



October 2026

Prepared for
UFG

Audited by
fedebianu
panda
yBorg fellows

UFG Factory and Treasuries

Smart Contract Security Assessment

Contents

1	Review Summary	2
1.1	Protocol Overview	2
1.2	Audit Scope	2
1.3	Risk Assessment Framework	3
1.3.1	Severity Classification	3
1.4	Key Findings	3
1.5	Overall Assessment	3
2	Audit Overview	4
2.1	Project Information	4
2.2	Audit Timeline	4
2.3	Audit Resources	4
2.4	Critical Findings	4
2.5	High Findings	4
2.6	Medium Findings	4
2.6.1	Platform fee fraction uses the curve fee, but graduated pools charge an independent hook fee	4
2.7	Low Findings	6
2.7.1	Matured route replacements can erase a later emergency pause	6
2.7.2	Pons fee rescues bypass UFG’s cut and strand memecoins in the router	7
2.7.3	UFGRevenueGovernance guardian nomination cannot be cancelled and never expires	10
2.7.4	FundingCollector controller can replace the guardian instantly, bypassing the route-change safeguard	11
2.7.5	Keeper settle() order does not commit to the output asset, so minOut is reinterpreted after a destination change	13
2.8	Gas Savings Findings	14
2.8.1	Use ReentrancyGuardTransient in the factory module, as the revenue splitter already does	14
2.9	Informational Findings	15
2.9.1	Curve fees below Pons’s sweep floor can stay in the curve, because the router never sweeps it	15
2.9.2	Coins that a buyback vest credits to the router stay in the Pons escrow and still count as supply	17
2.9.3	Splitting a redemption lowers the effective exit fee	19
2.9.4	A creator can deploy a coin’s treasury with a different exit fee from the one its launch named	20
2.9.5	Every empty escrow claim emits FeeClaimFailed and wastes gas	22
2.9.6	UFG’s share of volume is above the stated 0.02% during Pons’s snipe-tax window	23
2.9.7	UFGLaunchTreasury.redeem accepts the treasury itself as recipient	24

1 Review Summary

1.1 Protocol Overview

United Founders Group (UFG) has built a protocol that gives each coin launched on Pons its own treasury. **UFGFactory** deploys a fee router and a treasury for each coin. The router receives the coin's creator fees, keeps a fixed share for UFG and pays the rest to the treasury, and a holder can burn the coin for a share of the reserve. The revenue module receives UFG's share, splits it between a corporate branch and a branch that funds the treasuries of the UFG token, and converts each part through Uniswap v4 routes under a timelocked governance. **UFGTreasuryV2** backs the UFG token with sfrxETH next to the live v1 treasury, and the funding system moves ETH from Robinhood Chain to Ethereum, mints sfrxETH and sends it to v2. The factory and the revenue module are not deployed. Treasury v2 and the funding system are live.

1.2 Audit Scope

This review covers 21 Solidity files in four modules, totaling approximately 1836 lines of code. It was an AI-driven review: yAgent searched the code, and the auditors triaged and verified every reported issue before including it in this report.

```

├─ UFG-factory/src
│   ├── UFGFactory.sol
│   ├── UFGLaunchFeeRouter.sol
│   ├── UFGLaunchTreasury.sol
│   └─ interfaces
│       ├── IUFGPonsLaunchFactory.sol
│       └─ IUFGRevenueSplitter.sol
├─ UFG-revenue/src
│   ├── IRevenue.sol
│   ├── RevenueAsset.sol
│   ├── RevenueFeedPriceSource.sol
│   ├── UFGRevenueConverter.sol
│   ├── UFGRevenueGovernance.sol
│   ├── UFGRevenueSplitter.sol
│   └─ UniswapV4RevenueRoute.sol
├─ shared/src/interfaces
│   └─ IPonsV2.sol
├─ treasury-v2/src
│   ├── UFGTreasuryV2.sol
│   └─ interfaces
│       └─ IUFGTreasury.sol
└─ treasury-v2-funding/src
    ├── FundingCollector.sol
    ├── SfrxEthFundingHelper.sol
    ├── interfaces
    │   ├── IExternal.sol
    │   └─ IFundingRoute.sol
    └─ routes
        ├── AcrossRoute.sol
        └─ CanonicalRoute.sol

```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
■ Critical	0
■ High	0
■ Medium	1
■ Low	5
■ Informational	7

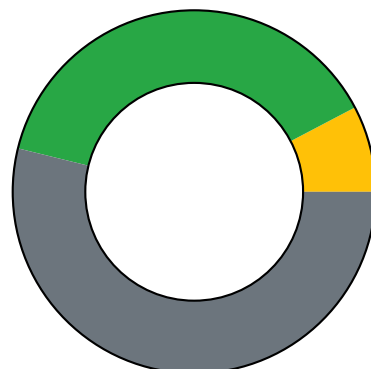


Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

2 Audit Overview

2.1 Project Information

Protocol Name: UFG

Repository: <https://github.com/u fgfinancialdefi/ufg-audit>

Commit Hash: 50824c3e32eaa9a2ebc0d1d1e9c0895522e73aae

Final Commit Hash: 26d23fab1ea6568c7d1553a4615e7beb58d3c2de

Commit URL: <https://github.com/u fgfinancialdefi/ufg-audit/tree/50824c3e32eaa9a2ebc0d1d1e9c0895522e73aae>

2.2 Audit Timeline

The audit was conducted on October 7, 2026.

2.3 Audit Resources

- Code repository and the brief (README) of each module

2.4 Critical Findings

None.

2.5 High Findings

None.

2.6 Medium Findings

2.6.1 Platform fee fraction uses the curve fee, but graduated pools charge an independent hook fee

The platform fee fraction is derived from the bonding curve's `feeBps`, but after graduation Pons charges the pool's `hookFeeBps`, an independent parameter. UFG's share of volume is therefore wrong for every graduated coin whose two fees differ, and it can never be corrected.

Technical Details

`UFGFactory._fractionFromCurve()` reads `creatorTaxBps`, `feeBps` and `protocolFeeShareBps` from the curve and `platformFeeFraction()` computes:

$$\text{denominator} = \text{curveFeeBps} \times (10,000 - \text{protocolFeeShareBps}) + \text{creatorTaxBps} \times 10,000$$

The NatSpec at `UFGFactory.sol:190-192` and assumption A4 of the README state that Pons hands the curve's fee values to the pool at graduation, so "one fraction holds in both phases". The Pons V2 source at the commit the README pins (`ponsdotdev/ponsfamily@fb7b593`) shows otherwise:

- On the curve, `feeBps` is the launch config's `curveFeeBps` (`PonsV2LaunchDeployer.sol:97`, `PonsV2LaunchFactory.sol:826`).
- In the pool, `PonsV2MemeHook._afterSwap` charges `info.hookFeeBps` (`PonsV2MemeHook.sol:465`). That value comes from the `FeePolicySnapshot` stored in `_launchFeePolicies[token]` at launch and passed to `registerPool` (`PonsV2LaunchFactory.sol:1371-1383`). It is a copy of the hook's global `hookFeeBps` at launch time.
- The two are separate settings. `launchToken` checks them separately (`config.curveFeeBps + creatorTaxBps` at `PonsV2LaunchFactory.sol:801`, `policy.hookFeeBps + creatorTaxBps` at `:804`). The Pons owner can change either one (`updateLaunchConfig` for the curve fee, `PonsV2MemeHook.setHookFeeBps` for the pool fee), and different launch configs can have different `curveFeeBps` against a single global `hookFeeBps`.
- Pons exposes the value that the pool uses through `getLaunchFeePolicy(token)` (`PonsV2LaunchFactory.sol:395`), but the factory never reads it.

