code4rena

Sequence

Smart Contract Security Assessment

Audit dates: Nov 11 — Nov 17, 2025

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Sequence: Transaction Rails smart contract system. The audit took place from November 11 to November 17, 2025.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 0 unique vulnerabilities with a risk rating in the categories of HIGH and MEDIUM severity.

Additionally, C4 analysis included 31 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Sequence team.

Scope

The code under review can be found within the C4 Sequence: Transaction Rails repository, and is composed of 9 smart contracts written in the Solidity programming language and includes 444 lines of Solidity code.

The code in C4's Sequence: Transaction Rails repository was pulled from:

- Repository: https://github.com/0xsequence/trails-contracts
- Commit hash: `b9ff2e7dd79ce3ed75172fa1b42bea88218c0ca6`

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges

- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Low Risk and Non-Critical Issues

For this audit, 31 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xBug_X** received the top score from the judge.

The following wardens also submitted reports: 0xki, 0xshdax, Agontuk, Ahmerdrarerh, ameng, Ashitosh, aster, billyrg131, Diavolo, excel, freebird0323, HalfBloodPrince, home1344, kind0dev, KKKKK, l3gb, lioblaze, luckygru, pfapostol, pranaygaurav1234567, rayss, redfox, rfa, richa, Sathish9098, Silverwind, TOSHI, undefined_joe, yaractf, and zcai.

[01] Router shim + router integration lacks coverage for the `allowFailure` invariant

Vulnerability details

`TrailsRouterShim.handleSequenceDelegateCall` forwards arbitrary payloads to `ROUTER`. Only `TrailsRouter._validateRouterCall` enforces `allowFailure = false`. Current tests wire the shim to mock routers, so there is no regression test proving that a Sequence wallet using the shim cannot bypass the `allowFailure` restriction by forwarding rogue selectors to the real router.

Attack path: deploy shim, point it at a genuine router, craft a payload with an invalid selector (or `allowFailure = true`), and rely on the absence of an end-to-end test to hide regressions that might someday skip `_validateRouterCall`.

Impact

A future refactor that accidentally removes `_validateRouterCall` (or weakens it) could go unnoticed, allowing wallets to execute `aggregate3Value` calls with `allowFailure = true`, undermining router invariants.

Recommended mitigation steps

Add Forge tests that deploy a real `TrailsRouter`, connect it through the shim, and assert that attempts to send `allowFailure = true` (or wrong selectors) revert, proving the invariant across both contracts.

[02] Chain-fork domain separator behavior is untested and easily misunderstood

Vulnerability details

`DOMAIN_SEPARATOR` hard-codes `block.chainid` at deployment, while `_verifyAndMarkIntent` re-reads `chainid()` when hashing the intent. On a chain fork, a signature prepared for pre-fork `chainId` may still pass (or fail) depending on which side updated `chainId`. No regression test demonstrates that signing for chain X and executing on chain Y correctly fails.

Attack path: sign an intent on a forked testnet, replay it post-fork where `DOMAIN_SEPARATOR` differs but the runtime `chainid()` inside the struct matches, leading to confusing acceptance/rejection patterns.

Impact

Developers may incorrectly assume cross-chain or fork scenarios are impossible, potentially exposing multi-chain deployments to replay risk or operational confusion.

Recommended mitigation steps

Document the behavior and add a negative test that signs with one `chainId` using `vm.chainId` then executes under another, confirming rejection. Consider recomputing `DOMAIN_SEPARATOR` on-demand using `block.chainid` for clarity.

[03] Fee commitments can be bypassed by setting `feeCollector = address(0)`

Vulnerability details

Both `depositToIntent` and `depositToIntentWithPermit` accept non-zero `feeAmount` values but only transfer the fee if `feeCollector != address(0)`.

Attack path: a relayer convinces a user to sign an intent specifying `feeAmount > 0`, yet sets `feeCollector = address(0)`. On-chain execution succeeds, no fee is transferred, and there is no event proving non-payment. The relayer can still claim off-chain that a fee was paid because the signature payload included it.

Impact

Users or downstream services relying on signed parameters may believe a fee was settled even though it was silently skipped, enabling dishonest relayers to obtain free service.

Recommended mitigation steps

Revert when `feeAmount > 0` but `feeCollector == address(0)`, or automatically enforce `feeCollector != address(0)` inside the signature schema so off-chain commitments match on-chain enforcement.

