%	6 of 6 · 100%

SEVERITY DISTRIBUTION



CLIENT	REPOSITORY	METHODS
Bucket Launchpad	Private	Manual Review, AI Review

TIMELINE · SEPTEMBER 2026

KICK-OFF	END OF AUDIT	REMEDIATIONS START	REMEDIATIONS END
16.09.2026	22.09.2026	22.09.2026	23.09.2026

STARTING COMMIT HASH	95dbc5f2101a1ee6b5e6b1f0dcea4ab3258fca2a
FINAL COMMIT HASH	41f4c42e2c5419b1b02ba9c05f1f98b5d0637005

CONTRACTS IN SCOPE

src/BackedToken.sol	src/CreatorVest.sol
src/CurveMath.sol	src/GraduationMath.sol
src/InitOnlyHook.sol	src/LaunchCurve.sol
src/LaunchDeployer.sol	src/LaunchFactory.sol
src/LaunchFeeSplitter.sol	src/LaunchMath.sol
src/LaunchRegistry.sol	src/LaunchToken.sol
src/LpTopUpErc20.sol	src/PoolSeeder.sol

03 PROJECT SUMMARY

What Bucket Launchpad is

Bucket Launchpad is a bonding-curve token launchpad on Robinhood Chain. A creator launches a fixed-supply ERC-20 that trades on a constant-product curve against an approved quote asset, and once the curve sells out the launch graduates into a permanently locked full-range Uniswap v4 position whose trading fees are split between the protocol, the creator and the holders' distribution engine on terms fixed at launch. A token can also hold a basket of backing assets that holders redeem for, and a launch that cannot open a pool switches to refund mode.

SECURITY POSTURE

BEFORE REMEDIATION 95DBC5F2

Refund mode and rescue

C-01 · H-01 · H-05 · M-02 · L-09 · L-12

One failed graduation attempt was enough to switch a launch at its bar into one-way refund mode, and refunds paid the live curve price with no minimum output, so early refunders were paid out of later ones. A rescue run inside a PoolManager unlock could force the same outcome in one transaction, and burns made before refunds opened left part of the pot unreachable.

Graduation and pool seeding

H-02 · H-03 · H-04 · L-04 · L-08 · L-10

The candidate pool keys for a graduation were predictable and could be taken for dust, by initialising them with a little liquidity or by filling a boundary tick to the per-tick cap, and a squatted key reverted the creator's whole launch. A stale balance in the shared seeder read as a squat too, the seeding path assumed plain ERC-20 transfers, and a lock owner could pass a liquidity value that removed principal.

AFTER REMEDIATION 41F4C42E

At the final commit the Bucket team had fixed 28 of the 31 findings. Refunds pay one rate fixed when they open, a rescue must fail twice an hour apart with proof that every candidate key is held by funded liquidity, graduation refuses to run inside a PoolManager unlock, and every graduated pool key carries an init-only hook so only the seeder can open it. The seeder settles only its own balance delta, redemption legs are isolated so one refusing asset cannot block the basket, refunds redeem backing on the holder's behalf, the LP top-up is bounded by a price band and a deadline, vesting top-ups restart the schedule with the received amount, and the deploy scripts verify every wiring step after broadcast. Each fix was re-run against the proof-of-concept test written for the finding, and the unit and invariant suites pass at the final commit. The three acknowledged findings (L-05, L-07, I-04) were reviewed and accepted by the team.

BEFORE REMEDIATION 95DBC5F2

AFTER REMEDIATION 41F4C42E

Backing and redemption

M-03 · M-04 · L-01 · L-06 · L-11 · I-07

Redemption moved every backing asset in one call, so a single frozen or non-ERC-20 asset blocked the whole basket. Refunding a backed launch parked the tokens instead of burning them, which stranded their backing claims, a burn could take the supply to zero while backing was held, dust redemptions burned for nothing, and a launch could name its own token as a backing asset.

Fee splitting and LP top-ups

M-05 · L-02 · L-05 · L-07 · I-05

The splitter's LP top-up relied on a minimum-liquidity floor that could not stop a sandwich, since the attack moves the price rather than the quantity. Rotating the creator recipient orphaned its unpaid arrears, the protocol and distributor recipients were fixed for life, a fixed gas cap could leave arrears unpayable, and any token sent to the splitter was paid out as fee income.

Creator vesting

M-01 · L-03 · I-03 · I-06

A top-up to an existing vest was partly claimable at once because the schedule kept its original start, the vest recorded the requested amount rather than the received one, a lock was not tied to a launch or its creator, and a fully claimed vest could never be re-locked.

Consistency, trust and deployment

I-01 · I-02 · I-04 · I-08

The curve and the factory enforced different fee ceilings, every recipe rule reverted with the same error, the accepted quote assets carried their issuers' freeze, burn and upgrade powers, and the deploy scripts had gaps in their post-deploy checks.

04 RECOMMENDATIONS

Beyond this audit

Measures that extend the protection of this review after launch, independent of any individual finding.

R-01	**Monitor the launch lifecycle on chain**	Index the registry and curve events and alert on rescue attempts, refund openings, failed auto-graduations and payouts that land in arrears. A launch that sits at its bar or a fee leg that goes unpaid should be visible to the team within minutes.
R-02	**Publish asset requirements for creators**	Document which quote and backing assets the launchpad supports and what they must look like: plain ERC-20s with no transfer fee and no rebasing. Track issuer announcements for every asset on the allow list so a change in behaviour is known before it affects a launch.
R-03	**Keep the fee recipient keys under strong custody**	Hold the protocol and distributor recipient addresses in hardware-backed multisigs with a written custody procedure, and document the creator recipient transfer flow for creators so a planned change of address is routine.

06 CONTRACTS & ROLES

Core contracts

CONTRACT	ROLE
`LaunchFactory.sol`	Entry point for every launch. Creates the token, curve and registry record, runs the creator's opening buy, and later graduates the launch by draining the curve into a locked Uniswap v4 position behind a fee splitter. Also holds the rescue path that opens refunds when no pool can be opened.
`LaunchCurve.sol`	The pre-market for one launch: a constant-product curve with a virtual quote reserve. Handles buys, sells, the decaying snipe tax, fee sweeps to the three recipients, and the refund path once refunds are open.
`LaunchToken.sol`	The fixed-supply ERC-20 minted for a launch, with an optional basket of backing assets that holders redeem tokens for pro rata. No owner, no mint and no transfer hook.
`LaunchRegistry.sol`	Append-only record of every launch: its curve, terms and phase, and after graduation its pool, lock and splitter. Also tracks the creator recipient and its three-day propose and accept transfer.
`LaunchFeeSplitter.sol`	Owns a graduated launch's locked position and splits collected trading fees between the protocol, the creator and the distributor on immutable terms. Takes the distributor's LP top-ups and keeps arrears per recipient.
`LpTopUpErc20.sol`	The permanent full-range liquidity lock for an ERC-20 pair on Uniswap v4. Its owner can add liquidity and collect fees. Principal has no exit path.

CONTRACT	ROLE
`PoolSeeder.sol`	Opens a pool and seeds it in one transaction: initialises at the launch price, deploys the lock, adds the full-range seed and hands the lock to its owner. Only the factory may open a key that carries the launchpad hook.
`InitOnlyHook.sol`	A minimal Uniswap v4 hook that implements only beforeInitialize and admits a single initialiser, the seeder. Once the pool is open it is never called again, so the pool trades like a plain one.
`LaunchDeployer.sol`	Holds the creation code for the per-launch curve, token and splitter so the factory stays under the contract size limit. Callable only by the factory.
`CreatorVest.sol`	Opt-in linear vesting for launch creators. The current creator recipient locks tokens for 30 days to four years, and they release linearly to the locker only, with no admin and no early exit.
`BackedToken.sol`	A standalone fixed-supply token backed by a basket of assets it holds. Supply only goes down, through burns and pro-rata redemptions.
`CurveMath.sol`	Constant-product quote math for the curve.
`GraduationMath.sol`	Derives the pool's opening price from the curve's closing reserves.
`LaunchMath.sol`	Shared price and full-range liquidity math for seeding and top-ups.

06 CONTRACTS & ROLES

Privileged roles

Launch creator.

Chooses the quote asset, tier, trade fee, routing and recipes at launch and receives the creator share of fees. The creator recipient can be moved through a three-day propose and accept transfer in the registry, and may lock tokens in CreatorVest.

Protocol recipient.

Receives the launchpad's share of every trade fee, on the curve and in the splitter. Fixed when the factory is deployed and has no other powers.

Distributor.

The engine wallet that receives the holders' share of fees and converts it into the rewards recipe off chain. It is the only caller of the splitter's LP top-up.

Deployment address.

Loads the accepted quote assets and their tiers into the factory and seals the list, and names the factory as the only caller that may open a hooked key in the seeder. Both are one-time actions, after which the address has no powers and the factory has no owner.

Lock owner.

Owns a launch's liquidity lock, which is the fee splitter for every launch graduated through the factory. It can add liquidity and collect fees. Principal cannot be withdrawn by anyone.

07 FINDINGS

What we found

ID	TITLE	SEVERITY	STATUS
C-01	Forced refund mode lets the first refunder rug late buyers for dust	CRITICAL	FIXED
H-01	`refund` payout depends on the order of the calls	HIGH	FIXED
H-02	A graduation pool key can be squatted for a few thousand wei	HIGH	FIXED
H-03	A stale balance in the shared Seeder forces healthy launches into refunds	HIGH	FIXED
H-04	Filling the boundary tick to Uniswap v4's per-tick liquidity cap permanently bricks graduation for dust	HIGH	FIXED
H-05	A crossing buy and rescue run inside a PoolManager unlock force a healthy launch into refund mode atomically	HIGH	FIXED
M-01	`topUp` releases part of the new deposit immediately	MEDIUM	FIXED
M-02	One failed `rescue` attempt opens refunds for good	MEDIUM	FIXED
M-03	One frozen backing asset blocks redemption of the whole basket	MEDIUM	FIXED
M-04	Refunding a Backed Launch Permanently Strands Backing Claims	MEDIUM	FIXED

ID	TITLE	SEVERITY	STATUS
M-05	The `minLiquidity` Mitigation for LP Top-Up Sandwiching Is Dimensionally Incapable of Working	MEDIUM	FIXED
L-01	`burn` can take the supply to zero while backing is still held	LOW	FIXED
L-02	Rotating the creator recipient orphans its unpaid arrears	LOW	FIXED
L-03	`CreatorVest` records the requested amount, not the received amount, in a shared balance	LOW	FIXED
L-04	The seeding path and the opening buy are not fee-on-transfer safe	LOW	FIXED
L-05	`protocol` and `distributor` recipients cannot be rotated	LOW	ACKNOWLEDGED
L-06	`LaunchToken.redeem` burns tokens for a zero payout	LOW	FIXED
L-07	The splitter pays out any balance it holds as fee income	LOW	ACKNOWLEDGED
L-08	A squatted key reverts the creator's launch instead of leaving it Climbing	LOW	FIXED
L-09	`refund` has no `minQuoteOut`	LOW	FIXED
L-10	A lock created through the permissionless PoolSeeder can have its permanent principal withdrawn by its owner	LOW	FIXED
L-11	A Non-ERC-20 Backing Leg Bricks Every Backing View and Redemption	LOW	FIXED

ID	TITLE	SEVERITY	STATUS
L-12	Pre-Refund Burns Leave Curve Quote Permanently Unreachable	LOW	FIXED
I-01	`LaunchCurve` and `LaunchFactory` enforce different ceilings for the same `feeBps`	INFORMATIONAL	FIXED
I-02	`_checkRecipe` reverts with the same `BadRecipe` error for every rule	INFORMATIONAL	FIXED
I-03	`CreatorVest.lock` is not tied to a launch or to its creator	INFORMATIONAL	FIXED
I-04	Allowed Quote Assets Retain Issuer Freeze, Block, Burn, and Upgrade Trust	INFORMATIONAL	ACKNOWLEDGED
I-05	The Fixed Payout Gas Cap Can Make Arrears Permanently Unpayable	INFORMATIONAL	FIXED
I-06	A Fully Claimed CreatorVest Can Never Be Re-Locked	INFORMATIONAL	FIXED
I-07	A Launch Can List Its Own Token as Its Backing Asset	INFORMATIONAL	FIXED
I-08	Deployment Script Review	INFORMATIONAL	FIXED

CRITICAL SEVERITY · 1 FINDING

C-01 CRITICAL FIXED

Forced refund mode lets the first refunder rug late buyers for dust

Three weaknesses chain into one attack:

1. `seed` treats any nonzero liquidity as a funded squat, so all 8 candidate graduation keys can be taken for a few thousand wei each.
2. `rescue` opens refunds after one failed graduation attempt and there is no way back.
3. `refund` pays the curve price at the moment of the call, so the first refunder takes the most and the last takes what is left.

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L459-L468

```
if (gasleft() < MIN_RESCUE_GAS) revert NotEnoughGas();
try this.selfGraduate(token) {
    return;
} catch (bytes memory err) {
    // empty returndata is out-of-gas or a failed CREATE; NotEnoughGas is
    // the seed loop saying the same about one of its eight attempts
    if (err.length == 0 || bytes4(err) == NotEnoughGas.selector) revert NotEnoughGas();
    registry.markRefunding(token);
    c.openRefunds();
}
```

The creator runs it end to end: launch with a small opening buy at the floor price (snipe-exempt), call `seed` with `liquidity = 1000` on all 8 keys, wait for buyers to fill the curve, call `rescue`, then call `refund` before anyone else. Steps 1 and 2 are permissionless, so a bystander can push any launch into refund mode too. The profit only needs an early bag, which the creator always has.