The router stores the fraction once in `activate()` and has no way to update it, so the error lasts for the life of the coin.

Impact

Medium. After graduation, the real creator rate is $\text{hookFeeBps} \times (10,000 - \text{share}) + \text{creatorTax} \times 10,000$, but the router applies the fraction built from `curveFeeBps`. These figures use the README's values (protocol share 3,000 bps, creator tax 100 bps, curve fee 100 bps, fraction 20,000 / 1,700,000):

<code>hookFeeBps</code> at launch	Real creator rate after graduation	What UFG actually takes	Who loses
50	135 bps	1.59 bps (−21%)	UFG
100	170 bps	2 bps (correct)	—
300	310 bps	3.65 bps (+82%)	the coin
1,000 (Pons max)	800 bps	9.4 bps (4.7×)	the coin

Most of a coin's volume usually comes after graduation, so this mispricing covers most of the fees it pays. When `hookFeeBps` is higher than `curveFeeBps`, the extra fee comes out of the treasury reserve that backs the coin. When it is lower, UFG under-collects. No attacker is needed: a normal Pons configuration where the two fees differ is enough. Known issue K9 covers only price-impact drift in the pool, not a different fee.

Recommendation

- In `deployTreasury`, read `PONS_FACTORY.getLaunchFeePolicy(token)` and build a second fraction from `hookFeeBps` and that snapshot's `protocolFeeShareBps`.
- Store both fractions in the router, and apply the pool fraction to fees claimed once the coin's Pons phase is `PoolCreated`.

- A simpler alternative: have `deployTreasury` revert with `UnsupportedFeePolicy` when `hookFeeBps != curve.feeBps()`, so that such coins go through `rescue`.
- In either case, correct the NatSpec at `UFGFactory.sol:190-192` and assumption A4 in the README.

Developer Response

Accepted. We compared the live factory and hook on Robinhood Chain with the pinned commit. The fee paths match the description. Both fees are 100 bps today, so no deployed coin is affected.

Fixed in `ufgfinancialdefi/ufg-audit@7b97cdc` (tag `audit-2026-10-fixes`). `deployTreasury` reads `getLaunchFeePolicy(token)` and builds a second fraction from the stored `hookFeeBps`. The router stores both fractions. It applies the pool fraction once the coin's Pons phase is `PoolCreated`. The router reads the phase from the Pons record on each claim until the record shows the pool, then stores that result. If the record call reverts, the claim uses the curve fraction and still completes. `totalPlatformFees` is the sum of each phase's claims at that phase's fraction. The NatSpec, section 6 and assumption A4 of the brief are corrected. A fork test sets the live hook fee to 3% before a launch and checks that the pool phase pays 0.02% of volume. The invariant suite runs with unequal fees and includes a graduation action.

One limitation remains and is documented in section 6: fees credited on the curve but not claimed before the pool exists pay the pool fraction. Anyone can call `collect` at graduation to avoid it.

2.7 Low Findings

2.7.1 Matured route replacements can erase a later emergency pause

Technical Details

`UFGRevenueConverter.setRoute()` replaces the complete route record and unconditionally initializes `paused` to `false` (`UFGRevenueConverter.sol#L49-L63`):

```
1 _routes[input][output] = Route(adapter, priceSource, maxInput, slippageBps, false);
```

This operation is authorized through a seven-day governance proposal, but the proposal records only the new adapter, price source, cap, and slippage allowance. It does not bind or preserve the pair's pause state at execution time. The whole-struct assignment therefore lets delayed configuration replacement overwrite the separate emergency authority enforced by `setRoutePaused()` (`UFGRevenueConverter.sol#L65-L72`) and checked in `_swap()` (`UFGRevenueConverter.sol#L107-L122`). The revenue brief states that a pause has no expiry (`README.md#L222`) and that pauses can only stop activity (invariant 7, `README.md#L235-L236`).

The controller can schedule a same-pair route update while it is benign. If a later incident makes both the active route and proposed replacement unsafe, the controller or guardian can pause the pair, but any account can still execute the matured proposal before it is canceled (`UFGRevenueGovernance.sol#L57-L63`). Execution installs the proposed route with `paused = false`. The keeper can then call `settle()` (`UFGRevenueSplitter.sol#L100-L134`) and, if no effective feed floor applies, sell cap-sized portions until the affected branch's claimable

input is consumed. An honest keeper does this too: its script prices `minOut` from `quote()`, which reads the deteriorated venue. An adverse or compromised keeper can use any minimum. Cancellation ([UFGRevenueGovernance.sol#L65-L71](#)), a branch or input-asset pause, or keeper removal can stop this path, but each requires a separate protective transaction.

Impact

Low. A successful exploit can cause medium execution-price loss across all then-claimable splitter revenue for the affected branch and asset. Destination and allocation accounting remain intact, and the path cannot access launch or treasury reserves. Exploitation requires an uncommon overlap of an unsafe uncanceled matured proposal, a later emergency pause, claimable routed revenue, and a route without an effective feed floor; the revenue module is undeployed and its production configuration remains undecided.

Recommendation

Preserve the existing `paused` value when replacing an already configured pair, while keeping initial route creation behavior separate if new routes should start enabled:

```
1 -     _routes[input][output] = Route(adapter, priceSource, maxInput,
    slippageBps, false);
2 +     bool paused = _routes[input][output].paused; // a replacement keeps an
    emergency pause
3 +     _routes[input][output] = Route(adapter, priceSource, maxInput,
    slippageBps, paused);
```

Require the guardian or timelock to call `setRoutePaused(..., false)` only after responders validate a paused pair's replacement.

Developer Response

Accepted. We checked `setRoute`, `setRoutePaused`, the governance wrapper and the brief. A matured replacement wrote `paused` as `false` and so undid a pause set during the delay, against the brief's statement that a pause has no expiry.

Fixed in [ufgfinancialdefi/ufg-audit@a4c7369](#) (tag `audit-2026-10-fixes-2`). `setRoute` reads the pair's current `paused` flag and carries it into the new record, so an emergency pause survives a matured proposal. A new pair still starts enabled. A unit test queues a replacement, pauses the pair as the guardian during the delay, executes the proposal, and checks that the pair stays paused and that `settle` reverts until the guardian unpauses it. Section 4.4 of the brief states that a replacement keeps the pause state.

2.7.2 Pons fee rescues bypass UFG's cut and strand memecoins in the router

Pons's owner-only rescue paths pay a coin's creator fees straight to the router instead of through the fee escrow. The router treats those payouts as donations, so UFG's share is never charged, and the memecoin part of a pool rescue stays in the router forever while the treasury keeps counting it as supply.