[04] Unchecked `amount + feeAmount` allows 256-bit overflow in permit flow

Vulnerability details

`depositToIntentWithPermit` verifies `permitAmount == amount + feeAmount` within an `unchecked` block. An attacker can choose `feeAmount` close to `2^256 - 1`, causing the addition to wrap and match a much smaller `permitAmount`.

Attack path: craft an intent with `amount = X`, `feeAmount = type(uint256).max - X + 1`, and `permitAmount = 1`. The unchecked addition wraps to 1, so `PermitAmountMismatch` is not raised, yet only 1 token needs to be permitted.

Impact

Relayers can bypass the requirement that `permitAmount` covers both deposit and fee, potentially causing under-approved transfers or execution failures later in the flow.

Recommended mitigation steps

Perform the addition in checked arithmetic (`if (feeAmount > type(uint256).max - amount) revert Overflow();`) before comparing to `permitAmount`.

[05] Intent state mutates before permit succeeds, causing irreversible nonce burns

Vulnerability details

`depositToIntentWithPermit` calls `_verifyAndMarkIntent` before invoking `IERC20Permit.permit`. `_verifyAndMarkIntent` marks the digest as used and increments `nonces[user]`.

Attack path: a relayer submits the transaction with an outdated permit signature so the `permit` call reverts. The revert happens **after** the nonce increment, so the user's nonce advances and the digest remains marked, effectively burning their signature without transferring funds.

Impact

Users can be griefed: a single failed permit (due to front-running, incorrect nonce, or token behavior) permanently consumes their intent, forcing them to re-sign.

Recommended mitigation steps

Reorder operations so that the permit call executes before `_verifyAndMarkIntent`, or add a reentrancy-safe pattern where nonces and `usedIntents` are updated only after all external calls succeed.

[06] `execute()` accepts stray ETH and leaves it stuck in the router

Vulnerability details

`TrailsRouter.execute` is payable but does nothing with `msg.value`. When called standalone, any attached ETH remains on the router, where it can later be swept by whoever controls subsequent router calls.

Attack path: an end user mistakenly sends ETH along with `execute`; their funds now sit on the router contract rather than being returned, awaiting anyone else's sweep.

Impact

Accidental deposits become communal funds that other callers can drain via later multicall sweeps.

Recommended mitigation steps

Revert when `msg.value != 0` for `execute`, or immediately refund `msg.value` to `msg.sender`.

[07] `_forwardToRouter` lacks calldata/value sanity checks

Vulnerability details

`TrailsRouterShim._forwardToRouter` blindly decodes `(bytes inner, uint256 callValue)` and forwards `callValue` ETH to `ROUTER`. There is no validation that:

1. `data.length >= 64`,
2. `callValue <= address(this).balance` in delegate-call context,
3. `inner` targets a known router selector.

Attack path: supply malformed calldata to force abi decode panics, or request more ETH than the wallet holds, causing deep-stack reverts and leaving the success sentinel unset. Worse, pointing `ROUTER` to a malicious contract allows arbitrary ETH drains because there is no selector whitelist.

Impact

DoS potential for Sequence wallets (reverts leave state half-updated) and configuration risks where ETH is forwarded to unintended targets.

Recommended mitigation steps

Validate the payload length before decoding, ensure `callValue` cannot exceed the wallet's balance, and optionally restrict `inner` to known router selectors or require `ROUTER == msg.sender`.

[08] `usedIntents` mapping never blocks replays (dead logic)

Vulnerability details

`_verifyAndMarkIntent` first checks `nonce == nonces[user]`, then increments `nonces[user]++`, and only afterward can any attempt reuse the same digest. Because the nonce moves forward, replays will always fail at the `InvalidNonce` check **before** `usedIntents` is consulted.

Attack path: Replay the exact same digest—the transaction reverts for `InvalidNonce`, not `IntentAlreadyUsed`, proving the mapping is redundant.

Impact

`usedIntents` consumes storage and gas without adding protection, giving a false sense of layered defence. If nonce logic is ever relaxed (parallel lanes), replays would succeed because the unused mapping was never tested meaningfully.

Recommended mitigation steps

Remove `usedIntents` or redesign intent identification so digest-based replay protection is meaningful. Update tests/docs accordingly.