IMPACT

Any launch that has reached its graduation bar can be forced into refund mode for dust, permissionlessly, and two harms follow.

The first is a denial of graduation. Steps 1 and 2 need no stake in the launch, so a bystander can brick any at-bar launch into one-way refund mode for about 20,000 wei plus gas. The squat costs 16,000 wei of quote and 4,000 wei of tokens in the PoC. The market the buyers paid for never opens and refund mode never reverts.

The second is the creator rug. Refund mode replaces a tradeable graduated token with a fixed pot that empties first-come, so late buyers race a shrinking pot instead of holding a position that could recover. In the PoC the creator pays 200 quote and is refunded 1,678, and the last buyer paid 1,850 and gets 225 back, a loss of 88%. Most of that multiple is the value of a floor-price bag, which the creator would keep even by graduating and dumping into the pool: the repo's own test `test_INFO_rescue_refund_versus_dumping_into_the_graduated_pool` measures the refund at 2,868 against 2,452 for dumping the same bag into the graduated pool, about 17% more. What the attack adds is that the creator can force this outcome for dust rather than hope for it, and that the late buyers' loss is immediate and permanent rather than a position they could hold. The repo's `test_ATTACK_creator_squats_own_launch_and_rugs_late_buyers` shows the same rug with a fully funded squat; this finding shows the squat is close to free. The promise of `rescue` that "money is never trapped, whatever an attacker does" becomes the rug vector of the launchpad.

C-01 · CONTINUED

RECOMMENDATION

The refund is forced only because a graduation key can be denied for dust and a single failed attempt is treated as permanent. Close those two, in order.

1. Make a squat cost real capital. The gate in `seed` is a magnitude blind check, so 1,000 units of full range liquidity, a few thousand wei, counts as a funded squat:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/PoolSeeder.sol#L95

```
if (poolManager.getLiquidity(key.toId()) == 0) revert PriceAlreadySet(current, sqrtPriceX96);
```

Require a minimum liquidity worth defending before a key is conceded, so a dust squat is pushed aside for dust and only a squat with real capital across all eight keys can deny a launch. Walking a wrong price pool to `sqrtPriceX96` the way the empty case already is would also work, but it must handle liquidity funded off the current tick, which `getLiquidity` does not see.

2. Do not open refunds on a single failed attempt. Let the first failed `rescue` record a timestamp and return, and open refunds only if a second attempt after a delay fails as well. A launch that can graduate later then does, and a one block condition no longer ends it.

These two stop the refund from being forced, which is what the attack depends on. The refund pricing is a smaller, separate question and paying one pooled average rate is not a safe change to make: the team removed the pooled average on purpose because the average itself over-paid an early buyer out of a late buyer's principal, and the tests `test_FIXED_refund_pays_the_curve_price_not_a_pooled_average` and `test_going_first_in_a_refund_is_worth_no_more_than_selling` encode that decision, so a pooled rate would fail them.

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchCurve.sol#L349-L354

```
// Unwind at the CURVE price, not a pooled average. Paying everyone the
// volume-weighted average handed an early buyer several times their
// entry out of a late buyer's principal, which made forcing a refund
// profitable and turned every denial-of-graduation trick into a rug.
// Priced exactly like sell(), with the fee waived: this is a wind-down,
// not a trade.
```

The only refund scheme that removes the early to late transfer entirely is to pay each holder their own cost basis and share any shortfall pro rata, which needs per buyer accounting the curve does not keep today. That is worth weighing only after the two changes above, and it is not required to close this finding.

HIGH SEVERITY · 5 FINDINGS

H-01 HIGH FIXED

`refund` payout depends on the order of the calls

`refund` prices each call like a sell against the live reserves, with the fee waived:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchCurve.sol#L355-L366

```
(uint256 qRes, uint256 tRes) = getReserves();
quoteOut = CurveMath.getAmountOut(tokensIn, tRes, qRes, 0);
// ... (comments omitted)
uint256 outstanding = launchSupply - trackedTokens;
if (tokensIn ≥ outstanding || quoteOut > trackedQuote) quoteOut = trackedQuote;
IERC20(token).safeTransferFrom(msg.sender, address(this), tokensIn);
trackedTokens += tokensIn;
trackedQuote -= quoteOut;
```

Once refunds are open the pot (`trackedQuote`) is fixed and nobody can buy. Every refund lowers `trackedQuote` and raises `trackedTokens` , so each later call is priced worse than the one before. What a holder gets back depends only on when their transaction lands, not on what they paid. In the PoC the same buyer, with the same tokens, is refunded 2,029 if first and 264 if last.

The comment above the code explains that the curve price was chosen over a pooled average so an early buyer cannot profit from a forced refund. It does not achieve that. An early buyer who is also first in line profits more, not less: in the repo's own `test_ATTACK_creator_squats_own_launch_and_rugs_late_buyers` the creator is refunded 5.7x their opening buy and the worst hit buyer loses 91%. Every other holder loses to whoever is fastest.

IMPACT

Refund mode is the launchpad's safety valve, and in practice it is a race. The first refunder gets far more than a pro-rata share and the last gets a fraction, so holders are pushed to front-run each other and bots can extract the difference. Since every graduation key can be taken for dust and a single failed `rescue` call opens refunds, the party who forces the refund can also be the first to refund.

RECOMMENDATION

Pay one rate to everyone, fixed when refunds open. Snapshot the pot and the outstanding tokens in `openRefunds` and pay `refundQuote * tokensIn / refundTokens` in `refund` . Keep the last-holder sweep. This removes the race and the front-running incentive. Early buyers still get the average price for tokens they bought cheaper, which is bounded and the same for everyone. Making sure a refund cannot be forced is a separate change: squatting a key must cost real capital, and `rescue` must not open refunds on a single failed attempt.

H-02 HIGH ● FIXED

A graduation pool key can be squatted for a few thousand wei

When a candidate pool key is already initialised at another price, `seed` gives up as soon as the pool holds any liquidity at all:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/PoolSeeder.sol#L86-L99

```
} else if (current ≠ sqrtPriceX96) {
    // ... (comments omitted)
    // Squatting must cost real, at-risk capital, so we only give up
    // when the squatter has actually funded the pool.
    if (poolManager.getLiquidity(key.toId()) ≠ 0) revert PriceAlreadySet(current,
   sqrtPriceX96);
    poolManager.unlock(abi.encode(key, sqrtPriceX96));
    (uint160 moved,,,) = poolManager.getSlot0(key.toId());
    if (moved ≠ sqrtPriceX96) revert PriceAlreadySet(moved, sqrtPriceX96);
}
```

`seed` itself is permissionless and the caller chooses `liquidity`. Calling it with `liquidity = 1000` at a wrong price puts 2,500 wei of tokens into a full-range position and makes `getLiquidity` nonzero, so the check is satisfied for dust. The factory tries the 8 spacings in `TICK_SPACINGS`, so 8 such calls, roughly 20,000 wei plus gas, deny every candidate and `rescue` is the only path left. A single-sided position parked off-tick through the pool manager works the same way and needs only the quote token.

IMPACT

Any launch can be denied its market for well under a cent. Its buyers are pushed into refund mode, where the payout depends on who refunds first, and the squatter can be the one who profits from it. This is the exact griefing the comment says the check prevents.

RECOMMENDATION

The keys are public, so squats cannot be prevented, only made expensive or pushed aside. When the pool holds liquidity at another price, do not revert right away. Swap toward `sqrtPriceX96` with a small budget, for example at most 1% of `max0` and `max1` plus a cap on tick crossings, and seed if the price lands on the target, recomputing `liquidity` from the balances left. A dust squat is then displaced for dust, and the squatter ends up selling into the launch below its price. A squat large enough to exceed the budget has real capital at risk on every one of the 8 keys, which is what the comment intended. Also let `rescue` open refunds only after a second failed attempt separated by a delay, so a remaining failure does not end the launch at once.

H-03 HIGH ● FIXED

A stale balance in the shared Seeder forces healthy launches into refunds

`PoolSeeder` is a shared, permissionless singleton used by every launch. `seed()` does not isolate the balance movement of its own call: it refunds the contract's **total** balance and derives the consumed amount from that same total.

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/PoolSeeder.sol#L106-L117

```
t0.safeTransferFrom(msg.sender, address(this), max0);
t1.safeTransferFrom(msg.sender, address(this), max1);
t0.forceApprove(address(lock), max0);
t1.forceApprove(address(lock), max1);
lock.addLiquidity(liquidity, max0, max1);   // consumes U ⩽ max
lock.transferOwnership(owner);

uint256 r0 = t0.balanceOf(address(this));    // TOTAL balance, not the delta
uint256 r1 = t1.balanceOf(address(this));
if (r0 > 0) t0.safeTransfer(msg.sender, r0);
if (r1 > 0) t1.safeTransfer(msg.sender, r1);
emit Seeded(..., max0 - r0, max1 - r1);      // TOTAL balance again
```

Let `D` be the balance already sitting in the Seeder before the call, `M` the `max` supplied by the factory, and `U` the amount the v4 position actually consumes. The unused part `M - U` is returned to the Seeder by the new lock, so line 113 reads `r = D + (M - U)` and line 117 computes:

```
M - r = M - (D + M - U) = U - D
```

Whenever `D > U` this underflows and reverts with `Panic(0x11)`.

There is a second, independent place from which `D` can enter the same subtraction. `LpTopUpErc20.addLiquidity()` also refunds its **total** post-call balance instead of the balance delta belonging to the current call:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LpTopUpErc20.sol#L80-L83

```
uint256 r0 = t0.balanceOf(address(this));
uint256 r1 = t1.balanceOf(address(this));
if (r0 > 0) t0.safeTransfer(msg.sender, r0);
if (r1 > 0) t1.safeTransfer(msg.sender, r1);
```

The next lock address is deterministic (`CREATE` from the public Seeder nonce). An attacker can transfer `D` quote to that still-empty counterfactual address before graduation. Once the lock is deployed there, it starts with `D` , receives `M` , consumes `U` , and sends `D + M - U` back to the Seeder. The Seeder therefore reaches the same `U - D` subtraction even though its own starting balance was zero.

This remains persistent across the whole eight-key loop: the panic reverts the lock creation and the Seeder nonce increment, so every catch/retry creates the lock at the **same preloaded address**. The pre-existing ERC-20 balance lives in the quote token contract, outside the reverted sub-call, and survives. Consequently, a balance snapshot applied only to `PoolSeeder` is incomplete: the foreign balance arrives during its call and looks like a fresh inflow unless the lock isolates its own starting balance too.

Two properties turn this arithmetic fault into a denial of service.

First, a pre-existing `D` is not part of the reverted sub-call's state changes, so it survives the revert. All eight candidate

H-03 · CONTINUED

tick spacings are then attempted and fail identically against the same stale `D`. The in-code justification for the retry loop is falsified here:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L369-L385

```
// A failed attempt reverts its own sub-call and leaves no state behind,
// so trying is free and honest.
```

The sub-call rolls back its **own** effects, but it cannot roll back foreign state.

Second, both handlers classify errors too coarsely. `_graduate()` only special-cases empty returndata (gas starvation) and `continue`s on any non-empty error, so a panic is indistinguishable from a legitimately occupied key. After eight attempts it raises `NoUsablePool()`. `rescue()` then treats any error that is neither empty nor `NotEnoughGas` as proof that no market can be opened, and calls `openRefunds()`:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L455-L468

```
try this.selfGraduate(token) {
    return;
} catch (bytes memory err) {
    if (err.length == 0 || bytes4(err) == NotEnoughGas.selector) revert NotEnoughGas();
    registry.markRefunding(token);
    c.openRefunds();
}
```

A pure accounting mistake is thereby promoted to an irreversible terminal state.

The trigger is a single unprivileged `transfer`, either into the shared Seeder or into its deterministic next lock address. There is no owner, no pause, and ERC-20 recipients cannot reject an unsolicited transfer.

IMPACT

An attacker can permanently deny market creation to a fully healthy launch, with no pool squatting of any kind and no privileged access.

This is not a re-report of #1. #1 covers how value moves once refund mode is entered; this is a refund **trigger** that no existing defence covers, and it is distinct from #2 and #13 because it requires no squat: all eight candidate keys are untouched when the poison fires.

- The launch reaches its graduation bar but can never open a v4 pool. The registry records `Phase.Refunding` and `poolId == bytes32(0)` permanently, and `graduate()` reverts with `NoUsablePool()` forever. Buyers are forced to exit on the bonding curve instead of into a public market.
- The attack is effectively free and repeatable. The poison capital is never spent, only parked: it survives unlimited graduation attempts, `rescue()`, and a complete refund wind-down without losing a wei. Costs are gas plus a temporary deposit the attacker controls. The N-6 reclaim primitive is not required, because the capital was never at risk.
- One deposit affects many launches. Because the Seeder is a global singleton, one poisoned balance blocks every launch sharing that quote whose graduation-side consumption is smaller than `D`. A single deposit was measured denying three independent launches.
- Monitoring the Seeder alone is insufficient. The counterfactual-lock variant keeps `quote.balanceOf(PoolSeeder) == 0` throughout. A direct Seeder poison can also be cleared afterwards, so a clean Seeder balance does not prove the attack never happened.
- No hot-fix is possible. The factory is deployed, sealed and unowned.