Technical Details

The router taxes only what arrives during its own escrow claim. Anything else that reaches it is treated as a donation and passed to the treasury in full ([UFGLaunchFeeRouter.sol#L112-L120](#), [#L168-L176](#)). The router also has no function that can move any asset other than its quote asset.

The live Pons V2 stack has two owner-only paths that pay the creator-fee recipient (the router) directly, bypassing [PonsV2FeeEscrow](#):

- [PonsV2LaunchFactory.rescueCurveFees\(token\)](#) calls [PonsV2BondingCurve.rescueFees\(\)](#) (L850 of the [verified live curve](#)), which sends the creator's whole bucket to `deployer`, the current creator-fee recipient, with a plain transfer (L868).
- [PonsV2MemeHook.rescuePoolFees\(poolId\)](#) pays each pending currency of a graduated pool directly to `info.creator` ([_rescueCurrency](#)). The hook takes its fee in the output currency of each swap, so after buys a large part of the pending fees is in the memecoin itself. The rescue pays that part to the router as memecoin.

Neither function has a delay or a precondition beyond having something pending. The brief's actor table says that Pons's owner can replace the recipient only after 3 days and "cannot move assets that a treasury or a router holds". That last statement remains true: the rescue paths take nothing that the router or the treasury holds. What the brief does not cover is that the owner can change, at once, how creator fees reach the router.

Pons's own natspec states when the paths are used. `rescueCurveFees` calls the curve's `rescueFees`, which exists because "a permissioned quote asset can stop delivering to" the escrow. `rescuePoolFees` covers "an approved pair token that later turns out to be fee-on-transfer, rebasing, or otherwise unable to move its exact nominal amount", and it rescues the memecoin leg "regardless of what the quote currency is". Pons has approved about 80 pair tokens. Nearly all are Robinhood stock tokens, whose shared [implementation](#) has pause, blacklist and `adminBurn` controls, and the rest include USDG, which the issuer can freeze. These are the cases that the brief places outside its assumptions A7 and A8. When one of those assumptions fails for a quote asset, Pons's documented recovery is the path that skips UFG's fee and strands the memecoin share. Both functions are present at the commit the brief cites ([fb7b593](#)) and in the verified live contracts.

Two consequences follow:

1. Quote-asset payouts arrive outside `_claimFromPons`, so `totalClaimed` does not grow and UFG's share is never charged on them. The router applies the README's rule "donations pay no platform fee" to fees that are not donations.
2. Memecoin payouts stay in the router forever. [UFGLaunchTreasury.effectiveSupply](#) subtracts only `DEAD` and `LOCKER` ([UFGLaunchTreasury.sol#L172-L177](#)), and `burnStray` burns only the treasury's own balance ([#L153-L160](#)). The coins can never be redeemed, yet every redemption divides by a supply that includes them. Coins that a user sends to the router by mistake end up the same way, but that is the sender's own loss, which K4 already accepts. This finding concerns the rescue payouts, which are the coin's own fee income.

Impact

Low. Both paths need an action by Pons's owner, and the amounts are bounded by the fees pending since the last sweep. The trigger is realistic, though: it is Pons's documented response to a quote asset that breaks A7 or A8, and most approved quote assets have issuer controls that can cause it. The accepted risks of the brief do not cover the effect:

- K4 accepts that assets sent to the router stay there as the sender's loss. Here the coins are fee income of the coin, they arrive through a Pons protocol path, and every holder bears the dilution.
- The Pons-owner risk that the brief accepts is the recipient override, which waits 3 days and is public. The rescue paths take effect at once and are not documented.
- The rule that donations pay no platform fee covers voluntary gifts, not creator fees.

The two effects, measured on the fork:

- UFG loses its share of every rescued quote-asset payout. On the fork, 0.017 ETH of creator fees on 1 ETH of volume reached the treasury untaxed, and UFG's 0.0002 ETH was never charged.
- The creator's share of pending memecoin fees is lost to the holders as reserve. The coins stay in the effective supply, so the remaining reserve is spread over coins that can never redeem, permanently. On the fork, a single 0.5 ETH buy in the pool followed by a rescue left 369,083 coins (0.04% of the effective supply) in the router.

Recommendation

Give the router a permissionless exit for the launch coin. Burning it keeps the supply rule exact and needs no owner:

```
1 +   /// @notice Burns any coins of this launch held here. Pons's pool-fee
   rescue pays the
2 +   ///           creator recipient in the memecoin, and users can send coins
   here by mistake.
3 +   function burnStray() external nonReentrant {
4 +       address t = treasury;
5 +       if (t == address(0)) revert NotActive();
6 +       IERC20 coin = UFGLaunchTreasury(payable(t)).token();
7 +       uint256 bal = coin.balanceOf(address(this));
8 +       if (bal != 0) coin.safeTransfer(UFGLaunchTreasury(payable(t)).DEAD(),
   bal);
9 +   }
```

For the untaxed quote-asset payouts, at least document in section 5 and section 8 that Pons's owner can pay creator fees outside the escrow, and that UFG's share is not charged on them. If UFG wants to charge them, the router would need to tax every inflow except an explicit, separate donation entry point.

Developer Response

Accepted. We checked both rescue paths in the live factory, curve and hook, and in the pinned commit. They pay the creator-fee recipient directly, as described.

Fixed in `ufgfinancialdefi/ufg-audit@8047cad` (tag `audit-2026-10-fixes`). The router has a permissionless `burnStray()` that sends any of the launch coin it holds to the dead address. A fork test calls `rescuePoolFees` as Pons's owner after a 0.5 ETH pool buy, finds the same 369,083 coins on the router, burns them and checks the supply. Section 5, section 8 and K4 of the brief describe both rescue paths and state that a rescued quote-asset payout reaches the treasury without the platform share. The router still charges the platform share only on the escrow delta. The rescue is an owner-only path with a limited amount, and a charge on every inflow would also apply to donations.

2.7.3 UFGRevenueGovernance guardian nomination cannot be cancelled and never expires

In `UFGRevenueGovernance`, a guardian nomination cannot be cancelled and never expires. A nominee whose key is compromised, or a nomination that governance no longer wants, stays acceptable until another 7-day proposal replaces it.

Technical Details

`nominateGuardian` can only be called by the timelock, and it rejects `address(0)`:

```
1 function nominateGuardian(address next) external {
2     if (msg.sender != address(TIMELOCK)) revert Unauthorized();
3     if (next == address(0) || next == guardian) revert BadAddress();
4     pendingGuardian = next;
5     emit GuardianNominated(next);
6 }
```

`acceptGuardian` has no deadline, and no function clears `pendingGuardian`. Controller nominations have `cancelControllerNomination`; guardian nominations have no equivalent.

To revoke a nomination, governance has to schedule another `nominateGuardian` with an address nobody can accept (for example `0x...dEaD`), which takes the full 7-day delay. Until that executes, the nominee can call `acceptGuardian` at any time. A nomination left in place is also still acceptable years later.