[09] Leftover router balances are globally stealable

Vulnerability details

`pullAndExecute` / `pullAmountAndExecute` pull assets into the router and delegate-call `Multicall3`. If any subcall refunds unused ETH or ERC-20 to `msg.sender`, the funds end up on the router contract. There is no per-user accounting.

Attack path: Victim calls `pullAmountAndExecute` where a target refunds 1 ETH back to `msg.sender` (the router). Later, an attacker calls public `execute` with an `aggregate3Value` payload sending `value` from the router to themselves, draining the leftover.

Impact

Any funds unintentionally left on the router after another user's session can be stolen by the first arbitrary caller—this is an immediate loss of user funds.

Recommended mitigation steps

After each execution, sweep residual balances back to the original caller, or maintain per-caller escrow that must be explicitly withdrawn. Alternatively, restrict public entry points once the router is meant only for delegate-call contexts.

[10] Persistent ERC-20 allowances let targets drain wallets later

Vulnerability details

`_injectAndExecuteCall` uses `SafeERC20.forceApprove` to grant `target` an allowance equal to the wallet's full balance but never revokes it after the call.

Attack path: a benign-looking contract receives the allowance, performs the intended action, then later (or through compromise) calls `transferFrom(wallet, attacker, allowance)` to steal all tokens.

Impact

Sequence wallets delegating through the router can lose their entire token balances long after the approved call completed.

Recommended mitigation steps

Always reset the allowance to zero in a finally-like section, even when the target call reverts. Safer still, transfer tokens directly into the target instead of using approvals where possible.

[11] `refundAndSweep` is re-entrancy prone and can over-refund

Vulnerability details

`refundAndSweep` sends the refund before updating any state or performing the sweep and lacks a re-entrancy guard.

Attack path: a refund recipient with a fallback can re-enter `refundAndSweep`, repeatedly claiming `refundAmount` before the sweep leg executes, draining the wallet and starving the intended sweep recipient.

Impact

An attacker designated as refund recipient can extract more than the capped refund and prevent the sweep recipient from receiving funds.

Recommended mitigation steps

Add `nonReentrant` protection (or use Checks-Effects-Interactions). Alternatively, perform the sweep first or store internal state indicating the refund has been processed before making external calls.

[12] Router lacks global re-entrancy defences for ERC777-style callbacks

Vulnerability details

Functions like `pullAmountAndExecute`, `injectSweepAndCall`, `sweep`, etc., can interact with ERC777 tokens that invoke hooks on `msg.sender`. While running inside a Sequence wallet via `delegatecall`, a malicious token can re-enter router functions such as `sweep` or `refundAndSweep` before the original call finishes.

Attack path: attacker deploys ERC777 token with `tokensToSend` hook that re-enters `sweep` and drains unrelated assets while the router still controls the wallet storage.

Impact

Complete loss of wallet assets during token transfers triggered through the router.

Recommended mitigation steps

Introduce a re-entrancy guard across router entry points or switch to pull-style mechanics where the router does not transfer tokens until after state is updated.

[13] Special native-asset ERC-20 addresses can bypass ETH-specific logic

Vulnerability details

Several chains expose the native token via a well-known ERC20 contract (e.g., CELO, Polygon's `0x…1010`, zkSync's `0x…800A`). The router assumes `address(0)` is the only native asset indicator.

Attack path: on such chains, an attacker can pass the wrapped-native address as an ERC20, causing the router to treat native coins as if they were ordinary tokens, enabling drain vectors similar to prior CELO incidents.

Impact

Misconfigured deployments on affected chains could enable token drains or mis-accounting for native assets.

Recommended mitigation steps

Add chain-specific guards disallowing known native-wrapper addresses unless explicitly expected, or generalize the native token abstraction so wrappers are handled safely.

[14] Sentinel storage desynchronizes when `_tstoreSupport` flips

Vulnerability details

`TrailsRouterShim` writes the success sentinel via `_setTstorish`, which chooses either `sstore` or `tstore` based on `_tstoreSupport`. If the mode toggles between write and read (e.g., `TstoreMode` helper flips the flag), `validateOpHashAndSweep` later reads from the opposite medium and reverts with `SuccessSentinelNotSet()` despite a prior success.