Note on profit: the severity does not rest on profitability. Of the `2368.385846` USDG net figure obtained by combining

H-03 · CONTINUED

this with #1, roughly 82% is ordinary curve appreciation available without any attack. Against the same `500e18` cost basis, an honest world that graduates and sells into the real pool returns `2452.13e18`, versus `2868.72e18` for the forced refund, so this bug's own contribution is approximately `417e18` (18%). The severity rests on permissionless, squat-free, capital-preserving, cross-launch, irreversible denial of market creation.

RECOMMENDATION

Settle only the balance delta produced by the current call **at both layers**.

`PoolSeeder` must snapshot its balances before pulling the caller's funds and refund only `post - pre`:

```
uint256 pre0 = t0.balanceOf(address(this));
uint256 pre1 = t1.balanceOf(address(this));

t0.safeTransferFrom(msg.sender, address(this), max0);
t1.safeTransferFrom(msg.sender, address(this), max1);
// approve + addLiquidity

uint256 post0 = t0.balanceOf(address(this));
uint256 post1 = t1.balanceOf(address(this));
uint256 refund0 = post0 - pre0;
uint256 refund1 = post1 - pre1;
if (refund0 ≠ 0) t0.safeTransfer(msg.sender, refund0);
if (refund1 ≠ 0) t1.safeTransfer(msg.sender, refund1);
emit Seeded(..., max0 - refund0, max1 - refund1);
```

`LpTopUpErc20.addLiquidity()` must likewise snapshot its own balances before `safeTransferFrom` and return only its call-local remainder, never the total post-call balance. Better still, return the exact principal deltas from `modifyLiquidity()` and make the Seeder use those values with an explicit `used ⩽ max` invariant. A Seeder-only `post - pre` patch is bypassed by the counterfactual-lock vector because `D` reaches the Seeder after its snapshot.

Foreign balances then cannot participate in either layer's subtraction. This matches the delta-accounting pattern already used in `LaunchCurve.buy()` (`// fee-on-transfer safe`). Moving the event before the transfers does not fix the defect: the operand, not the ordering, is wrong. Any recovery of mis-sent funds should be separate from `seed()` accounting.

Tighten graduation error classification to an allow-list. `_graduate()` should only skip errors that unambiguously mean a key is occupied (`PriceAlreadySet` and `PriceMoveNotFree`); panic, transfer, deployment and unknown errors must bubble up. `rescue()` should open refunds only after positively verifying that every candidate key is occupied by funded liquidity, rather than treating any internal error as evidence.

Fix the unchecked `uint256` to `int256` cast in `LpTopUpErc20.addLiquidity()` as well: an owner can pass `2^256 - L`, which the callback reads as `-L`, and remove a position documented as permanent. This does not affect the repeatability of this finding because the poison principal is never consumed.

Finally, document that unsolicited ERC-20 balances cannot be rejected. Once accounting is isolated, such balances may remain stranded unless a separate, non-interfering recovery mechanism is provided.

H-04 HIGH ● FIXED

Filling the boundary tick to Uniswap v4's per-tick liquidity cap permanently bricks graduation for dust

Every graduation seeds a single full range position at `minUsableTick` and `maxUsableTick` of one of eight hard-coded tick spacings. The lock mints the position here:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LpTopUpErc20.sol#L97-L106

```
(BalanceDelta delta,) = poolManager.modifyLiquidity(
    _key,
    IPoolManager.ModifyLiquidityParams({
        tickLower: TickMath.minUsableTick(_key.tickSpacing),
        tickUpper: TickMath.maxUsableTick(_key.tickSpacing),
        liquidityDelta: liquidity,
        salt: 0
    }),
    ""
);
```

The factory walks eight candidate spacings and takes the first key whose seed does not revert:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L374-L386

```
for (uint256 i = 0; i < TICK_SPACINGS.length && !seeded; i++) {
    (PoolKey memory k,) = _keyWithSpacing(token, l.quote, l.feeBps, TICK_SPACINGS[i]);
    uint128 liq = LaunchMath.fullRangeLiquidity(sqrtP, k.tickSpacing, amount0, amount1);
    if (liq == 0) continue;
    try seeder.seed(k, sqrtP, liq, amount0, amount1, address(this)) returns (LpTopUpErc20 lk) {
        key = k; lock = lk; seeded = true;
    } catch (bytes memory err) {
        if (err.length == 0) revert NotEnoughGas();
        continue;   // squatted and funded in range; the next key may be free
    }
}
if (!seeded) revert NoUsablePool();
```

Uniswap v4 caps the gross liquidity referenced by any single tick. When a `modifyLiquidity` call would push a tick's `liquidityGross` above `tickSpacingToMaxLiquidityPerTick`, the pool reverts `TickLiquidityOverflow`:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/lib/v4-core/src/libraries/Pool.sol#L165-L172

```
uint128 maxLiquidityPerTick = tickSpacingToMaxLiquidityPerTick(params.tickSpacing);
if (liquidityGrossAfterLower > maxLiquidityPerTick) {
    TickLiquidityOverflow.selector.revertWith(tickLower);
}
if (liquidityGrossAfterUpper > maxLiquidityPerTick) {
    TickLiquidityOverflow.selector.revertWith(tickUpper);
}
```

For spacing 200 that cap is about `3.83e34`. A launch's own seed liquidity for the deployed tiers is twelve to seventeen orders of magnitude below it, so the seed alone never approaches the cap. But the cap is per tick across all positions, and the seed always references the two boundary ticks. An attacker mints a cap sized band that shares one of those boundary

H-04 · CONTINUED

ticks, one tick spacing wide, sitting out of range so it holds only the launch token and costs dust. `seed` gives up only when the pool holds liquidity at the current tick:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/PoolSeeder.sol#L83-L99

```
(uint160 current,,,) = poolManager.getSlot0(key.toId());
if (current == 0) {
    poolManager.initialize(key, sqrtPriceX96);
} else if (current ≠ sqrtPriceX96) {
    if (poolManager.getLiquidity(key.toId()) ≠ 0) revert PriceAlreadySet(current,
   sqrtPriceX96);
    poolManager.unlock(abi.encode(key, sqrtPriceX96));
    (uint160 moved,,,) = poolManager.getSlot0(key.toId());
    if (moved ≠ sqrtPriceX96) revert PriceAlreadySet(moved, sqrtPriceX96);
}
```

The attacker's band is out of range, so `getLiquidity` reads zero and the price walk lands on the target. The lock then mints the full range position, which references the pre-filled boundary tick, so `modifyLiquidity` reverts `TickLiquidityOverflow`. That revert is non-empty and is not `NotEnoughGas`, so the factory reads it as a funded squat and moves to the next spacing. All eight boundary ticks are pre-filled the same way, so every key reverts and the factory ends in `NoUsablePool`. The curve, called through the crossing buy, catches this and parks the launch at the bar with both buys and sells shut, and one permissionless `rescue` call then opens one-way refund mode.

This is distinct from a dust squat that parks capital at the current price. Nothing in scope reads `liquidityGross` at the boundary ticks, the guard in `seed` reads only active liquidity at the current tick, and the price walk never crosses the out-of-range band. Because the band earns nothing and sits at a price no swap can reach, the attacker keeps it forever at zero cost and burns it later to recover the dust.

IMPACT

Any Climbing launch can be forced off its graduation path for dust. Once the curve reaches the bar it parks with buys and sells reverting, and one permissionless `rescue` turns it into one-way refund mode. Refund pays at the live curve price with the fee waived, so whoever refunds first is made whole from the pot and the last buyer eats the loss. In the proof of concept the last buyer recovers about 13 percent of principal, and all fees and snipe tax already swept are gone. The attacker can be the party who refunds first, so this is also a profitable rug in the hands of a creator or an early holder. The band capital is fully recoverable, so the attacker risks only gas.

The condition is permanent. The band cannot be removed by anyone but the attacker and is not reached by any swap, so it survives across time. A retry-after-a-delay guard on `rescue` does not help, because the second attempt fails identically. A minimum-active-liquidity or budgeted-swap guard on `seed` does not help either, because the band is out of range and reads as zero active liquidity while the mint still overflows.

For a launch whose opening buy crosses the bar inside `launch`, the same pre-fill instead makes `launch` revert with `NoUsablePool`.

RECOMMENDATION

Reading active liquidity is not enough, because the attack lives on the boundary ticks the seed always references, not at the current tick. Before conceding a key, read the gross liquidity already sitting on both boundary ticks (`StateLibrary.getTickLiquidity` at `minUsableTick` and `maxUsableTick`) and treat `gross + seedLiquidity > tickSpacingToMaxLiquidityPerTick` as a squatted key rather than a hard failure, so the loop moves to the next spacing on a fillable key instead of ending in `NoUsablePool`. That alone does not remove the attack, because every candidate boundary tick can be filled. Combine it with a position range the attacker cannot enumerate and pre-fill cheaply: for example, choose the graduated pool's tick spacing or the position's exact boundary ticks from graduation-time state, over a space wide enough that filling every candidate costs more than the value at stake, rather than from a fixed list of eight spacings and the extreme usable ticks. Whatever range is chosen, keep the atomic drain-then-seed structure so a rejected mint continues to unwind cleanly.

H-05 HIGH ● FIXED

A crossing buy and rescue run inside a PoolManager unlock force a healthy launch into refund mode atomically

The curve decides whether a graduation attempt genuinely failed or merely ran out of gas by inspecting the revert data:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchCurve.sol#L388-L411

```
function _starved(bytes memory err) private pure returns (bool) {
    if (err.length == 0) return true;
    if (err.length < 4) return false;
    bytes4 sel;
    assembly { sel := mload(add(err, 32)) }
    return sel == ILaunchFactoryGraduation.NotEnoughGas.selector;
}

function _tryAutoGraduate() private {
    if (readyToGraduate()) {
        try ILaunchFactoryGraduation(factory).graduate(token) {}
        catch (bytes memory err) {
            if (_starved(err)) revert OutOfGasProbe();
            emit AutoGraduationFailed(token);
        }
    }
}
```

Only empty returndata and the factory's own `NotEnoughGas` are treated as "not a real answer, retry". Any other non-empty revert is treated as a genuine refusal, so the crossing buy succeeds and the launch parks at the bar. The seed path re-enters the Uniswap v4 PoolManager through `unlock`:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LpTopUpErc20.sol#L73-L79

```
function addLiquidity(uint256 liquidity, uint256 max0, uint256 max1) external onlyOwner {
    if (!keySet) revert KeyNotSet();
    IERC20 t0 = IERC20(Currency.unwrap(_key.currency0));
    IERC20 t1 = IERC20(Currency.unwrap(_key.currency1));
    if (max0 > 0) t0.safeTransferFrom(msg.sender, address(this), max0);
    if (max1 > 0) t1.safeTransferFrom(msg.sender, address(this), max1);
    poolManager.unlock(abi.encode(int256(liquidity), msg.sender));
```

The v4 PoolManager reverts the four-byte `AlreadyUnlocked` when `unlock` is called while it is already unlocked:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/lib/v4-core/src/PoolManager.sol#L103-L104

```
function unlock(bytes calldata data) external override returns (bytes memory result) {
    if (Lock.isUnlocked()) AlreadyUnlocked.selector.revertWith();
```

An attacker opens their own PoolManager unlock and, inside the callback, calls the curve's `buy` for the crossing slice. The buy triggers `_tryAutoGraduate`, whose seed calls `addLiquidity`, whose `unlock` reverts `AlreadyUnlocked`. That revert is non-empty and not `NotEnoughGas`, so every one of the eight seed attempts is read as a squatted key and the factory ends in `NoUsablePool`. The curve treats `NoUsablePool` as a genuine refusal, emits `AutoGraduationFailed`, and the buy returns, leaving a healthy launch parked at the bar. Still inside the same unlock

H-05 · CONTINUED

callback, the attacker calls `rescue`, whose graduation attempt fails the same way, so the catch branch opens one-way refund mode:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L461-L470

```
if (gasleft() < MIN_RESCUE_GAS) revert NotEnoughGas();
try this.selfGraduate(token) {
    return;
} catch (bytes memory err) {
    if (err.length == 0 || bytes4(err) == NotEnoughGas.selector) revert NotEnoughGas();
    registry.markRefunding(token);
    c.openRefunds();
}
```

The attacker then refunds first, all within the one transaction. No pool key is squatted, no balance is donated, and no waiting is required: a healthy launch is turned into refund mode atomically, on demand. The repository's own regression test asserts that a crossing buy followed by rescue simply graduates; this shows it does not.

IMPACT

Any launch at or near the bar can be rugged in a single transaction. The launch parks with buys and sells reverting, refunds open, and refund pays at the live curve price with the fee waived, so whoever refunds first is made whole from the pot and the last buyer eats the loss. In the proof of concept a later buyer recovers about 20 percent of principal while the attacker, who refunds first from inside the callback, recovers nearly all of the crossing buy. The attacker's cost is the trade fee on the crossing slice, which is small, plus gas. Because the attacker positions to refund first, this is a profitable rug in the hands of a creator or an early holder, not only griefing.