Impact

Low. The guardian cannot move assets, but known issue K3 says it "can block all maintenance, including its own replacement": it can cancel every proposal and every controller nomination. If an unwanted nominee accepts, either because its key was compromised during the 7-day window or because a stale nomination was forgotten, it can never be removed. Governance is then frozen: no splitter upgrade, converter, route, destination or keeper change can execute. The preconditions (a compromised or unwanted nominee, before governance can replace the nomination) keep the likelihood low.

Recommendation

Let the controller or the current guardian cancel a pending guardian nomination, and give the nomination an expiry:

```
1 + uint256 public guardianNominationExpiresAt;
2 +
3   function nominateGuardian(address next) external {
4       if (msg.sender != address(TIMELock)) revert Unauthorized();
5       if (next == address(0) || next == guardian) revert BadAddress();
6       pendingGuardian = next;
7 +   guardianNominationExpiresAt = block.timestamp + DELAY;
8       emit GuardianNominated(next);
9   }
10 +
11 + function cancelGuardianNomination() external {
12 +     if (msg.sender != controller && msg.sender != guardian) revert
13         Unauthorized();
14 +     pendingGuardian = address(0);
15 +     emit GuardianNominated(address(0));
16 + }
17
18   function acceptGuardian() external {
19 -     if (msg.sender != pendingGuardian) revert Unauthorized();
20 +     if (msg.sender != pendingGuardian || block.timestamp >
21         guardianNominationExpiresAt) revert Unauthorized();
```

Developer Response

Accepted.

Fixed in [ufgfinancialdefi/ufg-audit@cd3abce](#) (tag `audit-2026-10-fixes`). A guardian nomination carries a 7-day acceptance window in `guardianReadyUntil`. After that window `acceptGuardian` reverts with `NotReady`. The controller or the current guardian can call `cancelGuardianNomination`, as with controller nominations. The roles table and the guardian change paragraph of the brief are updated.

2.7.4 FundingCollector controller can replace the guardian instantly, bypassing the route-change safeguard

The controller of `FundingCollector` can replace the guardian with no delay. A compromised controller can therefore remove the guardian first and then push a malicious route through the 7-day delay, which defeats the safeguard the README relies on.

Technical Details

The README names the guardian as the defence against a bad route proposal (section 8, consequence of A10): "A controller can propose a route that sends ETH to any address. The route becomes active after 7 days. [...] The guardian can make the fallback route active, which removes the proposal."

That defence only holds while the guardian stays in place. `setGuardian` is `onlyController` and takes effect immediately:

```
1 function setGuardian(address next) external onlyController {
2     emit GuardianSet(guardian, next);
3     guardian = next;
4 }
```

With a compromised controller key:

1. `proposeRoute(malicious)`. `_validate` only checks that the route has code, returns this collector from `COLLECTOR()` and returns at least `MIN_BATCH` from `maxAmount()`. A contract that keeps the ETH passes.
2. `setGuardian(address(0))`, or an address the attacker controls, in the same block. The old guardian can no longer call `useFallback` or `pause`.
3. After 7 days anyone calls `activateRoute`, and every later `sendBatch` (permissionless) sends the funding ETH to the malicious route.

Nobody else holds a role that can intervene during the 7 days. By contrast, revenue governance can change its guardian only through a delayed timelock proposal (`UFGRevenueGovernance.nominateGuardian`).

Impact

Low. Exploiting it requires a compromised controller, which assumption A10 trusts. Today the controller and the guardian are the same address (K13), so the guardian is not yet an independent check. The issue matters once the roles are split into separate multisigs, as the owner plans: the 7-day delay and the guardian then look like a safeguard against a compromised controller but do not provide one. The ETH at risk is everything the collector receives from the splitter's V2 branch after the malicious route becomes active.

Recommendation

Put guardian changes behind the same delay as route changes, so the old guardian can still react to a pending proposal. For example, `nominateGuardian(next)` stores the nominee with `readyAt = block.timestamp + ROUTE_DELAY`, and `acceptGuardian()` applies it after the delay. Alternatively, block `setGuardian` while a route proposal is pending, and let only the guardian clear a pending route during the delay.

Developer Response

Acknowledged, with a documentation change.

The guardian is a check on a wrong route proposal, such as a route with a bug. It is not a defence against a compromised controller key, which assumption A10 excludes. The controller will be a 4-of-5 multisig. Instant guardian replacement is a design choice: a guardian can delay batches without limit (K2), so the controller must be able to remove it at once. A delay on guardian changes would not close the case on its own. A compromised controller could

nominate a new guardian, wait 7 days, and then propose a route. The limitation on a compromised controller key is upstream: revenue governance can move the splitter's v2 branch away from the collector with a 7-day proposal.

Addressed in `ufgfinancialdefi/ufg-audit@3b6f8b8` (tag `audit-2026-10-fixes`). The consequence of A10, the roles table, K2 and K13 of the funding brief state this.

2.7.5 Keeper `settle()` order does not commit to the output asset, so `minOut` is reinterpreted after a destination change

A keeper's `settle()` order does not commit to the output asset. If a matured `setDestination` changes the CORPORATE asset before the order lands, the keeper's raw `minOut` is applied in the new asset's units, and on a route without a price source it can become a meaningless floor.

Technical Details

`settle(branch, inputAsset, amountIn, minOut)` reads `s.destinations[branch]` when it executes. `convert()` reads the destination from the splitter again and selects `_routes[inputAsset][output]`. When that route has no price source, `_swap()` uses the keeper's `minOut` as the only floor, and `settle()` checks `amountOut >= minOut` in the units of whatever the destination asset is at that moment.

`setDestination` takes effect immediately when the timelock executes it, and `UFGRevenueGovernance.execute()` is permissionless once the proposal has matured, so anyone can choose the moment of the switch. A keeper order sent with `minOut = 100e6` (100 units of a 6-decimal stablecoin) that executes after a switch to an 18-decimal asset accepts as little as `1e-10` units of that asset.

Scope and preconditions:

- Only the CORPORATE branch is affected. `_setDestination` forces the V2 branch's asset to `address(0)` (L174), so its output never changes.
- A route must already exist for `(inputAsset, newAsset)`. Otherwise `_swap` reverts with `InvalidRoute`.
- That route must have no price source. A price source sets its own floor that does not depend on `minOut`.
- The keeper's order must be pending exactly when the change executes, and the attacker needs control over transaction ordering (Robinhood Chain uses a first-come-first-served sequencer) plus the ability to move the new pool's price.
- The loss is limited to one sale, capped by the route's `maxInput`.

Impact

Low. With the preconditions above, one keeper sale can execute with almost no slippage protection, and the value is lost to whoever moves the new pool's price. Every precondition depends on a governance change announced 7 days in advance, which the keeper can see coming. This is a variant of K2 (the contract cannot check the keeper's minimum), but it needs no bad or stolen keeper key.

Recommendation

Make the order commit to the asset the keeper priced it in:

```
1 - function settle(uint8 branch, address inputAsset, uint256 amountIn, uint256
   minOut)
2 + function settle(uint8 branch, address inputAsset, uint256 amountIn, uint256
   minOut, address expectedOutputAsset)
3     external nonReentrant returns (uint256 amountOut)
4     {
5         ...
6         Destination memory d = s.destinations[branch];
7 +     if (d.asset != expectedOutputAsset) revert BadSettlement();
```

A deadline parameter is optional and adds protection against stale orders in general. Operationally, pause the CORPORATE branch before a proposal that changes its asset executes, and resume keeper sales only after the change.