Attack path: execute a routed call while `_tstoreSupport == false` (sentinel stored via `sstore`), then before sweeping, enable `tstore`. The sweep now fails, locking funds.

Impact

Funds can become temporarily inaccessible (DoS) because the sentinel appears missing even though the operation succeeded.

Recommended mitigation steps

Write the sentinel to both storage types or record which medium was used alongside the sentinel so the reader can fetch from the correct location. Clear sentinels after use to avoid stale data.

[15] Sentinels are replayable across arbitrary tokens/recipients

Vulnerability details

The sentinel only notes that “some call with `opHash` succeeded”; it does not bind to `token`, `recipient`, or amount.

Attack path: after one successful operation sets the sentinel, an attacker reuses the same `opHash` to invoke `validateOpHashAndSweep` for different tokens or recipients, draining unrelated assets long after the original operation.

Impact

One successful call grants indefinite sweeping rights for any asset associated with that wallet/`opHash` pair, leading to unauthorized fund transfers.

Recommended mitigation steps

Associate the sentinel with the specific token/recipient (and optionally amount) used in the original call, and clear it immediately after a sweep so it cannot be replayed.

[16] `depositToIntentWithPermit` rejects non-EIP-2612 tokens like DAI

Vulnerability details

The function casts every token to `IERC20Permit` and invokes the standard `permit`. Tokens like DAI use a legacy signature (`permit(address holder, address spender, uint nonce, uint expiry, bool allowed, …)`); calling the EIP-2612 version always reverts.

Attack path: users attempting to deposit DAI with permit will see their transactions fail, making the advertised UX inaccessible.

Impact

All tokens with non-standard permit interfaces (DAI, CHAI, some stablecoins) cannot use the permit-based deposit path, reducing compatibility.

Recommended mitigation steps

Introduce adapters or allow callers to specify the permit selector/encoding. Alternatively, maintain an allowlist of tokens known to support EIP-2612 and revert early for others with a clear message.

[17] `forceApprove` breaks tokens that forbid zero-allowance resets

Vulnerability details

`SafeERC20.forceApprove` sets allowance to zero before setting the desired value. Some ERC20s (e.g., USDT variants, bespoke governance tokens) revert when `approve(spender, 0)` is called while a non-zero allowance exists.

Attack path: the first `injectAndCall` succeeds; subsequent calls revert during the zero-reset, permanently DoS'ing those tokens.

Impact

Router functionality becomes unusable for zero-intolerant tokens after the first call.

Recommended mitigation steps

Detect whether the current allowance already equals the desired amount and skip the reset. For problematic tokens, use `safeIncreaseAllowance` / `safeDecreaseAllowance` patterns or transfer tokens directly instead of relying on allowances.

[18] Router tests never exercise return-data forwarding due to `onlyDelegatecall` revert

Vulnerability details

`testForwardToRouterReturnValue` invokes `handleSequenceDelegateCall` directly, causing it to revert in `onlyDelegatecall` before `_forwardToRouter` is executed. Thus, no test currently checks that router return data is propagated back to Sequence wallets.

Attack path: regressions that drop return data (or corrupt it) would pass tests because `_forwardToRouter` is never reached in coverage.

Impact

Potential future bugs in return-data handling could slip through CI unnoticed.

Recommended mitigation steps

Update the test to invoke the shim through a `delegatecall` (e.g., using a proxy wallet as other tests do), asserting that `_forwardToRouter` executes and its return data bubbles up.

[19] Unused balance helper functions can mask future inconsistencies

Vulnerability details

Functions `_nativeBalance()`, `_erc20Balance()`, and `_erc20BalanceOf()` are never invoked anywhere in the router. Because they are dead code, any future contributor might start using them under the assumption they are “canonical” helpers, yet there is no test coverage anchoring their behavior.

Attack path: is developmental rather than runtime. A later feature could rely on these helpers and subtly diverge from `_getBalance` semantics (e.g., forgetting delegate-call nuances), shipping bugs that tests never catch because the helpers themselves were never exercised.

Impact

Dead code increases maintenance risk and widens the surface for silent behavioral drift in future changes.

Recommended mitigation steps

Remove these helpers, or add unit tests that exercise them alongside `_getBalance`, ensuring they cannot diverge unnoticed if reused later.

Disclosures

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit

submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.