RECOMMENDATION

Do not let a graduation attempt that failed because the PoolManager was already unlocked be read as a genuine key refusal. At the top of the graduation body, reject the call when the PoolManager is already unlocked (read `Lock.isUnlocked` through `exttload` and revert `NotEnoughGas`), or in the seed loop treat `AlreadyUnlocked` the same way `NotEnoughGas` and empty returndata are treated, so `_starved` bubbles `OutOfGasProbe` and the crossing buy reverts rather than parking. With that in place, `rescue` called from inside an unlock reverts `NotEnoughGas` instead of opening refunds. Independently, make the step from a failed graduation to open refunds require more than one on-demand failure, for example a second failed attempt separated by a delay, so a single manufactured failure cannot end a launch.

MEDIUM SEVERITY · 5 FINDINGS

M-01 MEDIUM FIXED

`topUp` releases part of the new deposit immediately

`vestedOf` applies the elapsed fraction of the schedule to the whole `total`:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/CreatorVest.sol#L58-L64

```
function vestedOf(address token, address locker) public view returns (uint256) {
    Vest memory v = vests[token][locker];
    if (v.total == 0) return 0;
    uint256 elapsed = block.timestamp - v.start;
    if (elapsed >= v.duration) return v.total;
    return (uint256(v.total) * elapsed) / v.duration;
}
```

`topUp` only adds to `total` and keeps `start`:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/CreatorVest.sol#L49-L56

```
function topUp(address token, uint256 amount) external {
    Vest storage v = vests[token][msg.sender];
    if (v.total == 0) revert NoVest();
    if (amount == 0) revert BadTerms();
    if (!IERC20Minimal(token).transferFrom(msg.sender, address(this), amount)) revert
   TransferFailed();
    v.total += uint128(amount);
    emit ToppedUp(token, msg.sender, amount);
}
```

A top-up made half way through the schedule is therefore half claimable in the same block, and a top-up after the end date is fully claimable at once.

IMPACT

The contract exists as a public proof that a creator's tokens are locked. A creator can announce a top-up and take part of it straight back, so the on-chain badge overstates what is locked. Only the locker's own tokens move.

RECOMMENDATION

On `topUp`, pay out what is already claimable and restart the schedule from now to the same end date, over the unvested remainder plus the new amount. The old tokens keep their release rate and the new ones vest over the remaining time, which is what the natspec promises. `total` then means the amount still scheduled rather than the lifetime total, so adjust anything that reads `vestInfo` for the lifetime figure. A top-up after the end date has nothing to extend, so reject it (or start a fresh schedule if that is wanted).

M-02 MEDIUM ● FIXED

One failed `rescue` attempt opens refunds for good

`rescue` tries the graduation once and treats any non-empty revert as proof that no pool can ever be opened:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L459-L468

```
if (gasleft() < MIN_RESCUE_GAS) revert NotEnoughGas();
try this.selfGraduate(token) {
    return;
} catch (bytes memory err) {
    // empty returndata is out-of-gas or a failed CREATE; NotEnoughGas is
    // the seed loop saying the same about one of its eight attempts
    if (err.length == 0 || bytes4(err) == NotEnoughGas.selector) revert NotEnoughGas();
    registry.markRefunding(token);
    c.openRefunds();
}
```

`markRefunding` and `openRefunds` are one-way: `graduate` reverts `WrongPhase` for that launch forever, and `openRefunds` reverts on a second call. Nothing distinguishes a permanent, funded squat from a temporary condition: removable dust liquidity (the PoC), a quote token that temporarily blocks the seeder, or any other revert that would not recur on a later attempt. `rescue` is permissionless, so the caller needs no stake in the launch.

IMPACT

Anyone can turn a healthy, fully funded launch into refund mode during a condition that lasts one block. Holders then get a refund whose payout depends on who refunds first, instead of the market they paid for, and there is no way back. Because `seed` accepts dust liquidity as a funded squat, the condition is also close to free to create.

RECOMMENDATION

Require the failure to be observed twice with a delay in between. The first failed attempt records a timestamp and returns; refunds open only if a second attempt after `RESCUE_DELAY` fails as well. A successful attempt still graduates immediately. The launch sits at the bar during the delay, which is already its state between reaching the bar and the rescue.

M-03 MEDIUM ● FIXED

One frozen backing asset blocks redemption of the whole basket

`redeem` pays every backing asset in one loop and reverts as a whole if any transfer fails:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchToken.sol#L96-L106

```
function redeem(uint256 amount, address to) external nonReentrant returns (uint256[] memory
    paid) {
     if (!isBacked()) revert NotBacked();
     if (amount == 0) revert ZeroAmount();
     if (to == address(0)) revert ZeroAddress();
     paid = redeemableFor(amount);
     _burn(msg.sender, amount);
     for (uint256 i = 0; i < paid.length; i++) {
         if (paid[i] > 0) IERC20(_backingAssets[i]).safeTransfer(to, paid[i]);
     }
     emit Redeemed(msg.sender, to, amount);
}
```

Backing assets are chosen by the creator and can be any ERC-20, including tokenised stocks with compliance pauses or blacklists. If one asset refuses the transfer, the burn and the healthy legs revert too, and nothing lets a holder skip that leg. `BackedToken.redeem` has the same shape.

IMPACT

Holders cannot redeem anything while one asset is paused or blocks the recipient, even though the other legs could pay. `redeem` is the only exit for backing and there is no admin or rescue, so a permanent freeze (a delisted or migrated stock token) strands the entire basket.

RECOMMENDATION

Isolate the legs the way `LaunchFeeSplitter` isolates its recipients. Send each leg with a gas-capped low-level call; if it fails, record the amount as owed to the recipient and continue, so the burn and the other legs complete. Exclude the owed totals from `redeemableFor` so later redeemers are not paid out of parked amounts, and let the recipient pull its owed amount once the asset works again. Apply the same change to `BackedToken`.

M-04 MEDIUM ● FIXED

Refunding a Backed Launch Permanently Strands Backing Claims

A backed `LaunchToken` carries two independent claims: the quote exit provided by `LaunchCurve`, and the pro-rata basket claim provided by `LaunchToken.redeem()`. The terminal refund path settles only the former.

`LaunchCurve.refund()` transfers the holder's tokens into the Curve and updates its curve accounting, but it neither burns those tokens nor calls `LaunchToken.redeem()`:

```
uint256 outstanding = launchSupply - trackedTokens;
if (tokensIn ⩾ outstanding || quoteOut > trackedQuote) quoteOut = trackedQuote;
IERC20(token).safeTransferFrom(msg.sender, address(this), tokensIn);
trackedTokens += tokensIn;
trackedQuote -= quoteOut;
if (quoteOut ≠ 0) quote.safeTransfer(recipient, quoteOut);
```

Refunding is a terminal state. The Curve can no longer trade or graduate, and it has no function that can call `redeem()` or otherwise recover backing. Every refunded token therefore transfers its backing claim to a contract that can never exercise it.

This loss is unavoidable even if holders understand the problem and choose backing redemption instead of quote refunds. The Curve holds `reservedTokens` throughout the climbing phase. With the deployed `phantomQuote / graduationThreshold = 0.4` tiers, that reserve is approximately 28.6% of the initial supply. Once refunds open, those reserve tokens remain in the terminal Curve. If every circulating holder burns their tokens through `LaunchToken.redeem()`, `totalSupply == curve.balanceOf(token) == reservedTokens`, and all backing that remains belongs exclusively to the Curve.

Backing can be accumulated by the engine's backing route or sent directly to the token, so this does not depend on an empty or malformed basket. It is also distinct from issues #1 and #14, which concern quote-refund ordering and slippage; issue #5, which requires a backing token to fail; and issue #6, which concerns zero supply. Here ordinary ERC-20 backing becomes inaccessible even when every holder takes timely action.

Design context: an oversight, not intended behavior. Three pieces of in-repo evidence show the stranding was never a considered product rule:

1. The refund path is elaborately documented for the quote side: why refunds pay the curve price rather than a pooled average, how donations are handled, how the last holder is made whole. Yet the word "backing" appears nowhere in `LaunchCurve.sol`. The quote leg of the wind-down was designed; the backing leg was not.
2. The `refund()` natspec says "Burn tokens back to the curve", but the implementation transfers the tokens into the Curve. And even a natspec-accurate burn would not fix the reserve problem: PoC case 2 proves that once the circulating supply is gone, `totalSupply == reservedTokens` and the Curve's reserve still owns every remaining backing right. The reserve itself must be burned when refunds open (Recommendation 1).
3. The same codebase solves the isomorphic problem at graduation: `if (retire > 0) LaunchToken(token).burn(retire); // burned, not parked: never dilutes backing` (`LaunchFactory.sol:390`). Dead supply is deliberately prevented from holding backing claims in the graduation transition; the refund transition received no such treatment.

The stranded state also contradicts the token's own documented invariants, "nothing can take backing out except redemption" and "REDEEMABLE, PRO-RATA, FOREVER" (`BackedToken.sol:17-24`), and nothing in the refund flow discloses to a holder that accepting the quote forfeits the basket.

IMPACT

Once a backed launch enters `Refunding`, holders must choose between their quote claim and their backing claim for the same tokens. A holder who follows the advertised refund path receives quote but permanently transfers the backing claim

M-04 · CONTINUED

to the Curve. If all circulating tokens are refunded, the Curve holds the complete token supply and 100% of the backing is permanently inaccessible.

If holders instead redeem all circulating tokens for backing, the Curve's reserve allocation still owns approximately 28.6% of the initial backing rights under the deployed tier ratio. There is no sequence that lets holders recover both asset legs or fully exit the backing basket.

The stranded amount is unbounded, since backing keeps accumulating from every engine fee cycle, and the terminal state is attacker-reachable through forced-refund vectors such as issues #2 and #4. The strongest chain runs through issue #15: a single parked, fully recoverable deposit forces any healthy launch into `Refunding` with no squatting and no privileged access, and once the launch is there, this finding permanently destroys its entire accumulated backing. For a backing-heavy (XCORP-shaped) launch the backing is the product's value floor, so the #15 → N-7 chain is an existential risk to that launch, even though the attacker does not directly receive the stranded backing: the value is destroyed, not stolen.

RECOMMENDATION

Settle both claims as part of the refund lifecycle:

1. When refunds open, burn the Curve's unsold/reserved tokens so that backing rights remain only with the circulating holders.
2. In `refund()`, pull the holder's tokens and have the Curve call `LaunchToken.redeem(tokensIn, recipient)` so the tokens are burned and the backing basket is paid to the same recipient alongside quote.
3. Track quote refunds with a separate `refundedTokens` value instead of reusing `trackedTokens` as both reserve accounting and a physical token balance.
4. Isolate failures per backing leg so one frozen asset cannot roll back the quote refund and every healthy backing payout.

If the intended product rule is that accepting quote explicitly forfeits backing, the forfeited assets still need a reachable, disclosed beneficiary and a terminal settlement path. Leaving the claim owned by an inert Curve permanently destroys value.

M-05 MEDIUM ● FIXED

The `minLiquidity` Mitigation for LP Top-Up Sandwiching Is Dimensionally Incapable of Working

`LaunchFeeSplitter.topUp()` is the engine's LP leg. The distributor supplies both currencies and the splitter adds them as permanent full-range liquidity at the current spot price, which it reads directly from the pool:

```
(uint160 sqrtP,,,) = lock.poolManager().getSlot0(key.toId());
liquidity = LaunchMath.fullRangeLiquidity(sqrtP, key.tickSpacing, amt0, amt1);
...
if (liquidity < minLiquidity) revert TopUpSlippage(liquidity, minLiquidity);
```

`getSlot0` at line 120 is the only price input in the function, and the check at line 131 is its only slippage control. The natspec presents that parameter as the defence:

```
/// @param minLiquidity floor on the liquidity minted, computed off chain.
///        Without it the keeper's scheduled, predictable deposit can be
///        sandwiched, and because the position can never be withdrawn, the
///        mispricing is a permanent loss to the engine, creator and holders.
```

"Without it … can be sandwiched" states that the bound is what prevents the sandwich. It does not. `minLiquidity` bounds a **quantity**, while the manipulation it is meant to stop is a **price** move, and for a lopsided deposit the two are not aligned. Full-range liquidity is the minimum of the capacities supplied by the two legs:

```
L = min( amount_token * sqrtP ,  amount_quote / sqrtP )
```

If the deposit is token-heavy, the quote leg is the limiting one, so `L` is inversely proportional to `sqrtP`. An attacker who sells tokens pushes `sqrtP` **down**, which makes the fixed quote leg support **more** liquidity. The manipulated call therefore mints more than the honest call:

```
manipulatedLiquidity > honestLiquidity
```

This makes the bound unusable rather than merely loose:

- Any lower bound that permits execution at the honest price must be `⩽ honestLiquidity`.
- The attacker moves the price so that `manipulatedLiquidity > honestLiquidity`.
- The manipulated deposit therefore satisfies every bound that would have allowed the honest transaction.
- The attacker reverses the first trade against the newly enlarged, permanently locked position and keeps the difference.

No distributor compromise, no privileged access and no zero slippage parameter are required. The repository's existing `test_ATTACK_topUp_can_be_sandwiched_and_the_loss_is_permanent` demonstrates the sandwich only with `minLiquidity = 0` (`ReviewEcon.t.sol:769`), so the documented mitigation has never been exercised by the project's own suite. This finding shows that setting it correctly does not help.