Developer Response

Accepted.

Fixed in `ufgfinancialdefi/ufg-audit@f7f0a4d` (tag `audit-2026-10-fixes`). `settle` takes a fifth argument, the asset in which the caller priced `minOut`, and reverts with `BadSettlement` when the stored destination's asset differs. Section 4.2 of the brief states the rule. The changes table adds an operational rule: pause the corporate branch before a `setDestination` proposal executes. We did not add a deadline parameter. The expected asset closes this case, and K2 covers a stale order at a bad price.

2.8 Gas Savings Findings

2.8.1 Use `ReentrancyGuardTransient` in the factory module, as the revenue splitter already does

Technical Details

`UFGFactory`, `UFGLaunchFeeRouter` and `UFGLaunchTreasury` inherit OpenZeppelin's storage-based `ReentrancyGuard` (`UFGFactory.sol#L42`, `UFGLaunchFeeRouter.sol#L27`, `UFGLaunchTreasury.sol#L23`). `UFGRevenueSplitter` already uses `ReentrancyGuardTransient` (`UFGRevenueSplitter.sol#L14`). The module compiles for `cancun` (`hardhat.config.ts#L82`), and Robinhood Chain executes `TSTORE` and `TLOAD`: an `eth_call` that stores a transient slot and loads it back returns the stored value.

The storage guard writes its slot in each constructor, and every guarded call reads the slot cold and writes it twice. `deployTreasury` deploys a router and a treasury. A launch-treasury `redeem` passes two guards, the treasury's and the one in the router's `claim`.

Measured on the module's mocks at `50824c3`, with each call in its own transaction (`forge test --isolate`) and the project's compiler settings:

Call	ReentrancyGuard	ReentrancyGuardTransient	Saved
deployTreasury	2,813,169	2,773,021	40,148
collect	128,915	126,961	1,954
redeem	133,687	129,828	3,859
UFGFactory deployment	4,350,657	4,316,357	34,300

Every `collect` and every `redeem` of every coin saves the same amount, whatever the call does.

Impact

Gas. The factory module is not deployed yet, so the change costs nothing now. It changes the creation code and therefore the predicted CREATE2 addresses. That is harmless before the first deployment, because `predictAddresses` hashes the creation code when it runs.

Recommendation

```

1 -import {ReentrancyGuard} from "@openzeppelin/contracts/utils/ReentrancyGuard.
   sol";
2 +import {ReentrancyGuardTransient} from "@openzeppelin/contracts/utils/
   ReentrancyGuardTransient.sol";
3 -contract UFGLaunchFeeRouter is IPonsV2FeeEscrow, ReentrancyGuard {
4 +contract UFGLaunchFeeRouter is IPonsV2FeeEscrow, ReentrancyGuardTransient {

```

Make the same change in `UFGLaunchTreasury` and `UFGFactory`.

Developer Response

Accepted. We confirmed that Robinhood Chain executes `TSTORE` and `TLOAD` with an `eth_call` that stores a transient slot and loads it back, that both compiler configurations target Cancun, and that the installed OpenZeppelin release ships the transient guard.

Fixed in `ufgfinancialdefi/ufg-audit@26d23fa` (tag `audit-2026-10-fixes-2`). `UFGFactory`, `UFGLaunchFeeRouter` and `UFGLaunchTreasury` inherit `ReentrancyGuardTransient`, as the revenue module does. The unit, fuzz, invariant and fork suites pass unchanged, including the size-limit test. The creation code and so the predicted addresses change, which is harmless before the first deployment. D11 of the brief records the guard as a difference from the live v1.

2.9 Informational Findings

2.9.1 Curve fees below Pons's sweep floor can stay in the curve, because the router never sweeps it

Technical Details

`UFGLaunchFeeRouter.claim()` takes only what the Pons escrow has credited to the router (`UFGLaunchFeeRouter.sol#L168-L176`). Fees that the coin's bonding curve still holds

(`quoteFeeBalance`, `creatorTaxBalance`) reach the escrow only when someone calls the curve's `sweepFees`.

The live curve is created by the verified `PonsV2LaunchDeployer`. In its `PonsV2BondingCurve.sol`, `sweepFees` accepts two callers (L579-L587): the hook's `feeSweepOperator` and the address stored in `deployer`. Despite its name, `deployer` holds the creator-fee recipient. The launch deployer passes `params.creatorFeeRecipient` to the curve's constructor (`PonsV2LaunchDeployer.sol` L151), and `setCreatorFeeRecipient` overwrites it whenever the recipient changes (L343-L347). For a coin with a UFG treasury, that address is the router. The curve refuses a caller other than the operator only while a buyback is pending (`_requiresTrustedOperator`, L725-L727), and the factory accepts only coins with buyback off. A `sweepFees(0)` sent by the router therefore succeeds before graduation, without a swap. The router has no function that sends it, and no other UFG contract does.

The `feeSweepOperator` is an unverified contract, `0xa1018c1D9655292A2dE0F7dEa9a0F848EaA8cA83`. Anyone can call its `sweepCurve(curve)`, but it reverts with `BelowFloor()` while the curve's pending fees are below a per-asset floor that its owner sets. On 2026-10-14, `floor(address(0))` returns `3.7e15` (0.0037 ETH) and the floor for USDG is `1e7` (10 USDG). Below the floor, only the recipient can sweep.

The brief states the opposite. Section 8 says "Only Pons's sweep operator can move fees from a curve or a pool into the escrow" ([README.md#L315-L317](#)). The flow diagram shows "Pons sweep (Pons's operator only)" ([#L83](#)), and the actor table gives the role only to the operator ([#L214](#)). The fork test comment says that the curve's deployer "is Pons's own deployer contract, not the launcher" ([UFGFactory.fork.test.ts#L157-L158](#)). The test shows only that the creator and the factory are refused. That is expected, because neither is the recipient.

After graduation the [live hook](#) also accepts the recipient in `sweepPoolFees`, but only while no fees in the coin are pending (`PonsV2MemeHook.sol` L500-L508 and L611-L617). In the pool phase the module therefore depends on the operator in any case.

Impact

Informational. When trading on a curve stops with less than the floor pending, those fees reach the treasury only if the coin graduates, if new trades push them past the floor, if the floor is lowered, or if Pons's owner rescues the curve's fees. Nothing is lost while one of these can still happen. The amount per coin stays below the floor, which is 0.0037 ETH or 10 USDG at today's settings. For a coin that earned little, that can still be a large part of its reserve. The floor is set by the operator's owner, who can raise it.

On a fork of Robinhood Chain, a 0.1 ETH buy left `2e15` wei pending, under the `3.7e15` wei floor. The operator refused to sweep it. A sweep sent as the router added `1.68e15` wei to the reserve.

