yAudit

May 2026

Prepared for
VFAT

Audited by
fedebianu
adriro

Sickle Bridge Strategies Update 2

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

This protocol extends Sickle with cross-chain bridge flows. On the source chain, `BridgeWithdrawStrategy` packages farm harvest/withdraw actions and then delegates bridging to `BridgeLib` inside the user's Sickle. On the destination chain, bridge callbacks land in adapter contracts that either swap bridged tokens through `MultiSwapRouter` or forward them into `BridgeDepositStrategy` for farm/NFT deposits.

1.2 Audit Scope

This audit covers 48 smart contracts totaling approximately 6000 lines of code across 6 days of review. The initial scope was defined on commit `b2e947a`, and was later expanded to include the Aave Adapter in commit `afaef46`.

```
contracts
├── BridgeRouter.sol
├── SwapRouter.sol
├── bridges
│   ├── AcrossV3AmountAdapter.sol
│   ├── BridgeRouteReceiver.sol
│   ├── BridgeSwapReceiver.sol
│   ├── CcipDepositSender.sol
│   ├── CcipRouteSender.sol
│   ├── CcipSenderBase.sol
│   ├── CcipSwapSender.sol
│   └── adapters
│       ├── AcrossSwapAdapter.sol
│       ├── BridgeAdapterBase.sol
│       ├── CcipDepositAdapter.sol
│       ├── CcipRouteAdapter.sol
│       ├── CcipSwapAdapter.sol
│       ├── DeBridgeSwapAdapter.sol
│       ├── StargateSwapAdapter.sol
│       ├── WormholeCctpDepositAdapter.sol
│       ├── WormholeDepositAdapter.sol
│       ├── WormholeRouteAdapter.sol
│       └── WormholeSwapAdapter.sol
├── dex
│   ├── DexAdapterBase.sol
│   ├── IDexAdapter.sol
│   ├── ISwapRouter.sol
│   ├── SwapTypes.sol
│   └── adapters
│       ├── AaveV3Adapter.sol
│       ├── AerodromeRouterAdapter.sol
│       ├── AlgebraPoolAdapter.sol
│       ├── AlgebraRouterAdapter.sol
```

```
│       ├── BlackholeV2Adapter.sol
│       ├── CamelotV2Adapter.sol
│       ├── PancakeInfinityAdapter.sol
│       ├── SlipstreamRouterAdapter.sol
│       ├── SolidlyAdapter.sol
│       ├── UniswapV2Adapter.sol
│       ├── UniswapV3PoolAdapter.sol
│       ├── UniswapV3Router02Adapter.sol
│       ├── UniswapV3RouterAdapter.sol
│       ├── UniswapV4Adapter.sol
│       ├── VelodromeUniversalRouterAdapter.sol
│       └── WrapAdapter.sol
├── interfaces
│   ├── IBridgeAmountAdapter.sol
│   ├── external
│   │   └── across
│   │       └── IAcrossV3SpokePool.sol
│   └── libraries
│       └── IExactTransferLib.sol
├── libraries
│   ├── BridgeLib.sol
│   └── ExactTransferLib.sol
├── security
│   └── RouterAllowlist.sol
└── strategies
    └── BridgeDepositStrategy.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

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

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	4
Low	10
Informational	10

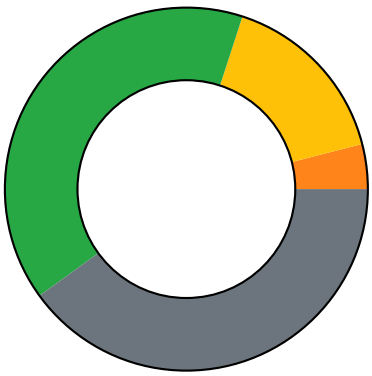

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The update materially expands the previously reviewed bridge surface. The design remains modular and uses explicit allowlists for routers, bridge targets, source senders, adapters, and strategy execution. The main residual risks are integration-boundary issues: balance-delta accounting, callback authentication, route forwarding collisions, native gas-drop handling, destination fallback semantics, and operator recovery of stranded bridge or sender balances.

2 Audit Overview

2.1 Project Information

Protocol Name: VFAT
Repository: https://github.com/vfat-io/sickle-contracts
Commit URLs:

- Initial commit: b2e947ae078452c56f1c2dd18fe790c7e2a7339f
- AaveV3Adapter: afaef467b480aa3b4085c604666303410418545d

2.2 Audit Timeline

The audit was conducted from May 20 to 26, 2026.

2.3 Audit Resources

- Code repositories and documentation

Category	Mark	Description
Access Control	Good	The audited contracts use layered allowlists and role checks across routers, bridge targets, CCIP senders, adapters, receivers, and signed strategy execution. Correctness still depends on consistent per-chain configuration.
Mathematics	Good	The code is not math-heavy. The relevant logic is balance-delta accounting, split amounts, fee quotes, refunds, gas-drop amounts, exact transfers, and minimum-output checks.
Complexity	Average	The update adds a broader bridge and swap surface. Modules are readable, but end-to-end flows span senders, adapters, receivers, routers, DEX adapters, fallback paths, and strategy deposits.
Libraries	Good	The repository relies on established external components such as Solmate, OpenZeppelin, LayerZero/Stargate interfaces, Wormhole/Across/deBridge integrations, and internal connector libraries. This gives a good base, although bridge-specific wrappers still need protocol-aware review.
Decentralization	Average	Operation remains centralized around admin-managed allowlists, bridge configuration, route policy, rescue paths, fee collectors, and gas-drop parameters.
Documentation	Average	Documentation and deployment context are useful, but some assumptions remain operational, including sender setup, router/factory allowlists, supported DEX payloads, fallback recipients, and recovery procedures.
Testing and verification	Good	The suite covers many bridge, receiver, router, adapter, and strategy flows, including failure paths. Some bridge behavior remains integration-specific and should be validated with configuration and fork review.

Table 2: Code Evaluation Matrix

2.4 Critical Findings

None.

2.5 High Findings

2.5.1 Cap-funded CCIP routes can redirect second-hop fee refunds and drain hub ETH

Technical Details

`BridgeRouteReceiver` allows a first-hop adapter with a nonzero `maxHubNativeFeeByAdapter` cap to spend hub-held native ETH when `params.nextHop.nativeValue` exceeds the `msg.value` delivered with the callback. The receiver only validates the next-hop bridge contract, selector, and amount offset before forwarding `nativeValue` into the opaque bridge call. It does not decode or overwrite bridge-specific native refund-recipient fields.
This is exploitable for allowlisted second-hop bridge selectors whose calldata includes an explicit native refund recipient, such as `send(SendParam,MessagingFee,address)` or `sendToken(SendParam,MessagingFee,address)`. An attacker can route through `CcipRouteSender` into `CcipRouteAdapter`, set `nextHop.nativeValue` within the hub cap, and encode the second-hop refund recipient as an attacker-controlled address.
`_checkHubNativeFeeCap()` authorizes the hub contribution, and `_callBridgeWithApprovals()` treats the route as successful as long as the receiver's balance decreases by the expected native amount. Any unused native fee refunded directly to the attacker bypasses `_refundExcessEth()`, because that function can only return ETH still held by the receiver.
The reachable source path is `CcipRouteSender.bridgeRoute()` -> `CcipRouteAdapter.ccipReceive()` -> `BridgeRouteReceiver.executeRoute()`.
The same adapter-wide sponsorship model also creates the related nested-hub risk described by ZeroCool Fix-review H-01 and Base L-06: if the allowlisted second-hop bridge is another CCIP route sender to another sponsored hub, each hub can apply its own local `maxHubNativeFeeByAdapter[CcipRouteAdapter]` cap to the same user-originated route. That variant is more topology-dependent, but it reinforces the same missing invariant: hub-native sponsorship is not bound to a specific source lane, route depth, bridge selector, or enforceable refund behavior.

Impact

High. A public caller can convert hub-funded native fee float into direct refunds to an external address whenever a capped route can invoke a second-hop bridge selector with attacker-controlled native refund fields. The loss is bounded per call by `maxHubNativeFeeByAdapter[CcipRouteAdapter]` and by the receiver's ETH balance, but the attack is repeatable and can deplete the shared hub-native reserve used by legitimate CCIP route continuations. Severity should scale with the configured cap, available hub ETH float, and whether allowlisted second-hop selectors expose external native refund recipients.

Recommendation

Only allow hub-funded native routes for bridge selectors whose native refund recipient is fixed to the receiver, or parse and overwrite the refund-recipient field to `address(this)` before the

call. For opaque selectors whose refund behavior cannot be enforced, require the caller to source-fund the full native value and keep the hub cap at zero.
As a defense-in-depth measure, make hub-native sponsorship provenance-aware and hop-aware. Cap sponsorship by authenticated source lane and sender, or deploy isolated adapters per sponsored lane. For multi-hub CCIP routes, include an authenticated route-depth budget and reject routes that would consume multiple hub subsidies unless the user explicitly funds each hop.

Developer Response

Fixed in #827 @ 42a08461.
`RouteSelectorPolicy.refundPolicy` half + corresponding patch step in `_validateAndPatchCalldata`. When any portion of `nextHop.nativeValue` comes from hub float, the `(bridge, selector)` pair must be admin-registered with one of: `0` (reject), `HUB_REFUND_IMPLICIT` (bridge refunds `msg.sender`, no patching), or an offset (receiver overwrites that word with `address(this)` before the call).

- Struct field: `contracts/bridges/BridgeRouteReceiver.sol:127`.
- Sentinel: `:192`.
- Admin setter: `:278`.
- Refund-policy gate + offset patch: `:738` onwards.

2.6 Medium Findings

2.6.1 CCIP route sender's fee-token collision guard is ineffective across chains

Technical Details

`CcipRouteSender._requireForwardTokenIndependentOfNextHop()` attempts to reject routes where the forwarded fee token collides with the next-hop bridge token:

```
if (forwardToken == params.nextHop.bridgeToken) {
    revert ForwardTokenConflictsWithNextHopBridgeToken();
}
```

The comparison is made on the source chain. `forwardToken` is a source-domain token address, while `params.nextHop.bridgeToken` is the destination hub token address. For CCIP token transfers, the delivered destination token can differ from the source token address. A source-side raw address comparison can therefore miss a collision that only exists after CCIP maps the token to its destination representation.
The destination `BridgeRouteReceiver.executeRoute()` performs the real same-chain collision check in `_requireFeeTokenIndependent()` before the protected `try this.routeExternal(...)` wrapper. A collision therefore reverts the whole `CcipRouteAdapter.ccipReceive()` call instead of entering the route fallback path.

Impact

Medium. Users can receive a successful source quote and send a route that is guaranteed to fail on destination due to token-domain collision. Because CCIP token execution is atomic, a receiver revert also reverts the attempted token delivery, and the message only becomes eligible

for manual execution. Manual execution retries the same deterministic payload, so it will hit the same pre-catch destination check again.
The user principal and forwarded fee token are therefore effectively unrecoverable through the Sickle contracts for that message. There is no failed-message handler in `CcipRouteAdapter`, the receiver is immutable, and the route cannot fall back cleanly. This is a permanent-failure path for a source-validated route, so Medium is more appropriate than Low.

Recommendation

Validate the source-to-destination token mapping before sending, or move the collision validation into a destination path that is catchable and can fall back cleanly.

Developer Response

Fixed in vfat-io/sickle-contracts#851, commit ab9655bb.

- Split the destination check: `feeToken == token` still hard-reverts before baselines.
- Moved `feeToken == params.nextHop.bridgeToken` into `routeExternal()` before bridge execution, so it is caught by `executeRoute()` and falls back cleanly.
- Added `test_executeRoute_fallsBack_when_feeToken_equals_bridge_token()`.

2.6.2 Public `getOrDeploySickle` lets attackers preseed another user's Sickle automation approval

Technical Details

`StrategyModule.getOrDeploySickle(owner, approved, referralCode)` is declared `public`, so every strategy that inherits it (`FarmStrategy`, `NftFarmStrategy`, lending strategies, `BridgeDepositStrategy`) exposes this function as an external entrypoint. When an unprivileged caller invokes this function through one of the inheriting strategies, the call forwards to `SickleFactory.getOrDeploy()`.
`SickleFactory.getOrDeploy()` enforces only `registry.isWhitelistedCaller(msg.sender)`, where `msg.sender` is the strategy contract itself. The factory fully trusts the `(admin, approved, referralCode)` tuple supplied by the whitelisted caller and does not validate that the external EOA has any relationship to `admin`. When `_getSickle(admin)` finds no existing Sickle, `_deploy()` creates a new clone with the caller-supplied `approved` value set during initialization in `Sickle.initialize()` and `SickleStorage._initializeSickleStorage()`.
Once the Sickle exists, subsequent calls to `getOrDeploy()` with different `approved` or `referralCode` arguments return the existing Sickle without modifying the stored `approved` address. This allows an attacker to front-run a victim's first interaction with the protocol by calling `FarmStrategy.getOrDeploySickle(victim, attacker, attackerReferral)`, which deploys the victim's deterministic Sickle with `owner = victim` and `approved = attacker`. When the victim later interacts with any strategy that resolves the Sickle via `getOrDeploySickle(victim, intendedApproved, intendedReferral)`, the factory returns the already-deployed Sickle and ignores the victim's intended `approved` value.
The `onlyApproved(sickle)` modifier enforces `sickle.approved() == msg.sender` on automation entrypoints such as `FarmStrategy.harvestFor()`, `FarmStrategy.compoundFor()`, `FarmStrategy.exitFor()`, and the corresponding NFT strategy functions

`NftFarmStrategy.harvestFor()`, `NftFarmStrategy.compoundFor()`, `NftFarmStrategy.exitFor()`, and `NftFarmStrategy.rebalanceFor()`. Once the victim has configured automation settings for a position, the attacker-controlled `approved` address can invoke these entrypoints on the victim's Sickle, triggering harvest, compound, exit, or rebalance actions with caller-chosen parameters that are validated only against coarse automation flags and trigger conditions in the settings registries, rather than being bound to user-approved swap or zap details.

Impact

Medium. An attacker can predeploy a victim's deterministic Sickle with `approved = attacker` before the victim interacts with the protocol. When the victim later uses any strategy (including `deposit`, direct-to-Sickle bridge flows, or manual completion paths), the victim's intended `approved` automator is silently ignored and the attacker retains the automation role. Once the victim enables automation settings for a position, the attacker can unilaterally trigger `harvestFor`, `compoundFor`, `exitFor`, or `rebalanceFor` on farm and NFT positions. These actions charge protocol fees to the victim, may impose unwanted exits or slippage at poorly timed moments, and can race or grief legitimate automation. While funds are ultimately returned to the victim's Sickle owner address through the helper libraries, the victim's strategy behavior is no longer under their exclusive control and the attacker can force fee-bearing automation actions without user consent.

Recommendation

Change `StrategyModule.getOrDeploySickle()` from `public` to `internal` so arbitrary external callers cannot relay through whitelisted strategy contracts to invoke `SickleFactory.getOrDeploy()` with attacker-chosen `approved` values. Internal uses within strategies will continue to function correctly.
If a public deployment helper is required for specific workflows, restrict it to require `owner == msg.sender` or validate a user signature authorizing the full `(owner, approved, referralCode)` tuple before forwarding to the factory. Ensure that later first-use flows do not silently inherit an attacker-seeded `approved` address by checking whether the caller is the owner before accepting an existing Sickle or by allowing the owner to reset the `approved` address during their first legitimate interaction.

Developer Response

Fixed in #827 @ 42a08461.
`StrategyModule.getOrDeploySickle` changed from `public` to `internal`. No longer reachable as an external entrypoint on inheriting strategies; the factory can't be relayed through a whitelisted strategy with attacker-supplied `(owner, approved, referralCode)`. Internal uses across `FarmStrategy`, `NftFarmStrategy`, `BridgeDepositStrategy`, etc. continue via inheritance.

- `contracts/modules/StrategyModule.sol:38`.

2.6.3 `SwapRouter` residual balances can be stranded and later drained through pool callbacks

Technical Details

`SwapRouter` can retain residual token balances during successful swaps. The router pulls the full input amount, dispatches execution to the selected adapter, and accepts the step based on the observed `tokenOut` balance delta in `SwapRouter._dispatchAdapter()`. It does not also verify that `tokenIn` or intermediate token balances were fully consumed or refunded.
This matters for bridge receiver flows such as `BridgeSwapReceiver.swapExternal()` and `BridgeRouteReceiver._swapToBridgeToken()`, which approve the bridged input to `SwapRouter` and finalize the route when the output-side minimum checks pass. If an adapter or downstream router under-consumes the input while still producing enough output, the leftover input can remain in `SwapRouter` instead of being returned to the recipient.
`VelodromeUniversalRouterAdapter` makes this path explicit by appending a `SWEEP` command that sends unconsumed input back to `SwapRouter`.
Those residual balances are not only recoverable by admin-style `sweep()` logic. The public pool callbacks in `SwapRouter.uniswapV3SwapCallback()` and `SwapRouter.algebraSwapCallback()` are not bound to `_activeAdapter` or to an active swap context. They validate that `msg.sender` is a legitimate pool, decode an arbitrary payer from callback data, and transfer from `SwapRouter` whenever the decoded payer is `address(this)`.
An attacker can therefore initiate a swap directly against a valid allowlisted V3, Slipstream, or Algebra pool with the attacker as recipient and callback data naming `SwapRouter` as payer. During the callback, the pool is valid, the payer is `SwapRouter`, and any compatible residual balance held by `SwapRouter` can be paid into the pool, converting the stuck balance into output for the attacker.
Attack path:

1. Execute a bridge receiver swap or route through an adapter/downstream router that under-consumes `tokenIn` while still satisfying the required output minimum.
2. Observe that the unconsumed input remains in `SwapRouter`; for `VelodromeUniversalRouterAdapter`, the trailing `SWEEP` sends unused input to `SwapRouter`.
3. From an attacker account, call a valid allowlisted V3, Slipstream, or Algebra pool directly, setting the attacker as swap recipient.
4. Encode callback data with `payer == address(SwapRouter)`.
5. The pool calls back into `SwapRouter`; the callback validates the pool and transfers the owed token from `SwapRouter` to the pool.
6. The attacker receives the pool output, and the residual router balance is drained.

The callback path can also be reproduced from any manually prefunded or stale compatible token balance in `SwapRouter`.

Impact

Medium. Users can lose part of a bridged or swapped input even though the destination swap or route succeeds, because residual input is stranded in `SwapRouter`. Once stranded, any third party can drain compatible balances through the unauthenticated pool-callback path before the protocol recovers them with `sweep()`. This does not allow theft from normal fully-consuming atomic swaps, but it turns realistic residual or stale router balances into externally extractable value.

Recommendation

Account for residual balances at the router level. Snapshot `tokenIn` and relevant intermediate token balances before dispatching an adapter, then either refund unexpected leftovers to the recipient or fallback recipient in the same transaction, or revert when leftovers are not expected.
Bind pool callbacks to an active swap context. Before invoking direct V3, Slipstream, or Algebra pool swaps, store the expected pool, token to pay, payer, and maximum amount. In each callback, require that the callback matches the stored context, consume the expected amount, and clear the context after the adapter call. Do not allow arbitrary external pool calls to make `SwapRouter` pay solely because callback data names it as payer.
Avoid using `SwapRouter` as the terminal recipient for downstream router sweeps unless the swept balance is immediately accounted for and refunded in the same call.

Developer Response

Fixed in #827 @ 42a08461.

1. **Pool callback arming** — closes the drain side.
2. **TokenIn residue refund** — `_swap` / `swapSplit` snapshot the tokenIn balance pre-pull. Post-swap residue above the snapshot refunds to the recipient; consuming pre-existing residue reverts `ResidualInputConsumed`.

- `_settleTokenInResidual`: `contracts/SwapRouter.sol:274`.
- `_swap` snapshot + settle: `:219,259`.
- `swapSplit` same: `:301,346`.
- New error: `:125`.

2.6.4 Stargate compose messages remain replayable after rescue and can consume later same-token balances

Technical Details

The Stargate adapters authenticate compose delivery, but they do not bind execution to a per-message balance or retire a queued compose when operators rescue funds after a failed callback.
`ILayerZeroComposer.lzCompose()` receives a `_guid`, but the Stargate adapter implementations ignore it. `StargateAdapterBase._authorizeCompose()` checks only that the caller is a whitelisted endpoint, `_from` is a whitelisted Stargate pool, and the decoded `srcEid` is allowed. It then reads `amountLD` and the user payload from the message, without marking the compose `_guid` or message hash as consumed.
If a compose delivery reverts after Stargate has delivered tokens to the adapter, the tokens remain parked on the adapter and the LayerZero compose remains replayable. Admins can remove those parked balances through `BridgeAdapterBase.rescueTokens()` or `rescueETH()`, but rescue does not invalidate the still-queued compose.
On replay, the adapters only check the aggregate current balance. `StargateRouteAdapter`, `StargateSwapAdapter`, and `StargateDepositAdapter` all require or wrap up to `amountLD`, then call `_forwardAvailable()`, which forwards from the adapter's current token balance rather than from a message-local escrow. Therefore, after the original parked funds are rescued, an old compose can later be replayed against unrelated same-token funds that become parked on the same adapter.
The stale payload controls the downstream action. In the route flow, `BridgeRouteReceiver.executeRoute()` can fall back to `params.fallbackRecipient` if the

replayed route parameters revert. In the swap flow, `BridgeSwapReceiver._swapOrFallback()` falls back to the payload recipient when the swap reverts. In the deposit flow, `StargateDepositAdapter.lzCompose()` forwards the replayed balance to `BridgeDepositStrategy.executeStargateDeposit()`, where the old signed payload can consume the later balance if the signed Stargate intent is still valid.
This requires an operational sequence: a Stargate compose must fail and leave funds parked, operators must rescue the parked balance without invalidating the queued compose, and later same-token funds must become parked on the same adapter. That makes the likelihood low, but the stale compose remains a live claim on future aggregate balances after rescue.

Impact

Medium. Later users' parked same-token balances can be consumed by a stale Stargate compose payload. Depending on the adapter type, the replay can force fallback to an old recipient, run an old route, execute an old swap payload, or deposit the later balance through an old Stargate deposit intent. The amount is capped by the stale compose's `amountLD`, but it can equal the full balance parked by a later failed delivery.

Recommendation

Do not treat admin rescue as independent from queued compose state. Add a way to invalidate a compose `_guid` or message hash before or during rescue, and make each Stargate adapter reject invalidated compose messages. Operationally, rescue should only happen after the related compose is replayed successfully, permanently blocked, or explicitly invalidated on-chain.
For a stronger design, avoid aggregate-balance replay entirely by tracking per-compose accounting: record the expected token and amount for each compose hash when funds are parked, consume only that record on replay, and clear or invalidate it when funds are rescued.

Developer Response

Fixed in #827 @ 42a08461.
`StargateAdapterBase` consumes the LZ compose `_guid` irreversibly in the outer `lzCompose`, then dispatches the inner work via `try this.lzComposeExternal()` so a downstream revert leaves the guid consumed and emits `ComposeProcessingFailed`. LayerZero retry cannot replay against later same-token balances.

- Storage / errors / events: `contracts/bridges/adapters/StargateAdapterBase.sol:28,48,79`.
- Outer try/catch: `:112`.
- `_authorizeCompose` consume-before-process: `:251-281`.
- Admin pre-rescue: `invalidateComposeGuid`.

2.7 Low Findings

2.7.1 `BridgeRouter` strands refunded source-token residue from `swapAndBridge()` swaps

Technical Details

`SwapRouter` refunds unconsumed first-input ERC20 residue to the `recipient` passed to `swap()` / `swapWithFee()`. That is correct for direct user swaps, but bridge flows use a technical recipient so the bridge contract can custody the output before forwarding it.
Two bridge paths share the same root issue:

1. `BridgeRouter._executeSourceSwap()` calls `SwapRouter.swapWithFee()` with `recipient = address(this)` during `swapAndBridge()`. After the swap, `_executeBridgeAndRefund()` reconciles the bridge token and ETH, but not the original source token when it differs from the bridge token.
2. `BridgeRouteReceiver._swapToBridgeToken()` approves the route input token and calls `msr.swap()` with `recipient = address(this)` so it can custody the next-hop bridge token. On successful route continuation, the receiver refunds excess ETH and fee tokens, but it does not reconcile the original swap input token.

In both cases, if `SwapRouter._settleTokenInResidual()` returns unconsumed input to the `recipient`, the residue lands on the technical bridge contract instead of the end user or route fallback recipient.

Impact

Low. Successful swap-and-bridge flows can leave unconsumed source or hub-route input tokens in `BridgeRouter` or `BridgeRouteReceiver`. The user receives the bridge output, but any refunded input residue is silently retained by the technical recipient and requires admin rescue or sweep. The currently registered swap adapters are exact-input oriented and generally consume the provided input or revert. The practical live source of residue is narrow, primarily router-style sweep behavior such as the Velodrome Universal Router path, so expected residue is likely small and operational recovery is available.

Recommendation

Use a residual recipient that is distinct from the swap output recipient, or snapshot and reconcile the source input token around the MSR call in each bridge caller.
For `BridgeRouter`, refund post-swap source-token surplus above the pre-swap baseline to `context.recipient`.
For `BridgeRouteReceiver`, refund post-swap route-token surplus above the pre-swap baseline to `params.fallbackRecipient`.

Developer Response

Fixed in PR#851, commit ab9655bb.

- `BridgeRouter._executeSourceSwap()` snapshots the source-token baseline and refunds post-MSR surplus to the bridge `recipient` when `tokenIn != bridgeToken`.
- `BridgeRouteReceiver._swapToBridgeToken()` does the same to `params.fallbackRecipient` when route input differs from next-hop bridge token.
- Added regression tests for both paths.

2.7.2 `SwapRouter` can strand cyclic or later-hop ERC20 residual inputs

Technical Details

`SwapRouter._swap()` snapshots only `steps[0].tokenIn` before pulling input and calls `_settleTokenInResidual()` once after output transfer. `swapSplit()` mirrors the same pattern for the common split input token.
The residual settlement is incomplete for ERC20 multi-hop routes because `_settleTokenInResidual()`:

```
1 if (tokenIn == tokenOut) return;
```

This creates two variants of the same root issue:

1. **Cyclic routes can hide first-token residue.** In an `A -> B -> A` route, refunded `A` from an earlier hop can be included in the later hop's `balanceBefore(A)`. The final output delta can exclude that earlier residue, and settlement then skips because `tokenIn == tokenOut`.
2. **Later-hop input residues are not settled.** In an `A -> B -> C` route, only `A` is settled. If a later hop receives `B` and a router-style adapter such as Velodrome Universal Router sweeps unused `B` back to `SwapRouter`, the intermediate residue remains in `SwapRouter`.

Impact

Low. Successful swaps can strand user-attributable ERC20 assets in `SwapRouter`. The funds are not stolen by another user during the swap, but they are no longer returned to the intended recipient and are only recoverable through admin sweep logic. `sweep()` sends funds to the fee collector, so recovery is operational rather than automatic. Each variant requires a route shape or downstream refund behavior that is not the normal exact-input happy path, and there is no attacker profit path by itself.

Recommendation

Under the currently registered adapters the only live residue source is the Velodrome Universal Router adapter's trailing `SWEEP` (direct-pool adapters revert on partial fill and V4 / Infinity settle the full input). A full per-call ledger over every hop token is disproportionate to a Low, self-inflicted, admin-recoverable issue.

- **Cyclic and later-hop variants:** fix at the source — have the Velodrome adapter forward its sweep residue to the recipient or revert on leftover like the pool adapters. One local change, no per-swap cost.
- **Optional defense-in-depth:** add a `SwapRouter`-level per-call ledger only if partial-fill-tolerant adapters are ever registered via `setDexAdapter`. The cyclic case must settle before `_applyFeeAndTransfer`, since with `tokenIn == tokenOut` the balance delta conflates residue, fee, and output.

Developer Response

Fixed in PR#851, commit ab9655bb.

- Cyclic first-token residue is settled because `_settleTokenInResidual()` no longer skips `tokenIn == tokenOut`.
- `VelodromeUniversalRouterAdapter` now reverts with `ResidualInputReturned` if the trailing sweep returns input to `SwapRouter`, closing later-hop residue fail-closed.
- Added first-hop and later-hop Velodrome returned-input tests.

2.7.3 Reentrant gas-drop callbacks can create phantom reserved refunds and DoS native-fee route continuations

Technical Details

`BridgeRouteReceiver.executeRoute()` and `executeRouteWithNativeGasDropToken()` snapshot `preCallEthBalance` before entering a protected external self-call. Inside that route execution, primary-token gas drops use `_reserveNativeGasDrop()`, while second-token gas drops use `_sendNativeGasDropToken()`. Both paths can unwrap WETH and send native ETH to `params.fallbackRecipient` with all available gas.
Most inbound route adapters are endpoint-gated, which limits reachability. However, the reentry path is not impossible:

- `WormholeRouteAdapter.executeVAAv1()` is externally callable and relies on Wormhole VAA verification rather than `msg.sender` gating.
- `AggLayerRouteAdapter.executePendingRoute()` is `public payable` once a pending route exists.

A malicious `fallbackRecipient` that has a valid nested route available can reenter through one of those reachable adapters during the native gas-drop callback.
If the nested route records a failed ETH refund, `_refundExcessEth()` increments `reservedRefundEth`. The outer route still uses its older ETH baseline and can later treat the nested ETH as excess and successfully refund it. Successful ETH refunds do not decrement `reservedRefundEth`, so the reservation can become larger than the receiver's actual ETH balance.

Impact

Low. The receiver can end with `reservedRefundEth` over-accounted relative to its ETH balance. Later routes with `callValue > 0` can revert with `WouldConsumeReservedRefund`, including source-funded native-value continuations, not only hub-funded ones. Exploitation requires a valid nested route, a user-controlled gas-drop recipient, and a failed nested ETH refund. The impact is route liveness/accounting disruption, not direct theft. Operators can repair the accounting through `rescueETH`, although a phantom reservation larger than the actual receiver balance may require replenishing ETH before reducing the reservation.

Recommendation

Add a reentrancy guard around `executeRoute()` and `executeRouteWithNativeGasDropToken()`, or make failed-refund reservations scoped and decrementable so a later successful refund cannot leave phantom reserved balance.

Developer Response

Fixed in PR#851, commit ab9655bb.

- `BridgeRouteReceiver` now inherits solmate `ReentrancyGuard`.
- Added `nonReentrant` to `executeRoute()` and `executeRouteWithNativeGasDropToken()`; self-call wrappers remain unguarded.
- Added `test_executeRoute_nativeGasDrop_reentrant_callback_blocked()`.

2.7.4 `AlgebraRouterAdapter` does not enforce the configured custom deployer allowlist

Technical Details

`AlgebraRouterAdapter` supports Algebra Integral swaps by decoding a user-supplied `deployer` from `SwapStep.dexData`. In `AlgebraRouterAdapter._swap()`, any 32-byte `dexData` value is decoded as an address and forwarded as the `deployer` argument to `IAlgebraIntegralSwapRouter.exactInputSingle`. The local interface models that field directly in `ExactInputSingleParams`.
When `SwapRouter._executeStep()` processes `DexType.Algebra`, it checks only that `step.router` is allowlisted. It does not know which Algebra factory or custom deployer the selected router will use for the swap. `RouterAllowlist` nevertheless has explicit custom deployer controls through `requireAllowedCustomDeployer()`, `setCustomDeployer()`, and the deployment-plan sync path in `updateAllowedCustomDeployers()`.
This creates an inconsistency between Algebra execution paths. Direct Algebra pool swaps validate both the factory and custom deployer before calling the pool; `AlgebraPoolAdapter._isAlgebraPoolFromFactory()` calls `requireAllowedCustomDeployer(factory, customDeployer)` for non-standard custom pools. Router-style Algebra swaps skip that control and rely only on the external router being allowlisted.
The `deployer` value is chosen by whoever constructs the `SwapStep`, usually the user or an off-chain route builder. A third party cannot alter it after the transaction or bridge-deposit payload is signed, but a compromised or incorrect route builder can route through a custom Algebra deployer that governance never approved, as long as the allowlisted external Algebra router accepts that deployer.

Impact

Low. Governance cannot enforce the configured custom deployer allowlist for Algebra Integral swaps executed through `AlgebraRouterAdapter`.

Recommendation

For Algebra Integral mode, derive or configure the relevant factory for the selected router and call `allowlist.requireAllowedCustomDeployer(factory, deployer)` before forwarding the swap. If the router's factory cannot be read reliably, reject nonzero `deployer` values in the router adapter and require custom-pool routes to use `AlgebraPoolAdapter`, where the factory and custom deployer can be validated against the actual pool.

Developer Response

Fixed in #827 @ 42a08461.
`AlgebraRouterAdapter` rejects nonzero `deployer` (auditor's stated fallback when the router factory can't be derived on-chain). Custom-deployer routes must use `AlgebraPoolAdapter`, which validates factory + deployer against the actual pool.

- Reject: `contracts/dex/adapters/AlgebraRouterAdapter.sol:46`.
- Error `CustomDeployerMustUsePoolAdapter: :26`.

Standard Algebra Integral pools (`deployer == address(0)`) still work through the router adapter.

2.7.5 `nativeGasDropToken` conflicts can strand fallback funds in BridgeRouteReceiver

`BridgeRouteReceiver.executeRouteWithNativeGasDropToken()` documents that `nativeGasDropToken` must differ from both the route token and the fee token, but this invariant is only enforced inside the external self-call. Because the outer function catches that revert and continues through fallback accounting, conflicting token configurations can leave part of the delivered funds stuck in `BridgeRouteReceiver`.

Technical Details

`executeRouteWithNativeGasDropToken()` records independent refund baselines for `feeToken` and `nativeGasDropToken`, then calls `routeWithNativeGasDropTokenExternal()` in a `try/catch`. The same-token check for the gas-drop token is performed later in `_sendNativeGasDropToken()`:

```
if (nativeGasDropToken == token || nativeGasDropToken == feeToken) {
    revert NativeGasDropTokenConflicts();
}
```

That revert does not reject the call at the top level. It is caught by `executeRouteWithNativeGasDropToken()`, which then runs the fallback path and refunds each token bucket according to the baselines captured before the conflict was checked.
When `nativeGasDropToken == token`, the fallback first transfers up to `amount` of the route token to `params.fallbackRecipient`. The later gas-drop-token refund uses a baseline that includes the route principal, so the delivered gas-drop balance can remain in the receiver instead of being returned. When `nativeGasDropToken == feeToken`, the fee-token and gas-drop-token refund baselines overlap. Depending on the relative fee and gas-drop amounts, the second refund can treat part of the delivered balance as pre-existing and leave it behind.

Impact

Low. Invalid route configurations that violate `BridgeRouteReceiver`'s own token-independence invariant can strand delivered tokens in the receiver until an admin rescues them. Users receive fallback for the primary route token in the common case, but the separately delivered gas-drop or fee-token funds can be retained by the contract.

Recommendation

Validate the `nativeGasDropToken` independence invariant in `executeRouteWithNativeGasDropToken()` before computing refund baselines and before entering the caught self-call. If `nativeGasDrop > 0`, revert when `nativeGasDropToken == token` or `nativeGasDropToken == feeToken`.

Developer Response

Fixed in #827 @ 42a08461.
The `nativeGasDropToken == token` / `== feeToken` check moved from inside the caught self-call to the outer entrypoint `executeRouteWithNativeGasDropToken`, before any refund baselines are computed. Conflicts revert up front instead of catching in the outer try and stranding the gas-drop balance.

- `contracts/bridges/BridgeRouteReceiver.sol:441-442`.

2.7.6 CCIP route sender accepts unsupported fee and bridge tokens collision routes that revert before protocol fallback

Technical Details

The `CcipRouteSender` contract accepts dual-token route shapes where the forwarded second token matches the next-hop bridge token, even though this configuration is guaranteed to fail on the destination chain before the receiver's fallback logic executes.
In `CcipSenderBase._requireValidForwardToken()`, the source-side validation rejects only three cases: zero-address forwards, duplicates of the first bridged token, and duplicates of the CCIP fee token. The function does not validate whether the forwarded token will collide with `params.nextHop.bridgeToken` on the destination side.
When a caller submits a route with `forwardAmount > 0`, `params.nativeGasDrop == 0`, and `forwardToken == params.nextHop.bridgeToken`, the source chain accepts the message and quotes a valid fee through `getFee()`. The CCIP message is sent to the hub chain.
On arrival, `CcipRouteAdapter._forwardAndExecute()` decodes the second delivered token as `feeToken` when `params.nativeGasDrop == 0` and calls `receiver.executeRoute()` with that value. Inside `BridgeRouteReceiver.executeRoute()`, the first action is a call to `_requireFeeTokenIndependent()`, which explicitly rejects the case where `feeToken == params.nextHop.bridgeToken`. This validation occurs before the receiver enters its protected `try this.routeExternal(...)` wrapper that provides fallback-to-`fallbackRecipient` behavior. The revert bubbles out of `ccipReceive()`, bypassing the normal protocol fallback path and leaving the message in a failed-delivery state that requires manual intervention.
The validation gap is subtle: the source chain enforces independence between the forwarded token and the first bridged token, but does not enforce independence between the forwarded token and the second-hop bridge token. This mismatch allows quotes and sends that encode destination-guaranteed failures.

Impact

Low. A caller can pay the source-side CCIP send fee for a dual-token route that is guaranteed to fail on the destination before the receiver's normal fallback logic executes. The concrete protocol-level harm is wasted source-side bridging fees and loss of the advertised automatic fallback-to-`fallbackRecipient` behavior for that route. Ultimate principal recovery then depends on CCIP failed-message handling outside this codebase.
The finding does not demonstrate theft of protocol-held funds, protocol-wide denial of service, or forced harm to uninvolved users. The main affected party is the caller who submits the malformed route parameters. The exact custody state of principal after the destination revert depends on CCIP off-ramp semantics that are outside the provided code, so the verified outcome is loss of automatic fallback and entry into external failed-message or manual-recovery handling, not permanent principal loss.

Recommendation

Mirror the destination invariant on the source side. In `CcipRouteSender`, reject fee-forwarding dual-token routes where the forwarded token equals `params.nextHop.bridgeToken`. Enforce this check in both `getFee()` and `bridgeRoute()` so quoted routes cannot encode a destination-guaranteed failure.
Add a validation helper that checks the fee-forwarding shape:

```
function _requireValidFeeForwardingRoute(
    address forwardToken,
    uint256 forwardAmount,
    BridgeRouteReceiver.RouteParams calldata params
) internal pure {
    if (forwardAmount > 0 && params.nativeGasDrop == 0) {
        if (forwardToken == params.nextHop.bridgeToken) {
            revert FeeTokenConflictsWithBridgeToken();
        }
    }
}
```

Call this helper in both `getFee()` and `bridgeRoute()` after the existing `_validateRoute()` check. This preserves the existing fee-forwarding feature while preventing source-side acceptance of routes that cannot complete or safely fall back on the destination side.

Developer Response

Fixed in #827 @ 42a08461.
`CcipRouteSender._requireForwardTokenIndependentOfNextHop` called from both `getFee` and `bridgeRoute`. Mirrors the destination invariant on the source side; dual-token fee-forwarding routes whose `forwardToken == params.nextHop.bridgeToken` can't quote/send.

- Helper: `contracts/bridges/CcipRouteSender.sol:192`.
- Call sites: `:79`, `:112`.
- Error `ForwardTokenConflictsWithNextHopBridgeToken`: `:34`.

Skips when `forwardToken == token` so existing `_requireValidForwardToken` (`DuplicateForwardToken`) fires first.

2.7.7 `AerodromeRouterAdapter` does not enforce the configured factory allowlist

Technical Details

The `AerodromeRouterAdapter` accepts a user-supplied factory address without validating it against the protocol's `RouterAllowlist`. When `SwapRouter._executeStep()` processes a step with `DexType.AerodromeRouter`, it enforces `allowlist.requireAllowed(step.router)` but performs no factory validation. The adapter then decodes `(bool stable, address factory)` from `step.dexData` and embeds that factory directly into the `IAerodromeRouter.Route` struct before calling the external Aerodrome router.
While `RouterAllowlist` exposes `requireAllowedFactory()`, the adapter never invokes it. This contrasts with other factory-aware adapters such as `UniswapV3PoolAdapter` and `AlgebraPoolAdapter`, which validate factories on-chain. The fork tests demonstrate that Aerodrome swaps succeed after allowlisting only `Base.AERODROME_ROUTER`, with no separate approval of `Base.AERODROME_FACTORY`.
The factory value is determined by whoever constructs the `SwapStep`, typically the user or an off-chain route builder. No third party can modify it after the transaction is signed, but a user relying on a compromised route builder or submitting malicious calldata can select any factory that the external Aerodrome router accepts via its own `factoryRegistry()`, even if that factory was never approved or was later removed from `RouterAllowlist`.

Impact

Low. Governance cannot independently approve or disable specific Aerodrome-compatible factories through the protocol's `RouterAllowlist`. Swaps can execute through venues that were never added to the allowlist or were explicitly removed, bypassing the intended on-chain venue-governance boundary. However, swaps still go through an allowlisted router, and the caller's `minAmountOut` check provides slippage protection against obvious underdelivery. The concrete harm is loss of venue-provenance enforcement rather than immediate fund loss.

Recommendation

Decode the `factory` parameter in `AerodromeRouterAdapter._swap()` and call `allowlist.requireAllowedFactory(factory)` before building the route and invoking the external router. Add regression tests showing that Aerodrome and Velodrome router swaps revert unless both the router and the selected factory are allowlisted. This restores the protocol's ability to gate venue selection on-chain and aligns the Aerodrome adapter with the factory-validation pattern used by other adapters.

Developer Response

Fixed in #827 @ 42a08461.
`AerodromeRouterAdapter._swap` decodes `(stable, factory)` and calls `allowlist.requireAllowedFactory(factory)` before the external router call. Matches `UniswapV3PoolAdapter` / `AlgebraPoolAdapter`.

- `contracts/dex/adapters/AerodromeRouterAdapter.sol:36`.

2.7.8 Permissionless Wormhole route redemption can force hub-chain fallback

Technical Details

`WormholeRouteAdapter.executeVAAv1()` is permissionless. It calls `_redeemTransfer()`, which completes the Wormhole transfer and consumes the one-shot VAA before the route is validated or executed.
After redemption, the adapter forwards tokens and caller-supplied `msg.value` into `BridgeRouteReceiver.executeRoute()`. The receiver catches any revert from the route self-call and finalizes fallback to `params.fallbackRecipient` on the hub chain. A third-party redeemer can therefore submit the VAA first with insufficient `msg.value` for `params.nextHop.nativeValue`, or wait until route conditions such as `swapDeadline` are stale, causing the route to fall back after the VAA has already been consumed.
Because Wormhole VAAs are single-use, the legitimate relayer cannot later retry the original route with correct native funding or better timing. The user's funds are not stolen, but the automatic continuation is irreversibly collapsed into the degraded hub-chain fallback outcome.

Impact

Low. An arbitrary caller can censor a Wormhole route continuation at low cost. The principal is sent to the fallback recipient, but the user loses destination delivery and must manually re-route, paying additional fees and delay.

Recommendation

Pre-parse and validate route preconditions before calling `completeTransferWithPayload` wherever possible, especially native fee sufficiency and expired swap deadlines. For robust recovery, store redeemed funds in a retriable pending state instead of converting every post-redemption route failure into final fallback.

Developer Response

Fixed in #827 @ 42a08461.
`WormholeAdapterBase._peekTransferPayload` decodes the VAA's transfer payload without redeeming. `WormholeRouteAdapter.executeVAAv1` pre-validates native fee sufficiency and swap deadline before `_redeemTransfer`, so a permissionless caller can't burn the VAA into a hub-chain fallback.

- Peek: `contracts/bridges/adapters/WormholeAdapterBase.sol:85`.
- Native-fee guard: `contracts/bridges/adapters/WormholeRouteAdapter.sol:36-44`.
- Deadline guard: `:45-48`.
- Errors `InsufficientNativeForNextHop`, `SwapDeadlineExpired`: `:11-12`.

2.7.9 Failed `msg.value` refunds are not segregated from hub-native fee float

Technical Details

The `BridgeRouteReceiver` contract maintains an ETH balance that serves as working capital for second-hop routes whose adapters have been granted a nonzero `maxHubNativeFeeByAdapter` cap. When a route completes with excess ETH above the pre-call baseline, `_refundExcessEth()` attempts to send that ETH back to the `fallbackRecipient`. If the

recipient cannot accept ETH (for example, a non-payable contract), the function emits `RefundFailed` and leaves the ETH on the receiver, with the expectation that administrators will later recover it via `rescueETH()`.
The issue arises because failed refunds are not tracked separately from the receiver's general ETH balance. When a subsequent route arrives through an adapter with a nonzero `maxHubNativeFeeByAdapter` setting, `_checkHubNativeFeeCap()` authorizes spending from the receiver's entire ETH balance whenever `msg.value` is less than `nextHop.nativeValue`. The contract does not distinguish between ETH that was explicitly deposited as hub float and ETH that represents a prior user's failed refund awaiting recovery.
In `executeRoute()` and `executeRouteWithNativeGasDropToken()`, the receiver records `preCallEthBalance = address(this).balance - msg.value` as the baseline for refund calculations. After the route completes, `_refundExcessEth(params.fallbackRecipient, preCallEthBalance)` attempts to return any ETH above that baseline. When the refund send fails, no storage reserves the stranded amount for the intended recipient. A later route with `msg.value = 0` and `nextHop.nativeValue` set within the per-adapter cap can then consume that ETH as its own `hubContribution` in `_callBridgeWithApprovals()`.
The CCIP adapter flow demonstrates the intended use case. `CcipRouteAdapter.ccipReceive()` is non-payable, so CCIP-shaped second hops rely on the receiver's existing ETH balance to fund `nextHop.nativeValue`. The protocol configures a nonzero `maxHubNativeFeeByAdapter` for `CcipRouteAdapter` to enable this pattern. However, any ETH left by a failed refund becomes immediately available for these subsequent routes, even though the contract documentation specifies that failed refunds should be recovered through `rescueETH()`.

Impact

Low. When a user's ETH refund fails and remains on the receiver, that ETH can be consumed by a later unrelated route before administrators perform the intended recovery. The affected user loses their excess native value, which typically represents overpaid fees or leftover native gas after a successful bridge call. The loss is limited to the per-adapter cap configured for the later route, but repeated routes can drain the entire stranded balance. The contract's monitoring and recovery assumptions break down because a `RefundFailed` event does not guarantee that the ETH remains available for rescue operations.

Recommendation

Maintain a dedicated accounting variable to track ETH that is reserved for failed refunds and exclude it from the pool available for hub-native fee spending. When `_refundExcessEth()` fails, increment a `reservedRefundEth` accumulator by the refund amount. In `_checkHubNativeFeeCap()` and bridge spending logic, compute the spendable hub float as `address(this).balance - reservedRefundEth` and enforce that `hubContribution` does not exceed this spendable amount. Decrement the reserve only when administrators successfully recover the ETH to the intended recipient or when a retry mechanism delivers it to the original `fallbackRecipient`.
If maintaining per-recipient refund tracking is preferred, use a mapping to record pending refunds and allow recipients or administrators to explicitly claim them. This approach provides finer-grained recovery control and prevents cross-user balance mixing. Alternatively, if operational complexity is a concern, consider reverting the route when an ETH refund fails rather than emitting `RefundFailed`, though this is a stricter measure that may impact user experience for users with non-payable fallback recipients.

Developer Response

Fixed in #827 @ 42a08461.
`BridgeRouteReceiver.reservedRefundEth` accumulates ETH that should have refunded but couldn't (non-payable recipient). Reserved bucket is excluded from spendable hub float.

- Storage: `contracts/bridges/BridgeRouteReceiver.sol:199`.
- Failed-refund increment: `:990`.
- Outbound spend guard `WouldConsumeReservedRefund`: `:861`.
- `rescueETH` decrement: `:320`.

2.7.10 `emergencyDisableAdapter()` is admin-only and cannot be triggered by a guardian

`SwapRouter.emergencyDisableAdapter()` is restricted to `admin` only, even though the protocol elsewhere exposes a `guardian` role designed to disable functionality without involving the slower admin key.

Technical Details

`emergencyDisableAdapter()` clears the adapter slot for a given `dexType`, after which any swap step targeting that type reverts with `AdapterNotSet`. The function is gated by `if (msg.sender != admin) revert NotAdmin();`, so only `admin` can pull the kill switch. `GuardianAdmin` already defines an `onlyGuardianOrAdmin` modifier and is used elsewhere in the codebase so a faster, lower-trust guardian can disable functionality in an incident. `SwapRouter` does not inherit that role for this entry point, which means that when a faulty or compromised adapter must be disabled, the only path is through the admin key. This is inconsistent with the emergency framing of the function name and with the guardian pattern used in other contracts.

Impact

Low. In an incident requiring rapid adapter shutdown, response is bottlenecked on the admin key. The guardian role exists precisely to act faster than admin under these conditions, so excluding it widens the window during which a faulty or compromised adapter remains live.

Recommendation

Allow the guardian to call `emergencyDisableAdapter()` in addition to the admin, for example by inheriting `GuardianAdmin` and gating the function with `onlyGuardianOrAdmin`. Keep adapter registration (`setDexAdapter()`) admin-only so the guardian can only clear, not replace, adapter slots.

Developer Response

Fixed in #827 @ 42a08461.

- `SwapRouter.guardian` slot + `setGuardian`: `contracts/SwapRouter.sol:66,509`.
- `emergencyDisableAdapter` now `admin || guardian`: `:497`.
- `setDexAdapter` stays admin-only — guardian can only clear.
- New error `NotGuardianOrAdmin`: `:126`.

2.8 Gas Savings Findings

2.8.1 Duplicate `nativeGasDrop` conflict check in `routeWithNativeGasDropTokenExternal()`

`routeWithNativeGasDropTokenExternal()` re-checks the `nativeGasDrop` versus `params.nativeGasDrop` conflict that `executeRouteWithNativeGasDropToken()` already enforced before invoking the self-call.

Technical Details

`executeRouteWithNativeGasDropToken()` validates the gas drop conflict before dispatching the self-call:

```
if (nativeGasDrop > 0 && params.nativeGasDrop > 0) {
    revert NativeGasDropTokenConflicts();
}
```

It then invokes `this.routeWithNativeGasDropTokenExternal(...)`, which is gated by `if (msg.sender != address(this)) revert OnlySelf();`, so the only caller of this function is the upstream entry point that already enforced the same check. The duplicate validation inside `routeWithNativeGasDropTokenExternal()` therefore costs gas on every successful route without changing the reachable set of inputs.

Impact

Gas Savings.

Recommendation

Remove the duplicate `if (nativeGasDrop > 0 && params.nativeGasDrop > 0)` check from `routeWithNativeGasDropTokenExternal()` and rely on the upstream check in `executeRouteWithNativeGasDropToken()`.

Developer Response

Fixed in #827 @ 42a08461.
Removed the duplicate `(nativeGasDrop > 0 && params.nativeGasDrop > 0)` check from `routeWithNativeGasDropTokenExternal`. The only caller (`executeRouteWithNativeGasDropToken` via `OnlySelf` self-call) already performs it upstream.

- Inner: `contracts/bridges/BridgeRouteReceiver.sol:554`.
- Upstream check + self-call site: `:441,453`.

2.9 Informational Findings

2.9.1 `swapSplit()` empty first route panics before validation

Technical Details

`SwapRouter.swapSplit()` validates that `routes.length != 0` and that `routes.length == amounts.length`, but then immediately reads `routes[0][0].tokenIn` and `routes[0][routes[0].length - 1].tokenOut`. The intended empty-route validation exists in `_validateAndSumAmounts()`, but that function is called only after the first route has already been indexed. As a result, `swapSplit()` with `routes[0].length == 0` reverts with a Solidity array panic instead of the router's `InvalidStepsLength()` error.

Impact

Informational.

Recommendation

Move the `routes[0].length == 0` check before reading `routes[0][0]` and `routes[0][routes[0].length - 1]`. Keep the per-route validation in `_validateAndSumAmounts()` for subsequent routes.

Developer Response

Fixed in #827 @ 42a08461.
Empty-first-route check before indexing `routes[0][0]` in `SwapRouter.swapSplit`. Reverts with typed `InvalidStepsLength` instead of array OOB panic. Per-route validation in `_validateAndSumAmounts` still handles subsequent routes.

- `contracts/SwapRouter.sol:316`.

2.9.2 Bridge callback whitelists accept non-contract addresses

Technical Details

The local bridge callback whitelists do not require newly-enabled addresses to contain contract code. This applies to bridge endpoints in `BridgeAdapterBase._setWhitelistedBridgeEndpoint()`, swap adapters in `BridgeSwapReceiver._setWhitelistedAdapter()`, route adapters in `BridgeRouteReceiver._setWhitelistedAdapter()`, and deposit adapters in `BridgeDepositStrategy._setWhitelistedAdapter()`.
By contrast, `RouterAllowlist` rejects non-contract routers and bridge targets when enabling them. If an EOA or undeployed address is mistakenly whitelisted as a local callback adapter, that address can call receiver or strategy entrypoints directly.

Impact

Informational.

Recommendation

When enabling local callback endpoints or adapters, require `address.code.length > 0`. Keep this check scoped to same-chain executable endpoints; remote CCIP sender allowlists are cross-chain identities and should not use local code-length validation.

Developer Response

Fixed in #827 @ 42a08461.
`code.length > 0` checks when enabling local callback adapters / endpoints, mirroring `RouterAllowlist.setRouter` / `setBridge`.

- `BridgeAdapterBase._setWhitelistedBridgeEndpoint`: `contracts/bridges/adapters/BridgeAdapterBase.sol:53-54` (`EndpointNotAContract`).
- `BridgeRouteReceiver._setWhitelistedAdapter`: `:236-237` (`AdapterNotAContract`).
- `BridgeSwapReceiver._setWhitelistedAdapter`: `:100-101`.
- `BridgeDepositStrategy._setWhitelistedAdapter`: `:312-313`.

Test setUps updated with `vm.etch(addr, hex"01")` for placeholder router/endpoint addresses.

2.9.3 CCIP sender contracts accept direct transfers without a rescue path

Technical Details

Source-side CCIP sends go through `CcipSenderBase._bridgeMessage()`, used by `CcipDepositSender.bridgeDeposit()`, `CcipSwapSender.bridgeSwap()`, and `CcipRouteSender.bridgeRoute()`. The function refunds explicit native-fee overpayment, but the sender contracts also expose an unrestricted `receive()` and can receive ERC20 transfers directly.
Those pre-existing balances are not picked up by later sends.
`CcipSenderBase._pullAvailable()` snapshots the sender balance before pulling from the caller and bridges only the new balance delta. Unlike destination-side contracts such as `CcipAdapterBase.rescueTokens()`, `BridgeRouteReceiver.rescueTokens()`, and `BridgeRouteReceiver.rescueETH()`, `CcipSenderBase` exposes no admin rescue function for native value or ERC20s.

Impact

Informational.

Recommendation

Add admin-only `rescueTokens()` and `rescueETH()` functions to `CcipSenderBase`. If senders are intended to be strict pass-through contracts, remove the open `receive()` path unless router-originated refunds must be accepted.

Developer Response

Fixed in #827 @ 42a08461.
Admin-only `rescueTokens(token, to, amount)` and `rescueETH(to, amount)` on `CcipSenderBase`.

- `rescueTokens`: `contracts/bridges/CcipSenderBase.sol:313`.
- `rescueETH`: `:323`.

2.9.4 Redundant `feeToken == forwardToken` guards in `_approveRouter()` and `_clearRouterApproval()`

The router approval helpers re-check that `feeToken` differs from `forwardToken`, but this configuration is already rejected upstream by `_requireValidForwardToken()`, so the guards are dead code.

Technical Details

`_requireValidForwardToken()` enforces that when `forwardAmount > 0`, `forwardToken` is non-zero, distinct from `token`, and distinct from `feeToken`:

```
1 if (feeToken == forwardToken) revert DuplicateForwardToken();
```

This check runs in both `_getFeeForMessage()` and at the start of `_bridgeMessage()`, so any call path that reaches `_approveRouter()` and `_clearRouterApproval()` has already proven `feeToken != forwardToken` whenever `forwardAmount > 0`.
Despite this, `_approveRouter()` still gates the fee approval on `!(forwardAmount > 0 && feeToken == forwardToken)` and `_clearRouterApproval()` still gates the fee revocation on `feeToken != forwardToken`. These subexpressions can never be false at this point, so they add noise without protecting any state.

Impact

Informational.

Recommendation

Drop the `feeToken == forwardToken` subexpressions in `_approveRouter()` and `_clearRouterApproval()` and rely on the invariant established by `_requireValidForwardToken()`. The fee approval and revocation conditions then simplify to the `feeToken != address(0) && feeToken != token` checks.

Developer Response

Fixed in #827 @ 42a08461.
Removed the dead `!(forwardAmount > 0 && feeToken == forwardToken)` subexpression from `_approveRouter` and the `feeToken != forwardToken` subexpression from `_clearRouterApproval`. `_requireValidForwardToken` already rejects this case upstream whenever `forwardAmount > 0`.

- `_approveRouter`: `contracts/bridges/CcipSenderBase.sol:246`.
- `_clearRouterApproval`: `:279`.

2.9.5 `CcipSenderBase` duplicates `GuardianAdmin` logic instead of inheriting it

`CcipSenderBase` re-implements the same guardian, pause, and admin plumbing already provided by `GuardianAdmin`, adding maintenance overhead and a subtle behavioral divergence.

Technical Details

`CcipSenderBase` inherits from `Admin` and declares its own `guardian` storage, `GuardianSet` event, `NotGuardianOrAdmin` error, `setGuardian()`, and a `pause()` function that gates on `msg.sender != guardian && msg.sender != admin`. This is the same surface already exposed by `GuardianAdmin`, which centralizes `guardian`, `GuardianSet`, `setGuardian()`, and the `_onlyGuardianOrAdmin()` check used by every other CCIP and bridge contract.
Beyond duplication, the local versions differ in small but observable ways. `CcipSenderBase.setGuardian()` always emits and writes, while `GuardianAdmin.setGuardian()` short-circuits when `newGuardian == guardian`. The local error is `NotGuardianOrAdmin`, while `GuardianAdmin` uses `NotGuardianOrAdminError`. Off-chain consumers and integrations that key on the `GuardianAdmin` ABI must special-case `CcipSenderBase` descendants.

Impact

Informational.

Recommendation

Have `CcipSenderBase` inherit from `GuardianAdmin` instead of `Admin` and remove the duplicated `guardian` storage, `GuardianSet` event, `NotGuardianOrAdmin` error, and `setGuardian()`. Keep the pause flag local but route the access check through `_onlyGuardianOrAdmin()` so the contract uses the canonical guardian surface.

Developer Response

Fixed in #827 @ 42a08461.
`CcipSenderBase` inherits `GuardianAdmin` directly. Removed duplicated `guardian` storage, `GuardianSet` event, `NotGuardianOrAdmin` error, and `setGuardian`. `pause` uses canonical `onlyGuardianOrAdmin`.

- Inheritance: `contracts/bridges/CcipSenderBase.sol:19`.
- Constructor: `:47`.
- `pause`: `:56`.

2.9.6 CCIP adapters do not enforce unique delivered token entries

`CcipAdapterBase._forwardDelivered()` accounts delivered balances per token, so adapters that process two `destTokenAmounts` with the same ERC20 can strand the second amount instead of forwarding it to the receiver.

Technical Details

`_forwardDelivered()` computes delivery as the current adapter balance above `accountedTokenBalances[token]`, forwards up to `expectedAmount`, then resets `accountedTokenBalances[token]` to the adapter's remaining token balance. This works when each delivered token appears once in a CCIP message.
The CCIP deposit, swap, and route adapters each accept one or two delivered token entries and call `_forwardDelivered()` separately for the first and second entries. If both entries use the same token, the first call forwards the first amount and marks the leftover balance as accounted. The second call for the same token then sees no new delivery and returns zero, leaving the second amount in the adapter as accounted balance.
The canonical sender does validate this shape in `CcipSenderBase._requireValidForwardToken()` by rejecting `forwardToken == token`, so normal messages produced by the in-repository sender cannot hit this case. The destination adapters still rely on that source-side invariant and do not strictly validate uniqueness themselves. If a whitelisted sender is misconfigured, upgraded incorrectly, or otherwise sends duplicate token entries, the receiver can receive zero for the second token leg while funds remain in the adapter until admin recovery.

Impact

Informational.

Recommendation

Reject duplicate token addresses in the destination adapters when `message.destTokenAmounts.length == 2`, before calling `_forwardDelivered()` for either entry. This makes the receiver-side invariant explicit and keeps the adapter behavior correct even if a whitelisted sender violates the expected message shape.

Developer Response

Fixed in #827 @ 42a08461.
`CcipAdapterBase._requireDistinctTokens` called by all three CCIP destination adapters when `destTokenAmounts.length == 2`. A misconfigured source emitting the same token twice can't leave a leg stranded.

- Helper: `contracts/bridges/adapters/CcipAdapterBase.sol:197`.
- Error `DuplicateDeliveredToken: :30`.
- Call sites: `CcipRouteAdapter.sol:103`, `CcipSwapAdapter.sol:77`, `CcipDepositAdapter.sol:83`.

2.9.7 `nativeGasDropToken` fallback transfer is not surfaced in any event

`_bridgeDepositFallbackWithNativeGasDropToken()` transfers any available `nativeGasDropToken` to the sickle owner during a fallback but only emits `BridgeDepositFallback` for the main `token`, so off-chain consumers cannot observe the gas-drop refund.

Technical Details

When the bridge deposit fallback path runs with a separate `nativeGasDropToken`, the implementation transfers both the principal token and the gas drop token to `sickleOwner`:

```
uint256 fallbackAmount =
    _transferAvailable(token, sickleOwner, amount);
if (nativeGasDrop > 0) {
    _transferAvailable(
        nativeGasDropToken, sickleOwner, nativeGasDrop
    );
}
emit BridgeDepositFallback(sickleOwner, token, fallbackAmount);
```

The emitted `BridgeDepositFallback` event only carries `(sickleOwner, token, fallbackAmount)`. The `nativeGasDropToken` and the amount actually delivered to the user are not included in the event and no second event is emitted for the gas drop transfer. Indexers, refund reconciliations, and user-facing dashboards that rely on these events to display recovered funds will under-report the fallback in the gas-drop case.

Impact

Informational.

Recommendation

Either extend `BridgeDepositFallback` with `nativeGasDropToken` and the transferred gas drop amount, or emit a dedicated event for the gas drop refund alongside the existing one. Keep the event shape consistent with the equivalent fallback path in `BridgeSwapReceiver`.

Developer Response

Fixed in #827 @ 42a08461.
Dedicated `BridgeDepositGasDropFallback` event so off-chain indexers see the gas-drop refund leg separately from the principal-token `BridgeDepositFallback`.

- Declaration: `contracts/strategies/BridgeDepositStrategy.sol:100`.
- Emit inside `_bridgeDepositFallbackWithNativeGasDropToken`: `:617`.

2.9.8 `BridgeRouteReceiver` can route hub-funded native value into non-route selectors

Technical Details

`BridgeRouteReceiver` uses the shared `RouterAllowlist` for second-hop route calls. `_validateAndPatchCalldata()` checks that the bridge target is allowlisted, patches one in-bounds 32-byte word as the bridge amount, and then checks the selector. It does not know whether the selector is actually a route-compatible bridge entrypoint or whether the patched word is a principal amount field.
If a non-route selector such as Wormhole `Executor.requestExecution(...)` is allowlisted for another workflow, a route can call it through the receiver. In a native bridge shape,

`_callBridgeWithApprovals()` forwards `bridgeAmount + nativeValue` as raw ETH and only verifies that the receiver's ETH balance decreased enough. A raw executor-style callee can consume that ETH as a payment to an attacker-controlled payee even though no real second-hop bridge transfer occurred. If `maxHubNativeFeeByAdapter[CcipRouteAdapter]` is nonzero, the attacker can also consume the hub-funded native contribution.
The issue depends on allowlisting a selector that is safe only behind a dedicated wrapper such as `WormholeExecutorLib`, not safe as an opaque route target. The code currently has no per-selector metadata that separates bridge route selectors from other allowed bridge selectors.

Impact

Informational.

Recommendation

Split route-continuation allowlists from generic bridge or Sickle workflow allowlists. For every route selector, store metadata proving which calldata word is the amount field, whether native refunds must return to the receiver, and whether raw native value is allowed. Reject selectors without route metadata.

Developer Response

Fixed in #827 @ 42a08461.
Every invocation of `_validateAndPatchCalldata` now requires an admin-registered `RouteSelectorPolicy`. Calldata is patched using the **admin-pinned** `amountOffset` from the policy, not user-supplied. Dual-use bridge selectors can't be repurposed as route continuations.

- Struct: `contracts/bridges/BridgeRouteReceiver.sol:124`.
- Storage: `:186`.
- Admin setter `setRouteSelectorPolicy`: `:278`.
- Registration check + offset patch in `_validateAndPatchCalldata`: `:738`.
- Error `RouteSelectorNotRegistered`: `:61`.
- `NextHopParams.amountOffset` field removed.

2.9.9 Shared hub `composeFrom` allows cross-user Stargate deposit intent sniping

Technical Details

`BridgeDepositStrategy.executeStargateDeposit()` authenticates Stargate deposit intents by verifying a signature over
`(owner, token, paramsHash, intentId, deadline, srcEid, composeFrom)`. The signature scheme assumes that `composeFrom` uniquely identifies the upstream sender, which holds for direct Stargate transfers where `composeFrom` is a per-user EOA or dedicated sender contract. However, this assumption breaks in hub-routed flows where the second hop originates from a shared `BridgeRouteReceiver`.
When a first-hop delivery lands in `BridgeRouteReceiver`, the receiver becomes the Stargate caller for any second-hop bridge operation. The destination Stargate callback in `StargateRouteAdapter.lzCompose()` carries the hub receiver's address as `composeFrom` according to OFT compose semantics. Because `executeStargateDeposit()` binds only

`(srcEid, composeFrom)` for provenance without any user-unique upstream identifier, any attacker routing through the same hub can reuse a victim's signed deposit payload.
The exploit path requires the attacker to initiate their own first-hop route through any supported adapter that delivers to the target hub. In their route, `RouteParams.nextHop.bridgeCalldata` is crafted to perform a Stargate second hop carrying the victim's signed `(sickleOwner, params, intentId, deadline, signature)`. When `BridgeRouteReceiver.routeExternal()` calls `_executeBridge()`, the hub executes the Stargate send from its own address. The resulting destination callback presents the same `(srcEid, composeFrom)` provenance that the victim signed for: the hub chain EID and the shared hub receiver address. The signature verification in `_tryVerifyStargateSignature()` accepts the copied signature and marks `usedIntentIds[owner][intentId] = true`, consuming the victim's one-time authorization.
The victim's signed destination payload becomes observable through source-side transaction calldata or mempool data, as the `RouteParams` structure embedding the second-hop compose payload is passed onchain. The attacker must supply their own bridged funds matching the victim's signed deposit token and satisfying any `params.minAmount` requirement, but the attack succeeds regardless of whether the attacker-funded deposit into the victim's Sickle completes or reverts. Even if the deposit execution fails, `_finalizeVerifiedBridgeDeposit()` consumes the intent before the external self-call, so the victim's later legitimate delivery finds the intentId already used and falls back instead.
An attacker routing through the same Stargate hub can consume another user's one-shot deposit authorization with attacker-supplied funds. The attacker cannot steal victim assets, but can force an unwanted deposit into the victim's Sickle using attacker tokens or burn the victim's authorization before their real bridged delivery arrives. When the victim's legitimate second hop completes, signature verification fails on the consumed `intentId` and the intended automated deposit falls back to a simple token transfer. The victim loses destination-side automation and may waste bridge fees paid for a cross-chain deposit that never executes as intended.

Impact

Informational.

Recommendation

Introduce a dedicated authorization scheme for hub-routed Stargate deposits that binds a user-unique upstream identifier carried through the hub routing flow. This could include the authenticated source sender or owner, a per-route nonce, or a hash of the routed payload that cannot be replicated by another user's route. The signature verification should confirm that the upstream identity matches the claimed owner, preserving the one-shot authorization boundary even when the immediate Stargate sender is a shared contract.
Until that change is implemented, disable hub-routed Stargate deposits into `executeStargateDeposit()` by requiring that second-hop Stargate calls from `BridgeRouteReceiver` use a fallback path or a separate entrypoint that does not rely on `composeFrom` for user authentication.

Developer Response

Fixed in #827 @ 42a08461.
Destination (`BridgeDepositStrategy`): - `STARGATE_BRIDGE_DEPOSIT_TYPEHASH` binds single `bytes32 originator` — `contracts/strategies/BridgeDepositStrategy.sol:192`. -

`executeStargateDeposit(..., bytes32 originator, ...)` — `:432`. - `StargateDepositAdapter`: direct sends use `keccak256(srcEid, composeFrom)`; admin-allowlisted trusted hubs read originator from payload — `contracts/bridges/adapters/StargateDepositAdapter.sol:60-79`.
Hub (`BridgeRouteReceiver`): - `executeRoute` / `executeRouteWithNativeGasDropToken` take `bytes32 originator` — `contracts/bridges/BridgeRouteReceiver.sol:345,423`. - `_validateAndPatchCalldata` patches calldata at `originatorOffset` from `RouteSelectorPolicy`.
Adapter source-auth derivation: - CCIP: `CcipRouteAdapter.sol:67`. - Stargate: `StargateRouteAdapter.sol:35`. - Wormhole: `WormholeRouteAdapter.sol:74`. - AggLayer: `AggLayerRouteAdapter.sol:406`. - Across / DeBridge: pass `bytes32(0)` (no source-sender exposure at destination).

2.9.10 Destination bridge swaps cannot expire and may execute against stale DEX conditions

`BridgeSwapReceiver` executes destination-chain swaps using `block.timestamp` as the `SwapRouter.swap()` deadline, so a delayed bridge message can still execute a stale swap whenever the original `minAmountOut` remains satisfiable.

Technical Details

In `swapExternal()` and `swapWithNativeGasDropTokenExternal()`, the destination swap is dispatched as:

```
amountOut = msr.swap(
    swapSteps, swapAmount, minAmountOut, recipient, block.timestamp
);
```

The swap payloads accepted by the bridge swap adapters only carry `recipient`, `swapSteps`, `minAmountOut`, and `nativeGasDrop`. There is no user-supplied destination swap deadline propagated through the Across, CCIP, deBridge, Stargate, or Wormhole swap paths. Because bridge messages can be delayed by external bridge execution, relayer behavior, permissionless VAA execution, or bridge downtime, a swap built from a short-lived quote can still be executed much later. The usual `SwapRouter` deadline protection is effectively disabled for this surface, since the deadline is set to the same block in which the swap runs.
`BridgeRouteReceiver` already accepts a `swapDeadline` in `RouteParams` and forwards it to `SwapRouter.swap()`, showing the intended pattern for comparable cross-chain swap execution. Users remain protected against price loss below `minAmountOut`, but they have no way to cap the time window during which their destination swap is allowed to run.

Impact

Informational.

Recommendation

Add an explicit destination swap deadline to the swap payload format and forward it to `SwapRouter.swap()`. Extend the swap adapter payload decodes to include `uint256 swapDeadline`, plumb it through `executeSwap()` and

`executeSwapWithNativeGasDropToken()`, and pass it in both `swapExternal()` and `swapWithNativeGasDropTokenExternal()` instead of `block.timestamp`. Allow `type(uint256).max` only as an explicit opt-out, matching the route receiver pattern.

Developer Response

Fixed in #827 @ 42a08461.

- `BridgeSwapReceiver.executeSwap` / `executeSwapWithNativeGasDropToken` take trailing `uint256 swapDeadline` — `contracts/bridges/BridgeSwapReceiver.sol:153,181`.
- All 5 swap adapter payloads carry trailing `swapDeadline`: `AcrossSwapAdapter.sol:38`, `CcipSwapAdapter.sol`, `DeBridgeSwapAdapter.sol`, `StargateSwapAdapter.sol:47`, `WormholeSwapAdapter.sol:42`.
- Pass `type(uint256).max` to opt out.

yAudit

Sickle Bridge Strategies Update 2

Completed 2026-05-26