Prior art inside this repository, and the delta of this finding:

The sandwich risk is not new to this codebase; what is new is the verdict on the documented defense. The timeline matters for deduplication:

1. **First internal pass (M2, fixed).** `topUp` underflowed whenever the position had accrued more fees than the top-up cost, the same delta bug `LpTopUpErc20.sol:29-38` documents as fixed in the lock. The splitter re-introduced it; `test/ReviewCorrect.t.sol:344` is the regression test for the harvest-first fix.
2. **Second internal pass (FINDING 4, `test/ReviewEcon.t.sol:683`).** The sandwich was demonstrated end-to-end,

M-05 · CONTINUED

1. but only with the control switched off: `topUp(a0, a1, 0)` at line 769. Reproducing it today logs a 3,000-quote round trip returning `295129793584625266047 7` without the top-up versus `307838007877987454014 5` with it, so `12708214293362187966 8` (≈127.08 quote) moved from the permanently locked position to the attacker. The finding's own header already names the missing dimension: "`topUp()` has **no price bound**".
2. **The shipped response.** The `minLiquidity` parameter, whose natspec presents it as the defense: "Without it … can be sandwiched" (`LaunchFeeSplitter.sol:107-111`). Because FINDING 4 exercises the defense only at `minLiquidity = 0` , the control itself has never been tested by the project's own suite. The PoC in this report is the first execution of it with a correctly calibrated bound.

This finding closes the loop the earlier passes left open. FINDING 4 shows the attack works with the control off; N-8 shows the control cannot work even when calibrated to zero tolerance, because the bound constrains a quantity while the attack moves a price, and for a lopsided deposit the manipulated quantity exceeds the honest one. The remedy is therefore not "document the parameter better" or "set it tighter" but to add the dimension FINDING 4 already named: an explicit price bound, plus a deadline; see Recommendation.

IMPACT

An unprivileged searcher can sandwich any public, lopsided LP top-up and transfer part of a third-party-funded deposit to itself. The severity rests on five properties that hold together, not on the size of a single event.

1. **The defeated control is the only one.** The function has exactly one price read and exactly one slippage check. There is no defence in depth to fall back on, and the remedy is not "set a better parameter" but "add a parameter dimension that does not exist": a price bound and a deadline. A mitigation that is the sole protection and cannot work leaves the path entirely unprotected.
2. **The loss is permanent by construction.** The position can never be withdrawn: `LpTopUpErc20.addLiquidity()` takes a `uint256` liquidity delta that is only ever positive via the splitter, and `collect()` performs a zero-delta modify. A mispriced deposit is therefore not a temporary impairment a keeper can unwind at leisure, but a completed transfer.
3. **It is permissionless and needs no compromise.** Mempool visibility of a scheduled deposit is sufficient. No distributor key, no privileged role.
4. **It scales with every top-up and repeats indefinitely.** The natspec itself calls the deposit "scheduled, predictable". Each top-up is a fresh opportunity against the same permanently locked position, so the cumulative loss over a launch's life is unbounded rather than a one-off.
5. **The capital at risk is not the attacker's.** The subsidy comes out of the engine, creator and holder fee stream. The attacker funds the first leg with quote it recovers in the second, so it is trading with the deposit it is attacking.

The precondition is the normal case rather than an edge case. Under one-sided flow the two currencies accumulate independently, and both the interface and the unused-amount refund logic explicitly support unequal proportions. The reference scenario uses `80,000,000 LaunchToken + 1,000 quote` ; at the honest price roughly `40.8 million LaunchToken` matches `1,000 quote` , so the token leg is 1.96 times the balanced amount, not an extreme or zero-sided input. Even with `minLiquidity` set to the exact pre-attack expectation, the attacker gains `4,597,856.879709588063113463 LaunchToken` and ends with no uncounted quote. At the pre-attack price of about `0.0000245 quote/token` that is about `112.65 quote` , or 3.8% of the top-up's roughly `2,960 quote` pre-attack value.

RECOMMENDATION

Replace the quantity bound with explicit price and time bounds chosen by the distributor. Accept `minSqrtPriceX96` , `maxSqrtPriceX96` and `deadline` , and check the current pool price against the caller's own transaction parameters immediately before adding liquidity:

```
(uint160 sqrtP,,,) = lock.poolManager().getSlot0(key.toId());
if (sqrtP < minSqrtPriceX96 || sqrtP > maxSqrtPriceX96) revert TopUpSlippage(...);
```

```
if (block.timestamp > deadline) revert TopUpExpired();
liquidity = LaunchMath.fullRangeLiquidity(sqrtP, key.tickSpacing, amt0, amt1);
if (liquidity < minLiquidity) revert TopUpSlippage(liquidity, minLiquidity);
```

The reference bounds must come from a price the keeper approved before submitting the transaction, optionally strengthened with a TWAP or another trusted quote, because `minLiquidity` cannot be derived from that price in a way that constrains the executed one. Keep `minLiquidity` as a secondary quantity check: it still catches a deposit whose legs were mis-measured or mis-sized, and it is cheap. It cannot serve as the primary price-slippage control.

Rebalancing the two input legs near the reference-price ratio reduces the subsidy available to an attacker and is worth doing, but it is not a substitute for the price bound: the same asymmetry reappears whenever the legs cannot be pre-balanced. Private transaction submission reduces exposure and should be used, but it cannot replace on-chain bounds, because the bound is the only thing that protects a deposit whenever a searcher does see it.

LOW SEVERITY · 12 FINDINGS

L-01 LOW FIXED

`burn` can take the supply to zero while backing is still held

`redeem` pays the burned share of the backing, so the last holder redeeming empties the basket together with the supply. `burn` ignores the backing:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchToken.sol#L88-L91

```
function burn(uint256 amount) external {
    if (amount == 0) revert ZeroAmount();
    _burn(msg.sender, amount);
}
```

If the holders of the last tokens burn instead of redeem, `totalSupply` is zero, `redeem` can never be called again and there is no sweep. `BackedToken.burn` has the same shape. Through the factory this state is not reachable today, because the locked pool position always holds tokens, but both contracts can be deployed directly and `burn` is a live path (the factory calls it at graduation and the engine's buyback leg burns).

IMPACT

Backing left in the contract is stuck forever, and anything sent in later is stuck too. Only the last holders can cause it, by choosing the wrong exit.

RECOMMENDATION

Refuse a burn that would remove the last of the supply while any backing asset balance is nonzero. The factory's retire burn and the buybacks never burn the whole supply, so they are unaffected. Apply the same check in `BackedToken` using `backing` .

```
+  error BackingWouldStrand();

   function burn(uint256 amount) external {
       if (amount == 0) revert ZeroAmount();
+      if (amount == totalSupply()) {
+          uint256[] memory held = redeemableFor(amount);   // the whole basket when amount is
       the full supply
+          for (uint256 i = 0; i < held.length; i++) if (held[i] != 0) revert
       BackingWouldStrand();
+      }
       _burn(msg.sender, amount);
   }
```

L-02 LOW ● FIXED

Rotating the creator recipient orphans its unpaid arrears

When a fee push to a recipient fails, `_pay` parks the amount under that address and retries it on the next sweep:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchCurve.sol#L302-L313

```
function _pay(address to, uint256 amt) private {
    uint256 prev = owed[to];
    uint256 due = amt + prev;
    if (due == 0) return;
    if (_tryPay(to, due)) {
        if (prev != 0) { owedTotal -= prev; owed[to] = 0; }
    } else {
        owed[to] = due;
        owedTotal = owedTotal - prev + due;
        emit FeesStuck(due);
    }
}
```

`_sweep` always pays `creatorRecipient`, the live address from the registry. `acceptCreatorTransfer` replaces that address without moving `owed[oldRecipient]`:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchRegistry.sol#L135-L145

```
function acceptCreatorTransfer(address token) external {
    PendingTransfer memory p = pendingCreatorTransfer[token];
    if (p.to == address(0)) revert NoPendingTransfer();
    if (msg.sender != p.to) revert NotCreatorRecipient();
    if (block.timestamp < p.readyAt) revert TransferNotReady();
    if (block.timestamp > p.readyAt + TRANSFER_WINDOW) revert TransferExpired();
    address from = _creatorRecipient[token];
    _creatorRecipient[token] = p.to;
    delete pendingCreatorTransfer[token];
    emit CreatorRecipientChanged(token, from, p.to);
}
```

After a rotation no code path reads `owed[oldRecipient]` again. `LaunchFeeSplitter` has the same structure with `owed[token][to]`.

IMPACT

Fees that bounced before a rotation (a blocked or frozen recipient) stay in the contract forever, even after the old address can receive again. The loss is limited to the creator's own arrears.

RECOMMENDATION

Let an address pull its own arrears, independent of the live recipient lookup, in both the curve and the splitter. The `to` argument lets a recipient that cannot receive the quote (the reason the push bounced) direct the amount elsewhere.

```
// LaunchCurve
function claimOwed(address to) external nonReentrant {
    uint256 due = owed[msg.sender];
    if (due == 0) revert ZeroAmount();
```

L-02 · CONTINUED

```
    owed[msg.sender] = 0;
    owedTotal -= due;
    quote.safeTransfer(to, due);
}

// LaunchFeeSplitter, per currency
function claimOwed(address t, address to) external {
    uint256 due = owed[t][msg.sender];
    if (due == 0) revert ZeroAmount();
    owed[t][msg.sender] = 0;
    owedTotal[t] -= due;
    IERC20(t).safeTransfer(to, due);
}
```

L-03 LOW ● FIXED

`CreatorVest` records the requested amount, not the received amount, in a shared balance

`lock` and `topUp` store the `amount` argument as `total` and only check the `transferFrom` return value:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/CreatorVest.sol#L42-L43

```
if (!IERC20Minimal(token).transferFrom(msg.sender, address(this), amount)) revert
   TransferFailed();
vests[token][msg.sender] = Vest(uint128(amount), 0, uint64(block.timestamp), duration);
```

All lockers of the same `token` share one balance. With a fee-on-transfer token the contract receives less than the sum of the recorded totals, and `claim` pays the full recorded amount to whoever claims first.

IMPACT

The first claimer is paid with the second locker's principal, and the second locker's `claim` reverts for good. Launch tokens have no transfer tax and the vest is opt-in per token, so this needs a creator to lock a taxed token. Low.

RECOMMENDATION

Record the balance change instead of the requested amount in `lock` and `topUp`.

```
   interface IERC20Minimal {
+      function balanceOf(address who) external view returns (uint256);
       function transfer(address to, uint256 amount) external returns (bool);
       function transferFrom(address from, address to, uint256 amount) external returns (bool);
   }

       function lock(address token, uint256 amount, uint64 duration) external {
           if (amount == 0 || duration < MIN_DURATION || duration > MAX_DURATION) revert
       BadTerms();
           Vest storage v = vests[token][msg.sender];
           if (v.total != 0) revert AlreadyVesting();
-          if (!IERC20Minimal(token).transferFrom(msg.sender, address(this), amount)) revert
       TransferFailed();
-          vests[token][msg.sender] = Vest(uint128(amount), 0, uint64(block.timestamp),
       duration);
-          emit Locked(token, msg.sender, amount, duration);
+          uint256 received = _pull(token, amount);
+          vests[token][msg.sender] = Vest(uint128(received), 0, uint64(block.timestamp),
       duration);
+          emit Locked(token, msg.sender, received, duration);
       }

       function topUp(address token, uint256 amount) external {
           Vest storage v = vests[token][msg.sender];
           if (v.total == 0) revert NoVest();
           if (amount == 0) revert BadTerms();
-          if (!IERC20Minimal(token).transferFrom(msg.sender, address(this), amount)) revert
       TransferFailed();
-          v.total += uint128(amount);
-          emit ToppedUp(token, msg.sender, amount);
+          uint256 received = _pull(token, amount);
```

L-03 · CONTINUED

```
+           v.total += uint128(received);
+           emit ToppedUp(token, msg.sender, received);
        }

+       function _pull(address token, uint256 amount) private returns (uint256 received) {
+           uint256 before = IERC20Minimal(token).balanceOf(address(this));
+           if (!IERC20Minimal(token).transferFrom(msg.sender, address(this), amount)) revert
        TransferFailed();
+           received = IERC20Minimal(token).balanceOf(address(this)) - before;
+           if (received == 0) revert BadTerms();
+       }
```

L-04 LOW ● FIXED

The seeding path and the opening buy are not fee-on-transfer safe

`buy` measures what it received:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchCurve.sol#L211-L213

```
uint256 before = quote.balanceOf(address(this));
quote.safeTransferFrom(msg.sender, address(this), quoteIn);
uint256 received = quote.balanceOf(address(this)) - before;   // fee-on-transfer safe
```

The graduation path does not. `seed` pulls `max0` and `max1` and re-requests the same amounts one hop further:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/PoolSeeder.sol#L106-L110

```
t0.safeTransferFrom(msg.sender, address(this), max0);
t1.safeTransferFrom(msg.sender, address(this), max1);
t0.forceApprove(address(lock), max0);
t1.forceApprove(address(lock), max1);
lock.addLiquidity(liquidity, max0, max1);
```

`LpTopUpErc20.addLiquidity` (L77-L79) and `LaunchFeeSplitter.topUp` (L121-L125) repeat the pattern, and the opening buy in `launch` approves and buys with the nominal `openingBuyQuote` (L301-L303). With a quote that takes a fee on transfer, the second hop asks for more than was received and reverts.

IMPACT

Every graduation attempt for a launch on such a quote reverts, so the launch can only end in refund mode, and a launch with an opening buy reverts outright. Quotes are registered by the trusted `setup` key, so this is a configuration hazard rather than an attack.