Recommendation

Add a permissionless function to the router that sweeps the coin's curve. On the fixed router, which stores `token`:

```
1 +    /// @notice Moves the fees pending in the coin's bonding curve into the
    Pons escrow.
2 +    ///          Pons's operator skips curves below its floor. The curve also
    accepts
```

```

3 +   ///           this router, its creator-fee recipient. Anyone can call it.
4 +   function sweepCurve() external nonReentrant {
5 +       if (token == address(0)) revert NotActive();
6 +       IUFGPonsBondingCurve(PONS_FACTORY.getLaunchedToken(token).curve).
    sweepFees(0);
7 +   }

```

Add `sweepFees(uint256)` to `IUFGPonsBondingCurve` and a stub to its mock. A minimum of zero is safe: the curve refuses a sweep by the recipient while a buyback is pending, so this sweep never swaps. After graduation the curve reverts with `AlreadyGraduated`. Keep the call out of `claim()`, because an extra external call on every `collect()` and `redeem()` costs more gas than the residual is worth on most coins. The site or a keeper can call `sweepCurve()` and then `collect()` for curves that stopped trading below the floor.

Correct the flow diagram, the actor table and section 8 of the brief, and the comment in the fork test.

Developer Response

Accepted. We read the verified curve: `sweepFees` accepts the operator and the address in `deployer`, which holds the creator-fee recipient, and refuses a recipient only while a buyback slice is pending. We read the operator's floor over the public RPC, 3.7e15 wei for ETH on 2026-10-14. The brief and the fork test comment were wrong about who can sweep.

Fixed in `ufgfinancialdef/ufg-audit@7f431df` (tag `audit-2026-10-fixes-2`). We kept the sweep out of a new entry point. The router's `burnStray` looks up the coin's curve on the Pons factory and calls `sweepFees(0)` inside a `try`, so fees under the floor reach the escrow when anyone calls it, and `collect` takes them from there. A zero minimum never swaps, and the call is a no-op after graduation or during a Pons outage. The fork test now checks that the curve's `deployer` is the router, that the creator and the factory are refused, and that `burnStray` as a stranger moves a 1 ETH buy's pending fees into the escrow. On the fork the router swept 1e16 wei of curve fees and 1.7e16 wei were credited to it. The flow diagram, the actor table, section 8 and K12 of the brief describe the recipient sweep and the floor.

2.9.2 Coins that a buyback vest credits to the router stay in the Pons escrow and still count as supply

Technical Details

`UFGLaunchTreasury.effectiveSupply()` subtracts only the coins at `DEAD` and in the locker (`UFGLaunchTreasury.sol#L172-L177`). Coins held anywhere else count, including coins that the Pons fee escrow holds for the router. The router claims only its quote asset from the escrow: `_claimFromPons` calls `claim()` or `claimToken(quote)` (`UFGLaunchFeeRouter.sol#L168-L176`).

Pons can credit the coin itself to the router through a buyback vest. A coin that ran with buyback on has a five-year vest in `PonsV2BuybackVault`. `UFGFactory._checkRecord` checks only the record's current `buybackEnabled` flag (`UFGFactory.sol#L135-L146`), so such a coin can still get a treasury. There are two ways in the verified `PonsV2LaunchFactory`:

- A creator who launched as its own recipient with buyback on disables buyback and calls `transferCreatorFeeRecipient` to name the predicted router. `_setCreatorFeeRecipient` forwards the new recipient to the vest with `buybackVault.updateCreatorRecipient` (L988-L1006), and `deployTreasury` accepts the coin.
- A coin launched naming the predicted router with buyback on reaches the same state when Pons's owner disables its buyback. Disabling is the only change that the owner can make to that flag (`setBuybackEnabled`, L904-L921).

From then on, each `release(token)` in the `verified vault` credits the creator share to the router with `feeEscrow.creditToken(router, token, amount)` (PonsV2BuybackVault.sol L178-L198). The vault's protocol recipient can call `release`, and so can the router, which never does. Nothing in the module claims the credit. The router never calls `claimToken(token)`, and `rescueFees` reverts outside rescue mode.

The fix for the pool-rescue path (8047cad) adds `burnStray()` to the router, but that function burns only the coins that the router holds. The vest credit stays in the escrow. Assumption A6, "Pons credits creator fees in the quote asset only" ([README.md#L298](#)), does not hold for vest releases.

Impact

Informational. The path is outside the documented flow: it needs a coin that ran with buyback on before it had a treasury. The vest was bought with fees from before the treasury existed, so the holders lose nothing they were owed. Those coins also counted in the supply while they sat in the vault. What the holders miss is the increase in backing that burning the creator's share would give them.

On a fork of Robinhood Chain with the fixed code (`audit-2026-10-fixes`), a 2 ETH curve buy with buyback on locked 8.86e23 coins in the vest. One year after the treasury was attached, a release credited 1.24e23 coins to the router. After `collect()`, the treasury's `burnStray()` and the router's new `burnStray()`, the credit was still in the escrow. Claiming it as the router and burning it raised `backing()` from 16,800,000 to 16,802,083. The rest of the creator share follows over the five-year vest.

Recommendation

Make the router's `burnStray` claim the coin credit first. On the fixed router:

```

1      function burnStray() external nonReentrant returns (uint256 amount) {
2          address t = treasury;
3          if (t == address(0) || token == address(0)) revert NotActive();
4 +         // A buyback vest that names this router is released into the escrow
   in the coin.
5 +         try PONS_ESCROW.claimToken(token) returns (uint256) {} catch {}
6           IERC20 coin = IERC20(token);
7           amount = coin.balanceOf(address(this));

```

The `try` is required because the escrow reverts with `NoBalance` when there is nothing to claim. Update A6 and K4 to cover coins credited through a vest.

Developer Response

Accepted. We checked the live factory, the buyback vault and the escrow. Every recipient change updates the vest's beneficiary, `release` credits the creator share to that beneficiary in the coin, and only the credited address can claim its escrow balance. The router claimed its quote asset only, so a vest credit stayed in the escrow and counted as supply.

Fixed in `ufgfinancialdefi/ufg-audit@47d44d5` (tag `audit-2026-10-fixes-2`). The router's `burnStray` calls `claimToken(token)` on the escrow inside a `try` before it burns, so a vest release ends at the dead address. A unit test credits the coin to the router on the mock escrow, calls `burnStray`, and checks the credit at `DEAD` and the lower effective supply, and checks that an escrow outage does not stop the burn of what the router holds. A6, K4 and section 8 of the brief cover coins credited through a vest.

2.9.3 Splitting a redemption lowers the effective exit fee

Technical Details

Both treasuries leave the exit fee in the reserve and price each redemption against the current reserve and supply. `UFGLaunchTreasury.redeem()` computes `gross` from `treasury()` and `effectiveSupply()`, pays `net`, and burns the whole `amount` (`UFGLaunchTreasury.sol#L130-L145`):

```
1 uint256 t = treasury();
2 uint256 s = effectiveSupply();
3 if (s == 0) revert ZeroSupply();
4 uint256 gross = (amount * t) / s;
5 uint256 net = (gross * (BPS - EXIT_FEE_BPS)) / BPS;
```

`gross - net` stays in the reserve while the burn lowers the supply. The next redemption therefore gets a share of the fee that the previous one left. `UFGTreasuryV2.redeem()` does the same on its share leg (`UFGTreasuryV2.sol#L194-L214`).