RECOMMENDATION

Treat fee-on-transfer and rebasing quotes as unsupported and never register them with `addQuotes`. If they must be supported, measure the received balance at every hop (`seed` , `addLiquidity` , `topUp`) and derive `liquidity` from the received amounts rather than the requested ones. The opening buy fix is small:

```
   if (p.openingBuyQuote > 0) {
+      uint256 before = IERC20(p.quote).balanceOf(address(this));
       IERC20(p.quote).safeTransferFrom(msg.sender, address(this), p.openingBuyQuote);
-      IERC20(p.quote).forceApprove(curve, p.openingBuyQuote);
-      c.buy(p.openingBuyQuote, p.openingBuyMinOut, msg.sender);
+      uint256 got = IERC20(p.quote).balanceOf(address(this)) - before;
+      IERC20(p.quote).forceApprove(curve, got);
+      c.buy(got, p.openingBuyMinOut, msg.sender);
```

L-05 LOW ● ACKNOWLEDGED

`protocol` and `distributor` recipients cannot be rotated

The creator leg can move to a new address through the 3-day two-step transfer in the registry (`proposeCreatorTransfer` , `acceptCreatorTransfer`), and the curve and the splitter read it live. The other two legs are immutables set once in the factory constructor and copied into every curve and splitter:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L105-L106

```
address public immutable distributor;   // engine wallet
address public immutable protocol;      // our cut lands here
```

No function in the factory, the curve or the splitter can change them. If a key is lost, or the address is blocked by a quote token, its share of every launch keeps going there or piles up as `owed` , which is never re-divided.

IMPACT

Protocol and engine revenue on every existing and future launch is lost until a new factory is deployed and launches move over. Trusted-role key hygiene.

RECOMMENDATION

Point both immutables at contracts the protocol controls, such as a multisig or a small forwarder whose destination its signers can change, so the launchpad itself needs no change. Alternatively give the two legs the same propose/accept flow the creator leg has, keyed to the current recipient and read live from the registry. The share terms stay immutable either way.

L-06 LOW ● FIXED

`LaunchToken.redeem` burns tokens for a zero payout

`BackedToken.redeem` refuses a redemption that pays nothing:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/BackedToken.sol#L154-L161

```
out = backingOf(shares); // against the float BEFORE the burn
uint256 n = out.length;
uint256 total;
for (uint256 i = 0; i < n; i++) {
    total += out[i];
}
if (total == 0) revert NothingToRedeem(); // dust would burn for nothing
_burn(msg.sender, shares); // reverts on insufficient balance
```

`LaunchToken.redeem` has no such check:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchToken.sol#L100-L104

```
paid = redeemableFor(amount);
_burn(msg.sender, amount);
for (uint256 i = 0; i < paid.length; i++) {
    if (paid[i] > 0) IERC20(_backingAssets[i]).safeTransfer(to, paid[i]);
}
```

`paid` is all zeros when the backing recipe is set but nothing has been deposited yet (a launch before its first fee cycle), or when `amount` is small enough to round every leg to zero.

IMPACT

The caller's tokens are burned and nothing is paid. A wallet or user that redeems too early or in too small an amount loses the tokens for good. Self-inflicted, Low.

RECOMMENDATION

Mirror the guard from `BackedToken`.

```
+  error NothingToRedeem();

   function redeem(uint256 amount, address to) external nonReentrant returns (uint256[] memory
       paid) {
        if (!isBacked()) revert NotBacked();
        if (amount == 0) revert ZeroAmount();
        if (to == address(0)) revert ZeroAddress();
        paid = redeemableFor(amount);
+       uint256 total;
+       for (uint256 i = 0; i < paid.length; i++) total += paid[i];
+       if (total == 0) revert NothingToRedeem();
        _burn(msg.sender, amount);
```

L-07 LOW ● ACKNOWLEDGED

The splitter pays out any balance it holds as fee income

`_payOut` divides the contract's live balance, minus the parked arrears, between the three recipients:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFeeSplitter.sol#L135-L148

```
function _payOut(address t) private {
    uint256 held = owedTotal[t];
    uint256 raw = IERC20(t).balanceOf(address(this));
    // only NEW income is divided; arrears belong to whoever they bounced from
    uint256 bal = raw > held ? raw - held : 0;
    if (bal == 0 && held == 0) return;
    uint256 toProtocol = (bal * protocolBps) / BPS_DENOM;
    uint256 toCreator = (bal * creatorBps) / BPS_DENOM;
    uint256 toHolders = bal - toProtocol - toCreator;   // remainder to holders
    _pay(t, protocol, toProtocol);
    _pay(t, creator(), toCreator);
    _pay(t, distributor, toHolders);
    if (bal ≠ 0) emit Split(t, toHolders, toProtocol, toCreator);
}
```

`collect` is permissionless. Any token that reaches the splitter outside `lock.collect`, for example a transfer sent to it by mistake, is paid out to the three recipients on the next `collect`.

IMPACT

A mis-sent token is not recoverable by the sender; it goes to the fee recipients.

RECOMMENDATION

Account income explicitly: add the amounts returned by `lock.collect` in `collect` and `topUp` to a pending balance per currency and divide only that, leaving anything else where it is. The graduation leftovers the factory forwards can be registered through a small notify call from the factory. If the current behaviour is intended, document that any token sent to a splitter is treated as fee income.

L-08 LOW ● FIXED

A squatted key reverts the creator's launch instead of leaving it Climbing

An opening buy that clears the bar graduates inside `launch`. The comment says a squatted key must not revert the launch, and `_usableKey` is checked first:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L306-L316

```
// An opening buy big enough to clear the bar graduates here and now,
// but only if it can: the token address is predictable, so a griefer
// could otherwise squat the keys and revert the creator's entire
// launch. If it cannot graduate, the launch simply stays Climbing and
// anyone can graduate it later.
if (c.readyToGraduate()) {
    LaunchRegistry.Launch memory l0 = registry.byToken(token);
    (uint160 sp0,,) = _graduationPrice(token, l0);
    (,, bool okKey) = _usableKey(token, l0.quote, l0.feeBps, sp0);
    if (okKey) _graduate(token);
}
```

`_usableKey` reads `getLiquidity`, which only reports liquidity at the current tick:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L416-L418

```
if (current == 0 || current == sqrtP || poolManager.getLiquidity(key.toId()) == 0) {
    return (key, tokenIs0, true);
}
```

`seed` fails on liquidity anywhere between the current price and the launch price, which the comment above the seed loop (L364-L369) says itself. `_graduate` is called directly, not through a try/catch, so when the pre-check passes and the seed fails, `NoUsablePool` reverts the whole launch. The token address is known before launch (the deployer's next nonce) and a position parked off-tick on the quote side needs only the quote token, so the squat can be set up before the token exists, for dust.

IMPACT

A griefer can revert a creator's launch transaction, contrary to the design intent. The creator can retry without a bar-clearing opening buy and no funds are lost. The same squat then denies graduation later, since `seed` treats any liquidity as a funded squat.

RECOMMENDATION

Attempt the inline graduation through `selfGraduate` in a try/catch, as `rescue` does. `selfGraduate` is not reentrancy-guarded, so it can run under the guard of `launch`, and a failed attempt is rolled back and leaves the launch Climbing. `_usableKey` and `_graduationPrice` can then be removed.

```
    if (c.readyToGraduate()) {
-       LaunchRegistry.Launch memory l0 = registry.byToken(token);
-       (uint160 sp0,,) = _graduationPrice(token, l0);
-       (,, bool okKey) = _usableKey(token, l0.quote, l0.feeBps, sp0);
-       if (okKey) _graduate(token);
+       // a failed attempt is rolled back and the launch simply stays Climbing
+       try this.selfGraduate(token) {} catch {}
    }
```

L-09 LOW ● FIXED

`refund` has no `minQuoteOut`

`buy` and `sell` take a minimum output and `topUp` takes `minLiquidity`. `refund` takes none:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchCurve.sol#L345

```
function refund(uint256 tokensIn, address recipient) external nonReentrant returns (uint256
    quoteOut) {
```

Because `refund` prices each call against the live reserves, the payout moves with every earlier refund, and the amount a holder is quoted before sending can differ a lot from what the transaction pays.

IMPACT

A holder's refund can be front-run and pay a fraction of the quoted amount with no revert.

RECOMMENDATION

Add a `minQuoteOut` parameter and revert with `SlippageExceeded` when the payout is below it. If refunds are later changed to a single rate fixed when they open, the parameter is still cheap insurance.

```
- function refund(uint256 tokensIn, address recipient) external nonReentrant returns (uint256
    quoteOut) {
+ function refund(uint256 tokensIn, uint256 minQuoteOut, address recipient) external
    nonReentrant returns (uint256 quoteOut) {
      ...
      if (tokensIn ⩾ outstanding || quoteOut > trackedQuote) quoteOut = trackedQuote;
+     if (quoteOut < minQuoteOut) revert SlippageExceeded(quoteOut, minQuoteOut);
```

L-10 LOW ● FIXED

A lock created through the permissionless PoolSeeder can have its permanent principal withdrawn by its owner

`LpTopUpErc20` documents its principal as permanent: liquidity only ever goes in, and the absence of any exit path for principal is stated as the whole promise of the contract. `addLiquidity` takes the liquidity amount as a `uint256` and encodes it as an `int256` with a same-width cast:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LpTopUpErc20.sol#L73-L79

```
function addLiquidity(uint256 liquidity, uint256 max0, uint256 max1) external onlyOwner {
    if (!keySet) revert KeyNotSet();
    IERC20 t0 = IERC20(Currency.unwrap(_key.currency0));
    IERC20 t1 = IERC20(Currency.unwrap(_key.currency1));
    if (max0 > 0) t0.safeTransferFrom(msg.sender, address(this), max0);
    if (max1 > 0) t1.safeTransferFrom(msg.sender, address(this), max1);
    poolManager.unlock(abi.encode(int256(liquidity), msg.sender));
```

A same-width `int256(liquidity)` cast does not revert in Solidity 0.8, so a value in the upper half of the `uint256` range becomes a negative `int256`. The callback passes it straight to `modifyLiquidity` as `liquidityDelta`, which v4 only requires to fit `int128`, and a negative delta removes liquidity. The positive balance delta that results is then paid out to `to`, which is the caller:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LpTopUpErc20.sol#L94-L124

```
function unlockCallback(bytes calldata data) external returns (bytes memory) {
    if (msg.sender != address(poolManager)) revert OnlyPoolManager();
    (int256 liquidity, address to) = abi.decode(data, (int256, address));
    (BalanceDelta delta,) = poolManager.modifyLiquidity(
        _key,
        IPoolManager.ModifyLiquidityParams({
            tickLower: TickMath.minUsableTick(_key.tickSpacing),
            tickUpper: TickMath.maxUsableTick(_key.tickSpacing),
            liquidityDelta: liquidity,
            salt: 0
        }),
        ""
    );
    uint256 amt0 = _resolve(_key.currency0, delta.amount0(), to);
    uint256 amt1 = _resolve(_key.currency1, delta.amount1(), to);
    return abi.encode(amt0, amt1);
}
```

So the owner of a lock can call `addLiquidity` with `2**256 - L` and remove `L` units of the position, receiving the underlying principal.

Every lock created by the factory is safe from this. The factory hands the lock to a `LaunchFeeSplitter`, whose only `addLiquidity` caller is its `topUp`, which always passes a `uint128` computed from `fullRangeLiquidity`, and the splitter has no ownership-transfer path. But `PoolSeeder.seed` is permissionless and hands each fresh lock to whatever `owner` the caller names:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/PoolSeeder.sol#L101-L111

```
lock = new LpTopUpErc20(poolManager, address(this));
```

L-10 · CONTINUED

```
lock.setKey(key);

IERC20 t0 = IERC20(Currency.unwrap(key.currency0));
IERC20 t1 = IERC20(Currency.unwrap(key.currency1));
t0.safeTransferFrom(msg.sender, address(this), max0);
t1.safeTransferFrom(msg.sender, address(this), max1);
t0.forceApprove(address(lock), max0);
t1.forceApprove(address(lock), max1);
lock.addLiquidity(liquidity, max0, max1);
lock.transferOwnership(owner);
```

`PoolSeeder` is a deployed, in-scope contract whose own comments invite anyone to seed any ERC20 pair pool through it. Any lock created that way, with an externally owned account, a multisig, or an arbitrary contract as owner, therefore does not have the permanence its header promises, and the same is true for a factory lock if the splitter's ownership were ever movable in future.

IMPACT

For launches that graduate through the factory, nothing is lost: the lock owner is the splitter, which never passes a negative value and cannot be re-owned. The impact is confined to locks created through the permissionless `PoolSeeder` with an owner that can pass an arbitrary value. For those, the owner, or whoever compromises the owner key, can withdraw the entire principal and accrued fees of a position that is documented as permanently locked, so any third party who relied on that permanence loses the market they were promised. The gap is a latent deviation from a stated safety invariant rather than a live loss on the factory path, which is why this is rated Low.

RECOMMENDATION

Constrain the liquidity argument so it can only ever add. Type the parameter as `uint128` (the width `fullRangeLiquidity` already returns and the width `modifyLiquidity` accepts for a positive delta) and encode it as a strictly non-negative `int256`, or check `liquidity ≤ uint256(uint128(type(int128).max))` before the cast and pass a positive delta explicitly. The dedicated `collect` path already uses a zero delta for fee harvesting, so `addLiquidity` never needs to express a removal, and rejecting values outside the positive `int128` range preserves the documented no-exit-for-principal promise for every lock, including those created through the permissionless seeder.

L-11 LOW ● FIXED

A Non-ERC-20 Backing Leg Bricks Every Backing View and Redemption

`LaunchFactory._checkRecipe()` validates only that backing-asset addresses are nonzero and distinct, that the array length is bounded, and that weights sum to 10,000. It does not require an asset address to contain code or support the ERC-20 interface. `LaunchToken` then stores those addresses without further validation.

Both backing views call `balanceOf()` on every configured asset:

```
held[i] = IERC20(_backingAssets[i]).balanceOf(address(this));
```

```
out[i] = IERC20(_backingAssets[i]).balanceOf(address(this)) * amount / supply;
```

A high-level `balanceOf()` call to an EOA returns empty data and fails ABI decoding. A contract without a compatible `balanceOf()` either reverts or returns malformed data. Because `redeem()` begins by calling `redeemableFor()`, one invalid leg permanently reverts `backing()`, `redeemableFor()`, and every holder's redemption. Healthy backing already held in the other legs becomes inaccessible, the recipe is immutable, and the launch has no recovery path.

This is distinct from issue #5. Issue #5 requires a valid backing token to fail at runtime and may recover if that token is unfrozen. This defect accepts a deterministically invalid recipe at launch time and can make redemption unusable from the first block. The standalone `BackedToken` constructor has the same missing code/interface validation.

IMPACT

A creator mistake or malicious recipe can create a token that is presented on chain as backed while its entire backing basket is permanently non-redeemable. Any healthy assets later sent to the token are trapped together with the invalid leg. The creator selects the recipe and users can inspect it, so the issue is low severity, but the factory currently provides no deployment-time protection against an irreversible configuration.

RECOMMENDATION

Reject `asset.code.length == 0` in `LaunchFactory._checkRecipe()` and repeat the check in the token constructors as defense in depth. Code length alone cannot prove ERC-20 compatibility, so the launch flow should also perform a bounded `balanceOf(address)` probe and require exactly 32 bytes of valid return data.

For runtime resilience, isolate redemption legs instead of allowing one failing asset to roll back every healthy payout. A pull-based per-asset claim or an owed-balance mechanism can preserve claims until a temporarily failing asset becomes transferable again.

L-12 LOW ● FIXED

Pre-Refund Burns Leave Curve Quote Permanently Unreachable

`LaunchCurve.refund()` decides whether the caller is returning the last circulating tokens by deriving `outstanding` from the immutable original supply and the Curve's internal token reserve:

```
uint256 outstanding = launchSupply - trackedTokens;
if (tokensIn ⩾ outstanding || quoteOut > trackedQuote) quoteOut = trackedQuote;
IERC20(token).safeTransferFrom(msg.sender, address(this), tokensIn);
trackedTokens += tokensIn;
trackedQuote -= quoteOut;
```

That accounting observes buys, sells, and prior refunds, but it does not observe `LaunchToken.burn()`. If any circulating tokens are burned while the launch is climbing, `outstanding` permanently includes tokens that no longer exist. No future caller can return them, so the last-holder branch can never activate. Even after every live holder has returned every live token, part of `trackedQuote` remains in the terminal Curve with no callable recovery path.

Burning during the climbing phase is not merely an accidental edge case. The launch routing explicitly includes a buyback leg whose documented operation is to buy the launch token and burn it. Fees can be swept before graduation, so the engine can execute that route while the Curve is still climbing. If the launch later enters `Refunding`, the intended buyback creates the accounting gap.

This is distinct from issue #6. That issue concerns backing left after the entire token supply is burned and is not reachable for an official factory launch. Here any partial burn is sufficient, the affected asset is the Curve's quote reserve, and the factory's documented buyback route provides an official path to the state.

IMPACT

Holders cannot recover the full quote refund pool after any pre-refund buyback or other burn. The stranded amount grows with the missing token amount and remains locked indefinitely even when every existing holder acts promptly. Forced-refund vectors such as issue #15 make the terminal state attacker-reachable, but no attacker cooperation is required for the accounting mismatch itself.

An attacker can also create the same condition by burning their own purchased tokens before forcing a refund, although that variant sacrifices the burned tokens and is primarily griefing. The routine buyback path is the stronger impact because it converts value spent for all holders into a permanent refund shortfall.

RECOMMENDATION

Snapshot refundable supply from live token state when refunds open instead of reconstructing it from the original launch supply. For example, track the actual amount held outside the Curve using `totalSupply()` and the Curve's physical token balance, with explicit handling for tokens held by the Curve but not included in `trackedTokens`.

More robustly, burn the Curve's unsold reserve when refunds open and maintain dedicated `refundSupply` and `refundQuote` snapshots. Each returned token should be burned, and its quote payout should be calculated against those snapshots. This also composes with the backing settlement recommended for N7 and removes the overloaded use of `trackedTokens` as both AMM reserve accounting and refund completion accounting.

INFORMATIONAL SEVERITY · 8 FINDINGS

I-01 INFORMATIONAL FIXED

`LaunchCurve` and `LaunchFactory` enforce different ceilings for the same `feeBps`

The trade fee of a launch is one value, `feeBps`, chosen by the creator and used by the curve and by the graduated pool alike. Two contracts bound it, with two different ceilings.

The factory allows 1% to 5% and rejects anything else in `deriveSplit`:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L90-L91

```
uint16 public constant FEE_MIN_BPS = 100;
uint16 public constant FEE_MAX_BPS = 500;
```

The curve allows up to 10% and has no floor:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchCurve.sol#L46

```
uint256 public constant MAX_FEE_BPS = 1_000;
```

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchCurve.sol#L132

```
if (t.feeBps > MAX_FEE_BPS) revert FeeTooHigh();
```

A curve is only ever created by `LaunchDeployer.curve`, which is restricted to the factory, and `launch` runs `deriveSplit` before it deploys. The factory bound is therefore the only one that binds, and `FeeTooHigh` can never be raised. The curve's public `MAX_FEE_BPS` still advertises a 10% ceiling that the system does not honour.

The header comment of `LaunchCurve` adds a third number, a published 0.5% fee, which `FEE_MIN_BPS` makes impossible:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchCurve.sol#L24

```
///  1. Our fee structure. One trade fee (published 0.5%) charged on the quote
```

IMPACT

No on-chain impact today, a fee above 5% cannot reach a registered curve. The risk is in maintenance and integration. An integrator or UI reading `MAX_FEE_BPS` from the curve gets the wrong ceiling. A future factory that raises its own bound inherits a 10% curve limit that nobody reviewed against the rest of the fee math, for example the `launchLine` formula in `deriveSplit`, which was only reasoned about for 1% to 5%.

RECOMMENDATION

Keep one source of truth. Set `MAX_FEE_BPS` in `LaunchCurve` to `500` and add the matching `100` floor, so the curve enforces the same range as the factory. Correct the 0.5% figure in the `LaunchCurve` header comment to the real 1% to 5% range.

I-02 INFORMATIONAL ● FIXED

`_checkRecipe` reverts with the same `BadRecipe` error for every rule

`_checkRecipe` enforces seven rules and reverts with `BadRecipe` for all of them: recipe missing, too many assets, `assets` and `weights` length mismatch, zero asset, zero weight, duplicate asset, weights not summing to `BPS`.

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchFactory.sol#L324-L335

```
function _checkRecipe(LaunchRegistry.Recipe calldata r, bool required) private pure {
    uint256 n = r.assets.length;
    if (n == 0) { if (required) revert BadRecipe(); return; }
    if (n > MAX_RECIPE_ASSETS || r.weights.length ≠ n) revert BadRecipe();
    uint256 sum;
    for (uint256 i = 0; i < n; i++) {
        if (r.assets[i] == address(0) || r.weights[i] == 0) revert BadRecipe();
        for (uint256 j = 0; j < i; j++) if (r.assets[j] == r.assets[i]) revert BadRecipe();
        sum += r.weights[i];
    }
    if (sum ≠ BPS) revert BadRecipe();
}
```

`launch` runs this check on both `p.rewards` and `p.backing`, so one error covers every possible fault in two recipes.

IMPACT

No funds at risk. A creator whose `launch` reverts only sees `BadRecipe` and has to find the wrong field by trial and error. A UI cannot show a precise message either.

RECOMMENDATION

Use one error per rule. The checks and their order stay the same, only the revert data changes.

```
-  error BadRecipe();
+  error RecipeRequired();
+  error RecipeTooLong();
+  error RecipeLengthMismatch();
+  error RecipeZeroAsset();
+  error RecipeZeroWeight();
+  error RecipeDuplicateAsset();
+  error RecipeBadWeightSum();

       uint256 n = r.assets.length;
-      if (n == 0) { if (required) revert BadRecipe(); return; }
-      if (n > MAX_RECIPE_ASSETS || r.weights.length ≠ n) revert BadRecipe();
+      if (n == 0) { if (required) revert RecipeRequired(); return; }
+      if (n > MAX_RECIPE_ASSETS) revert RecipeTooLong();
+      if (r.weights.length ≠ n) revert RecipeLengthMismatch();
       uint256 sum;
       for (uint256 i = 0; i < n; i++) {
-          if (r.assets[i] == address(0) || r.weights[i] == 0) revert BadRecipe();
-          for (uint256 j = 0; j < i; j++) if (r.assets[j] == r.assets[i]) revert BadRecipe();
+          if (r.assets[i] == address(0)) revert RecipeZeroAsset();
+          if (r.weights[i] == 0) revert RecipeZeroWeight();
```

I-02 · CONTINUED

```
+           for (uint256 j = 0; j < i; j++) if (r.assets[j] == r.assets[i]) revert
      RecipeDuplicateAsset();
            sum += r.weights[i];
        }
-       if (sum != BPS) revert BadRecipe();
+       if (sum != BPS) revert RecipeBadWeightSum();
```

No contract catches `BadRecipe`, so no on-chain flow changes. The three `BadRecipe` assertions in `test/LaunchFactory.t.sol` at L197-L201 must switch to `RecipeRequired`, `RecipeBadWeightSum` and `RecipeDuplicateAsset`, and any off-chain decoder needs the new selectors. `LaunchFactory` grows by 70 bytes to 15,825, still 8,751 bytes under the 24,576 limit.

I-03 INFORMATIONAL ● FIXED

`CreatorVest.lock` is not tied to a launch or to its creator

`CreatorVest` presents itself as the creator badge for a launch, readable by anyone:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/CreatorVest.sol#L9-L14

```
/// @title CreatorVest
/// @notice Opt-in linear vesting for launch creators. No owner, no admin, no
///         pause, no early exit: once locked, tokens release linearly and only
///         to the locker. One vest per (token, locker); topUp extends the
///         amount but never shortens time. Anyone can read a creator's lock,
///         which is the whole point: the badge is verifiable on chain.
```

`lock` checks the amount and the duration, and nothing else:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/CreatorVest.sol#L38-L45

```
function lock(address token, uint256 amount, uint64 duration) external {
    if (amount == 0 || duration < MIN_DURATION || duration > MAX_DURATION) revert BadTerms();
    Vest storage v = vests[token][msg.sender];
    if (v.total != 0) revert AlreadyVesting();
    if (!IERC20Minimal(token).transferFrom(msg.sender, address(this), amount)) revert
   TransferFailed();
    vests[token][msg.sender] = Vest(uint128(amount), 0, uint64(block.timestamp), duration);
    emit Locked(token, msg.sender, amount, duration);
}
```

The contract holds no reference to the registry and never asks whether `token` is a launch or whether `msg.sender` is its creator. `token` can be any address with a `transferFrom` that returns true, and the locker can be anyone. A vest opened on a launch token by a stranger is the same record, and the same `Locked` event, as one opened by that launch's creator.

IMPACT

No funds are at risk. Each `(token, locker)` pair has its own slot and `claim` only ever pays the locker, so nobody can take another locker's principal or occupy another locker's slot.

What it costs is the badge and the scope. The record is not self-proving: a reader who wants to know the creator locked has to fetch the launch's creator from the registry and compare addresses, because nothing in the vest says it. An indexer listing the vests on a token shows a stranger's lock beside the creator's with nothing to separate them.

The scope cost is that the contract has to hold up for every ERC-20 in existence rather than for the one it was written for. All lockers of a token draw on one shared balance, so a token whose balances move on their own, by rebase or by a transfer fee, desyncs the recorded totals from what the contract actually holds. `uint128(amount)` is an explicit downcast, which 0.8 truncates silently instead of reverting, so a token with a supply above `2**128 - 1` can record less than it transferred. A `LaunchToken` has none of these traits: a plain OpenZeppelin ERC-20 with no tax, no hook and no rebase, minted once at the factory's fixed `economics.supply`, 1e27 on the live pad and about eleven orders of magnitude below that cast.

RECOMMENDATION

Check the registry in `lock`: require `token` to be a registered launch and `msg.sender` to be its current creator, and keep the per-`(token, locker)` accounting exactly as it is. That restricts the contract to the use it was written for and, as a side effect, bounds the token set to `LaunchToken`, whose transfer semantics are known.