A holder who splits a redemption into many calls gets part of their own fee back. Take an exit fee `f` and a holder with a fraction `x` of the effective supply. As the number of calls grows, the effective fee tends to $1 - (1 - (1 - x)^{(1 - f)}) / x$. At 15%:

Holder's share of the supply	1 call	10 calls	Limit
10%	15%	14.40%	14.34%
50%	15%	11.32%	10.96%
100%	15%	2.26%	0%

K2 of treasury v2 says that each redemption leaves 15% of its v2 claim in v2 (`treasury-v2/README.md#L227`), and K11 covers only rounding (`#L236`). Neither says that the holder can get part of that 15% back through later redemptions.

Impact

Informational. The effect matters only for large holders. No other holder loses principal. The others only receive less of the retained fees than a single redemption would have left them. The stated exit fee is therefore an upper bound, not the fee that a large holder pays. Treasury v2 is live and has no upgrade path, so for v2 only the documentation can change.

Recommendation

Document in both briefs that the exit fee applies to each call, and that a holder who splits a redemption pays less, down to the limit above. If the fee must not depend on how a holder splits a redemption, keep the retained fees out of the redeemable reserve, for example by sending them to a separate account. A per-address limit does not help, because a holder can move coins between addresses.

Developer Response

Accepted as documentation. We reproduced the table from the redemption formula of both treasuries. The effect is a property of every treasury that keeps the fee in the reserve, including the live v1, and the redeemer is one of the holders the fee is left to. Nobody loses principal, and the effect is small below a 10% share of the supply. Moving the fee out of the reserve would need a destination or an owner and would change the promise that UFG takes no part of it, so the code does not change.

Documented in `ufgfinancialdefi/ufg-audit@a22f2a6` (tag `audit-2026-10-fixes-2`). The factory brief (K13), the v2 brief (K17) and the v1 brief (K12) state that the exit fee applies to each call, that a holder who splits a redemption pays less, and the limit the effect tends to at each treasury's fee.

2.9.4 A creator can deploy a coin's treasury with a different exit fee from the one its launch named

Technical Details

The factory states that the exit fee is bound at launch (`UFGFactory.sol#L16-L17`):

```
1 /// The fee is hashed into the addresses, so the router the creator names
2 /// on Pons commits to it. Nobody can deploy the treasury with another fee.
```

The actor table of the brief also lists "Change the fee, the treasury or the router after the launch" among the things that the creator cannot do (`README.md#L209`).

The binding holds only while the coin's Pons recipient is the router named at launch. `rescue` hands the recipient role back to the creator and marks only the salt as used (`UFGFactory.sol#L168-L182`). `deployTreasury` accepts any salt whose predicted router is the coin's current recipient (`#L114-L146`). A creator can therefore:

1. Launch naming the router predicted for a 0% exit fee.

2. Call `rescue` before the treasury exists. This makes the creator the recipient.
3. Call Pons's `transferCreatorFeeRecipient` to name the router predicted for a 20% exit fee under a new salt. Pons applies a transfer by the current recipient immediately.
4. Call `deployTreasury` for that salt, or let anyone call it.

On the module's mocks these steps end with a 20% treasury for a coin that was launched naming the 0% router. The result is the same at `50824c3` and at `audit-2026-10-fixes`.

K1 accepts that the creator can rescue a coin that qualifies and keep its fees ([README.md#L323](#)). This case starts from the same step but ends in a state that K1 does not describe: the coin still gets a UFG treasury, with an exit fee that the launch did not name. K1's mitigation, the site deploying the treasury directly after the launch, works only if that deployment lands before the creator's `rescue`. After a rescue the creator can repoint the recipient and deploy at any later time.

Impact

Informational. The guarantee stated in the factory's NatSpec and in the actor table does not hold. Holders are no worse off than under K1, where the creator keeps all the fees and the coin has no treasury. The exposed buyers are those who traded between the launch and the rescue, relying on the fee that the launch named. The creator needs no other party and no delay.

Recommendation

Bind the fee to the coin, not only to the salt. Record rescued coins in `rescue` and refuse them in `deployTreasury`:

```
1      mapping(bytes32 => bool) public usedSalts;
2 +    /// @notice Coins handed back to their creator by `rescue`. They can never
    get a treasury.
3 +    mapping(address => bool) public rescued;
4    ...
5      if (launches[token].treasury != address(0)) revert AlreadyDeployed();
6 +    if (rescued[token]) revert InvalidLaunch();
7    ...
8      usedSalts[s] = true;
9 +    rescued[token] = true;
10     UFGLaunchFeeRouter router = new UFGLaunchFeeRouter{salt: s}(
        PONS_ESCROW, SPLITTER, PONS_FACTORY);
```

With this change the steps above revert in `deployTreasury`, and the module's unit, fuzz and invariant tests still pass. If rescued coins must stay eligible, state in the NatSpec, in the actor table and in K1 that a rescue releases the fee binding.

Developer Response

Accepted. We checked `rescue`, `deployTreasury` and the live Pons factory. A recipient's own `transferCreatorFeeRecipient` applies at once, so after a rescue the creator could point Pons at a router predicted for another fee and deploy that treasury. The guarantee in the NatSpec and the actor table did not hold.

Fixed in `ufgfinancialdefi/ufg-audit@cce508f` (tag `audit-2026-10-fixes-2`). The factory has a `rescued` mapping. `rescue` marks the coin, and `deployTreasury` refuses a rescued coin with `InvalidLaunch`. A unit test launches with the router for a 0% fee, rescues, repoints the mock recipient to the router for a 20% fee under a new salt, and checks that `deployTreasury` reverts. Section 4.3 and K1 of the brief say that a rescue is final for the coin.

2.9.5 Every empty escrow claim emits `FeeClaimFailed` and wastes gas

The live Pons escrow reverts when a recipient with no credit claims. The router calls it on every `collect()` and `redeem()`, catches the revert and logs it as `FeeClaimFailed`. Between Pons sweeps every claim is empty, so holders pay about 7,100 gas for a failing call on every redeem.

Technical Details

`_claimFromPons` always calls the escrow and logs any revert as a failure ([UFGLaunchFeeRouter.sol#L168-L176](#)):

```
1 try PONS_ESCROW.claim() returns (uint256) {} catch (bytes memory reason) { emit
  FeeClaimFailed(reason); }
```

The live `PonsV2FeeEscrow` reverts with `NoBalance()` when the caller has no credit (`_claim` at L108 and `_claimToken` at L124: `if (amount == 0) revert NoBalance();`). Every `collect()` and every `redeem()` that runs while the router has no new credit therefore emits `FeeClaimFailed(0xc2caa2a6)`. On a coin that trades rarely, that is most calls.

The event only records the caught revert, with its reason. The cost is gas: the escrow call fails on every `collect()` and every `redeem()` between Pons sweeps, and on a `redeem()` the holder pays for it.