I-03 · CONTINUED

The current creator is `creatorRecipient`, not the launch's `creator` field, which the registry keeps as history:

https://github.com/SB-Security/audit-2026-09-Bucket-Launchpad/blob/7c03df6d8714e1596befd61abeee780fb5b3ac6a/src/LaunchRegistry.sol#L113-L118

```
/// @notice The address paid the creator leg right now. The launch's
///         `creator` field is history; this is where the money goes.
function creatorRecipient(address token) external view returns (address) {
    if (_indexOfToken[token] == 0) revert NotALaunch();
    return _creatorRecipient[token];
}
```

```
  interface IERC20Minimal {
      function transfer(address to, uint256 amount) external returns (bool);
      function transferFrom(address from, address to, uint256 amount) external returns (bool);
  }

+ interface ILaunchRegistry {
+     function isLaunch(address token) external view returns (bool);
+     function creatorRecipient(address token) external view returns (address);
+ }
+
  contract CreatorVest {
```

```
  error TransferFailed();
+ error ZeroAddress();
+ error NotALaunchToken();
+ error NotCreator();
+
+ address public immutable registry;
+
+ constructor(address registry_) {
+     if (registry_ == address(0)) revert ZeroAddress();
+     registry = registry_;
+ }

  function lock(address token, uint256 amount, uint64 duration) external {
      if (amount == 0 || duration < MIN_DURATION || duration > MAX_DURATION) revert
     BadTerms();
+     if (!ILaunchRegistry(registry).isLaunch(token)) revert NotALaunchToken();
+     if (msg.sender != ILaunchRegistry(registry).creatorRecipient(token)) revert
     NotCreator();
      Vest storage v = vests[token][msg.sender];
      if (v.total != 0) revert AlreadyVesting();
```

Only `lock` needs the gate. `topUp` and `claim` both require an existing vest, and a vest can only exist if `lock` passed, so they stay as they are. `creatorRecipient` already reverts `NotALaunch` for an unregistered token, so the `isLaunch` line buys a clearer error rather than safety and can be dropped if a single call is preferred.

Nothing that works today stops working for a creator. After a recipient rotation the old address keeps its vest and can still top up and claim, and the new recipient can open its own, because a vest belongs to its locker and `claim` only pays the locker. The one thing the gate gives up is a holder who is not the creator locking a launch token as a public commitment; if that is wanted later it belongs in its own contract, since what makes the badge mean anything is that the locker is the creator.

A gated copy compiled against `LaunchRegistry` behaves as intended: the creator's lock on a launch token passes, an outside token reverts `NotALaunchToken`, a non-creator on a launch token reverts `NotCreator`, and after a recipient rotation only the new recipient can open a vest.

`CreatorVest` has no constructor today and takes the registry as an immutable, so it has to be deployed after the registry. The deployed instance is immutable, and the contract is standalone, so this ships with the next factory rather than as a patch to the live one.

I-04 INFORMATIONAL ● ACKNOWLEDGED

Allowed Quote Assets Retain Issuer Freeze, Block, Burn, and Upgrade Trust

Every launch inherits the control and liveness properties of its selected quote token. Core paths use ordinary ERC-20 transfers: buyers transfer quote into `LaunchCurve`, sellers and refunders receive quote from it, graduation transfers quote through the Factory and Seeder into Uniswap v4, and the graduated pool continues to depend on the same asset.

A read-only Robinhood Chain review on September 17, 2026 confirmed that this is a material trust assumption rather than a hypothetical token-standard concern. The allowed USDG address is an upgradeable proxy whose implementation exposes upgrade, facet-management, pause-related, mint, burn, and role-management functionality. The reviewed stock token is a beacon proxy; its control plane exposes upgrades and pausing, while the implementation exposes address blocking and administrative burning. These issuer controls can change the behavior or balance of quote assets already held by immutable launch contracts.

The protocol correctly isolates failed fee-recipient transfers through the `owed` ledgers. That mitigation does not cover a global pause, a block applied to the Curve, Factory, Seeder, or PoolManager, or an administrative burn of a Curve balance. The principal transfer itself then fails or the Curve becomes undercollateralized. A climbing launch below the graduation threshold cannot use `rescue()`, and a refunding launch cannot pay refunds while its quote transfer is disabled.

IMPACT

An issuer pause or address block can freeze buys, sells, graduation, and refunds for every launch using that quote. If the restriction is permanent, user principal is permanently inaccessible. Administrative burning can make `trackedQuote` exceed the Curve's physical quote balance, leaving it insolvent even if transfers are later re-enabled.

This is an informational trust-model disclosure rather than a permissionless vulnerability in Bucket's contracts. The quote issuer already has equivalent power over the underlying asset, and the current allowed tokens were deliberately selected. The important residual risk is that the launchpad is immutable and has no alternate settlement asset or migration path.

RECOMMENDATION

Document, per allowed quote, its proxy type, upgrade authority, pause authority, address-blocking behavior, and administrative balance controls. The UI should state that a launch freezes whenever its quote asset freezes and that Bucket cannot override the issuer.

Continuously monitor proxy implementations, beacon upgrades, role changes, pauses, blocks affecting launch contracts, and administrative burns. New quote assets should pass a formal control-surface review before being sealed into a future Factory. An alternate settlement or migration mechanism would require a new, carefully governed protocol version; adding an unrestricted owner to the current immutable design would introduce a larger trust boundary.

I-05 INFORMATIONAL ● FIXED

The Fixed Payout Gas Cap Can Make Arrears Permanently Unpayable

`LaunchCurve._tryPay()` and `LaunchFeeSplitter._tryPay()` cap every fee transfer at 120,000 gas:

```
(bool ok, bytes memory ret) = address(quote).call{gas: 120_000}(
    abi.encodeWithSelector(IERC20.transfer.selector, to, amt)
);
```

When the call fails, the amount is correctly recorded in `owed`. However, every retry uses the identical 120,000-gas call, and neither contract exposes a pull-based claim path. If an allowed token's valid `transfer()` path permanently grows beyond the cap, every fee payment for that token permanently fails even though the transfer would succeed with normal gas forwarding.

This can arise after an implementation or beacon upgrade to a regulated quote token that adds compliance, accounting, or callback work. It is different from a temporarily blocked recipient: unblocking resolves that case, whereas a permanently more expensive transfer can never pass the hard-coded retry path.

IMPACT

Creator, protocol, and distributor fee claims can accumulate indefinitely in the Curve or Splitter with no method that can pay them. Principal exits are intentionally protected because failed fee pushes do not revert the surrounding lifecycle operation, so the direct loss is limited to accrued fee income.

The currently reviewed quote implementations complete transfers below the cap, and an issuer capable of upgrading them already has stronger freeze powers. The issue is therefore informational, but it is a concrete liveness limit in the arrears design.

RECOMMENDATION

Add a pull-based `claimOwed(token, recipient)` path that clears the claim using checks-effects-interactions and forwards sufficient gas because the claimant pays for the transaction. Keep a bounded push path if it is needed to protect graduation and refund liveness, but do not make that same cap the only way to settle arrears.

The pull path should validate conventional ERC-20 return data, restore the claim if payment fails, and remain callable after graduation or refund mode. This also provides the natural place to address old-recipient arrears described in issue #7.

I-06 INFORMATIONAL ● FIXED

A Fully Claimed CreatorVest Can Never Be Re-Locked

`CreatorVest` stores exactly one `Vest` record per `(token, locker)` pair and keys the "already vesting" check on the record's `total` field:

```
Vest storage v = vests[token][msg.sender];
if (v.total ≠ 0) revert AlreadyVesting();
```

No code path ever clears the record. Once a vest has fully matured and been fully claimed, with the contract's balance of that token back to zero, a later `lock()` by the same locker for the same token still reverts `AlreadyVesting`, permanently. `topUp()` remains the only way to add funds, and a top-up on the expired schedule carries the old, elapsed timeline, so it is 100% claimable at once.

This is distinct from the already-known top-up schedule behavior tracked by issue #3 / fix M-01. That behavior concerns *when* topped-up funds can be claimed; this finding concerns the *persistence of the vest entry itself*, meaning what the locker can ever do afterwards. The known behavior says nothing about re-locking, and the M-01 reschedule does not clear the record either.

IMPACT

No third-party funds move and nothing is stolen; the impact is a permanent lifecycle restriction plus a commitment signal that survives its own meaning. The contract's stated purpose is a public, verifiable lockup badge. A creator who completed one honest vest can never present a fresh lock for the same token from the same address, so the most natural "I am locking again" gesture is impossible. `vestInfo` keeps returning the old `total` with `claimed == total` while the contract holds nothing, and any continued commitment must go through `topUp` on an expired schedule, which is the state where the badge stops describing a lockup at all.

RECOMMENDATION

Clear the record when it is drained: in `claim()`, when `claimed == total`, delete `vests[token][locker]`, so a later `lock()` starts a genuinely new schedule. If the one-shot property is deliberate, document it at the badge (`vestInfo` already exposes `claimed`, so readers can at least see the vest is spent) and revert `topUp` once the schedule has ended, so the two halves of the design agree.

I-07 INFORMATIONAL ● FIXED

A Launch Can List Its Own Token as Its Backing Asset

`BackedToken` 's constructor rejects a self-referential basket (`if (a == address(this)) revert SelfAsset()`), but `LaunchToken` 's constructor only rejects the zero address, and the factory-side recipe validation (`LaunchFactory._checkRecipe`) checks only for zero addresses, duplicates, zero weights, and the 10,000 weight sum. Neither excludes the launch's own token.

The token address is predictable at `launch()` time: the curve and the token are deployed in that order by the factory's immutable `LaunchDeployer` via plain `CREATE` , so the token address is `computeCreateAddress(deployer, nonce(deployer) + 1)` before the call executes. A creator can precompute it off-chain and pass it in `p.backing.assets` . Verified end to end: the launched token reports `isBacked() == true` with its own address as the sole backing asset, and the registry records the self-referential recipe.

The planned code-length check for issue #26 (fix L-12) would not close this: the self asset is a fully deployed ERC-20 with code, so the launch passes that check as well.

IMPACT

No one can extract value from anyone else through this path. Self-redemption is self-destructive: it burns `amount` and pays back `selfBalance * amount / supply` of the same token, strictly less whenever anyone else holds tokens. It is also non-dilutive, leaving every other holder's per-token claim on the self-holding unchanged.

The issue is that the listing signal becomes meaningless. The backing badge is a product feature: a backed launch advertises a token that accumulates a basket it can be redeemed against. A self-backed launch advertises the badge while the backing is circular, and `isBacked()` , the registry's recorded backing recipe, and any UI built on them cannot distinguish "backed by a basket of stocks" from "backed by itself".

RECOMMENDATION

Add the same guard `BackedToken` already has to the launch path:

```
if (asset == address(this)) revert SelfAsset();
```

The factory-level check cannot compare against `address(this)` because the token does not exist yet at `_checkRecipe` time; the cleanest place is the `LaunchToken` constructor, which knows its own address. Applying the same exclusion to the rewards recipe as well is cheaper than arguing why a self-referential rewards leg is harmless.

I-08 INFORMATIONAL ● FIXED

Deployment Script Review

1. The post-deploy check never confirms `setFactory()` landed

`VerifyLaunchpad` checks the hook flags, the hook-to-seeder pairing, and the registry-to-factory link. It does not check `seeder.factorySet()` or `seeder.launchFactory() == factory`. The `require` calls inside `run()` only execute in forge's simulation, so if the separate `setFactory` transaction at line 91 drops or fails, the verify step still passes. The result is the M-04 brick: launches collect money, then neither graduation nor rescue can ever succeed.

Fix: add both checks to `VerifyLaunchpad`.

2. The deployment record is written during simulation

`vm.writeJson` at line 126 runs in the dry-run pass. A run without `--broadcast`, or a broadcast that fails after simulation, still writes `config/launchpad_deployment.4663.json` with addresses that have no code. The comment says that file gets copied to droplets and trusted. The `OVERWRITE` guard then blocks the real run until someone sets `OVERWRITE=true`, at which point the guard protects nothing.

Fix: write the record from a post-broadcast step, or gate the write on `vm.isContext(ScriptBroadcast)`.

3. Staging and production share one record file

Fork, staging, and mainnet all use chain id 4663 and the same filename. A staging rehearsal with `OVERWRITE=true` replaces the production record. `DEPLOY_LABEL` is stored but nothing reads it.

Fix: put the label in the filename, or refuse to overwrite a record whose label differs.

4. The factory constructor does not check the registry points back at it

The registry is deployed with a nonce-predicted factory address. The seeder side is fail-closed because `PoolSeeder`'s constructor asserts the hook pairing. The registry side is fail-open. Any nonce drift between simulation and broadcast gives a sealed factory where every `launch()` reverts at the registry. `VerifyLaunchpad` catches it afterwards, but only if someone runs it.

Fix: `if (registry_.factory() ≠ address(this)) revert` in the `LaunchFactory` constructor. A replacement deploy is happening anyway, so the contract change is cheap now.

5. Hardcoded `POOL_MANAGER` with no chain id guard

Neither `PoolSeeder` nor `LaunchFactory` touches the manager in its constructor. A deploy against the wrong chain succeeds silently and only fails at graduation, with funds already on the curves.

Fix: `require(block.chainid == 4663)` and a code-length check on the manager.

6. Dead code in the staging rescue script

`StagingRescuePath` has an unreachable rescue-and-refund block after the `return` at line 140, including a second `stopBroadcast`. Cosmetic.

AUDIT COMPLETE

Stay secure.

Thank you for choosing SBSecurity.

WEBSITE

sbsecurity.net

TELEGRAM

@sbsecurity_audits

REPORT

Bucket Launchpad · September 2026