Impact

Informational. No funds are at risk. Each empty claim costs a failing external call, which holders pay on every `redeem()` between sweeps. Measured on a fork of Robinhood Chain against the live escrow, the check in the recommendation lowers `collect()` with nothing credited from 58,090 to 50,955 gas (7,135 saved, about 12%) and removes the event. `redeem()` runs the same claim first. With credit waiting the check adds 1,047 gas (130,911 to 131,958), because the extra view call comes before the claim. Between Pons sweeps every claim is empty, so the check saves gas overall.

Recommendation

Skip the escrow call when there is nothing to claim. Read the credit inside `try` too, so that an escrow outage still lets `claim()` deliver the funds already held, as it does now:

```
1 function _claimFromPons() private returns (uint256 claimed) {
2 +     if (quote == address(0)) {
3 +         try PONS_ESCROW.balanceOf(address(this)) returns (uint256 credit) {
4 +             if (credit == 0) return 0; } catch {}
5 +     } else {
```

```

5 +         try PONS_ESCROW.balanceOfToken(address(this), quote) returns (
      uint256 credit) { if (credit == 0) return 0; } catch {}
6 +     }
7         uint256 before = _held();

```

Developer Response

Accepted. One correction for the record: the brief does not present `FeeClaimFailed` as a signal. The event records a caught revert. However the point about gas is correct, and holders pay it on every redeem between sweeps which is wasteful.

Fixed in `ufgfinancialdefi/ufg-audit@af05aa3` (tag `audit-2026-10-fixes`). `_claimFromPons` reads the router's escrow credit first and returns when it is zero. Both calls run inside `try`, so an escrow outage still lets `claim()` deliver what the router holds. The mock escrow reverts with `NoBalance()` on an empty claim, as the live one does. Tests cover the empty claim, a logged failure, and a reverting balance view.

2.9.6 UFG's share of volume is above the stated 0.02% during Pons's snipe-tax window

Pons adds its snipe tax to the creator fee bucket, and the router takes its fixed fraction of everything it claims. UFG's share of a buy in a coin's first seconds is therefore far above the 0.02% of volume that the brief states, up to about 41 times in the launch second.

Technical Details

The live `PonsV2BondingCurve` ([verified source](#), deployed by the launch deployer) charges a decaying snipe tax on buys by non-exempt recipients right after launch. The live terms are `snipeTaxStartBps = 9_900` and `snipeTaxSeconds = 3`. The tax is `startBps >> ((elapsed * 14) / window) (currentSnipeTaxBps, L315)`, capped per buy at `10_000 - feeBps - creatorTaxBps - 100` (L468). The curve adds it to the base fee bucket with `_accrueFees(fee + snipeTax, tax)` (L506), so 70% of it is credited to the creator-fee recipient, which is the router. The router charges its fixed fraction on everything it claims ([UFGLaunchFeeRouter.sol#L114](#)).

For a standard coin (100 bps fee, 30% Pons share, 100 bps creator tax, fraction 20,000 / 1,700,000), UFG's take per buy is:

Seconds after launch	Snipe tax	UFG's take of the buy	Against the 2 bps target
0 (launch second)	9,700 bps (capped)	81.88 bps	40.9×
1	618 bps	7.09 bps	3.5×
2	19 bps	2.16 bps	1.08×
3 and later	0	2.00 bps	1×

Pons's owner sets both terms for new launches with `setSnipeTaxStartBps` and `setSnipeTaxSeconds`, up to `snipeTaxStartBps = 9_900` and `snipeTaxSeconds = 60` (`MAX_SNIPE_TAX_SECONDS`), and each curve keeps the values of its launch. With a 60-second window, UFG's take stays above $3.5\times$ the target for the first 22 seconds: $21\times$ at 5 s, $11\times$ at 10 s and $6\times$ at 15 s.

The brief states that UFG's share "is a fixed rate of the trading volume. The target is 0.02%", and section 6.1 describes the rate on the curve as exact. Neither holds inside the snipe window. The brief's reading of Pons is based on commit `fb7b593` of `ponsdotdev/ponsfamily` (now `pons-labs`), whose `curve` has no snipe tax. Only the deployed curve, [verified on Sourcify](#), charges it.

Impact

Informational. Only non-exempt buys in the first seconds after a launch deviate. With today's 3-second window only the launch second deviates materially; with the 60-second maximum, about the first 20 seconds would. UFG takes about 1.2% of the creator's snipe windfall from the coin's treasury, the same proportion as for ordinary creator fees, but this contradicts the stated volume rate and the README's example figures.

Recommendation

State in section 6 that UFG's share is a fixed fraction of creator fees, $1/85$ for the standard terms. Add that it equals 0.02% of volume only outside the snipe window, and only when the curve fee equals the pool fee (see [yAudit/ufg-yagent-scan#1](#)). The router cannot tell the snipe part of a claim apart from ordinary fees, so documentation is the practical fix.

Developer Response

Accepted as a documentation finding. We checked the live deployer bundle on Sourcify and the factory's live snipe terms. The figures match. UFG's fraction of creator fees stays exact inside the window, so there is no code change.

Addressed in `ufgfinancialdefi/ufg-audit@c2a890e` (tag `audit-2026-10-fixes`). Section 6 states UFG's share as a fixed fraction of creator fees, $1/85$ for the standard terms, which equals 0.02% of volume only under the standard fees. A new section 6.2 lists the exceptions: the snipe tax with the live terms and the $41\times$ figure, the pool fee, and price impact. Section 8 pins the brief to the live Sourcify-verified contracts by address and notes that the live curve differs from commit `fb7b593`.

2.9.7 UFGLaunchTreasury.redeem accepts the treasury itself as recipient

`UFGLaunchTreasury.redeem()` accepts the treasury itself as the recipient. The caller's coins are burned and the payout stays in the treasury.

Technical Details

`redeem()` only rejects the zero address:

```
1 if (to == address(0)) revert BadRecipient();
```

With `to == address(this)`, the coins are burned to `DEAD`, and then:

- for a coin quoted in ETH, `to.call{value: net}("")` sends the ETH back to the treasury, whose `receive()` accepts it;
- for a coin quoted in an ERC-20, `safeTransfer(address(this), net)` is a transfer to itself.

The reserve doesn't change and the effective supply drops, so the redeemer's share becomes a gift to the other holders.

Treasury v2 fixed the same issue after a private review. Its fixed findings list "The recipient of a payment could be v2", and [UFGTreasuryV2.sol:191](#) now rejects `to == address(this)`. `UFGLaunchTreasury` was derived from v1 and doesn't include that fix.

Impact

Informational. Only a caller who passes the treasury's address loses funds, and only their own. No other party is harmed. The check is still inconsistent with v2, where the same case was treated as a finding.

Recommendation

Reject the treasury as recipient, as v2 does:

```
1 - if (to == address(0)) revert BadRecipient();
2 + if (to == address(0) || to == address(this)) revert BadRecipient(); // a
   payout to the treasury itself would be a gift to the treasury
```

Developer Response

Accepted.

Fixed in `ufgfinancialdefi/ufg-audit@235eef0` (tag `audit-2026-10-fixes`).

`UFGLaunchTreasury.redeem` rejects the treasury itself and its own router as recipient with `BadRecipient`, as v2 does. The brief lists the check as D10 among the differences from v1.



UFG Factory and Treasuries

Completed 2026-10-07