yAudit

May 2026

Prepared for
VFAT

Audited by
fedebianu
adriro

Sickle Bridge Strategies Update

Smart Contract Security Assessment

Contents

1 Review Summary **2**
- 1.1 Protocol Overview . . . 2
- 1.2 Audit Scope . . . 2
- 1.3 Risk Assessment Framework . . . 2
  - 1.3.1 Severity Classification . . . 3
- 1.4 Key Findings . . . 3
- 1.5 Overall Assessment . . . 4

2 Audit Overview **4**
- 2.1 Project Information . . . 4
- 2.2 Audit Team . . . 4
- 2.3 Audit Timeline . . . 4
- 2.4 Audit Resources . . . 4
- 2.5 Critical Findings . . . 6
- 2.6 High Findings . . . 6
- 2.7 Medium Findings . . . 6
- 2.8 Low Findings . . . 6
  - 2.8.1 `BridgeRouteReceiver` native rescue sweeps the entire ETH balance . . . 6
  - 2.8.2 Fee refund traps unspent fees when the fee token equals the routed token 7
  - 2.8.3 `BridgeRouteReceiver` failed ETH refunds lack recipient-level accounting 7
- 2.9 Gas Savings Findings . . . 8
- 2.10 Informational Findings . . . 8
  - 2.10.1 Signed Sickle deposit amounts are not a global spend cap for farm and NFT execution . . . 8
  - 2.10.2 Generic bridge deposit intents can be consumed by front-running . . . 9
  - 2.10.3 Redundant `minAmount` check in `depositFromSickleWithSignature()` 10
  - 2.10.4 `BridgeDepositStrategy.pause()` reimplements `onlyGuardianOrAdmin` . . . 11
  - 2.10.5 Bridge consumption check in `_callBridgeWithApprovals()` can be satisfied by `nativeValue` or fee token overlap . . . 12
  - 2.10.6 Intermediate swap deadline is hardcoded to `block.timestamp` . . . 13
  - 2.10.7 Fallback rewrap in `executeRoute()` is unreachable . . . 14
  - 2.10.8 `BridgeRouteReceiver` accepts unexpected native principal from adapters 15
  - 2.10.9 `BridgeAdapterBase._forwardAvailable()` comment misleadingly suggests fee-on-transfer tokens are supported . . . 16
  - 2.10.10 Native bridge consumption check can be satisfied by the relay fee . . . 16
- 2.11 Final Remarks . . . 17

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 12 smart contracts totaling approximately 1774 lines of code across 4 days of review.

```
contracts
├── base
│   └── GuardianAdmin.sol
├── bridges
│   ├── BridgeRouteReceiver.sol
│   ├── BridgeSwapReceiver.sol
│   ├── adapters
│   │   ├── BridgeAdapterBase.sol
│   │   ├── StargateAdapterBase.sol
│   │   ├── StargateDepositAdapter.sol
│   │   ├── StargateRouteAdapter.sol
│   │   └── StargateSwapAdapter.sol
│   ├── interfaces
│   │   └── ILayerZeroComposer.sol
│   └── libraries
│       └── OFTComposeMsgCodec.sol
├── libraries
│   └── BridgeLib.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	0
Medium	0
Low	3
Informational	10

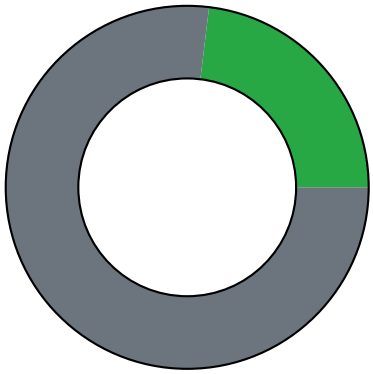

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The update narrows the review to the Stargate V2 migration, destination-side adapter hardening, and the route receiver used by hub-style bridge paths. The code remains compact and modular, with clear separation between adapter authentication, receiver execution, and strategy logic. The main residual risks are operational and integration-specific: Stargate pool/source configuration, replayable compose handling, balance-based downstream execution, and native-fee/refund accounting in second-hop routes.

2 Audit Overview

2.1 Project Information

Protocol Name: VFAT
Repository: https://github.com/vfat-io/sickle-contracts
Commit URL: 5a696e06c4a325177d1e50d2d327face7dbb33a7

2.2 Audit Team

fedebianu, adriro

2.3 Audit Timeline

The audit was conducted from May 11 to 14, 2026.

2.4 Audit Resources

- Code repositories and documentation

Category	Mark	Description
Access Control	Good	The audited contracts rely on layered authorization: admin and guardian controls, whitelisted bridge endpoints, whitelisted Stargate pools, source-EID checks, strategy adapter allowlists, and receiver bridge/selector allowlists.
Mathematics	Good	The code is not math-heavy. The relevant calculations are amount forwarding, minimum amount checks, balance-delta accounting, fee/native-value caps, and refund baselines. The main risks are accounting edge cases and live-balance semantics rather than complex arithmetic.
Complexity	Average	The adapter split keeps the Stargate-specific code relatively small, but the complete flow spans LayerZero compose payload parsing, local token mapping, swap/route receivers, downstream strategy multicalls, and hub-funded second-hop behavior. The design is understandable but has meaningful integration complexity.
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	The system remains operationally centralized around admin-managed bridge endpoint allowlists, pool-token mappings, source-EID configuration, receiver adapter lists, route allowlists, rescue functions, and hub-native fee caps. This is expected for a controlled bridge integration but should be treated as an explicit trust assumption.
Documentation	Average	The repository includes useful docs and deployment plans for the bridge system, but some safety properties depend on runbook-level configuration such as pool-token immutability, pending compose handling, and which adapters may receive nonzero hub-native fee caps. These assumptions should remain documented next to deployment procedures.
Testing and verification	Good	The test suite covers many bridge adapter, receiver, and strategy flows, including Stargate route behavior and failure handling. Coverage is helpful, although some bridge-level assumptions still rely on mocks, fork helpers, deployment-plan consistency tests, and operational configuration rather than full end-to-end production bridge execution.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

None.

2.7 Medium Findings

None.

2.8 Low Findings

2.8.1 `BridgeRouteReceiver` native rescue sweeps the entire ETH balance

Technical Details

`BridgeRouteReceiver.rescueETH()` is restricted to `onlyAdmin`, but it does not accept an `amount` parameter and always transfers `address(this).balance` to a single recipient. This is different from `rescueTokens()`, which lets the admin rescue a specific ERC20 amount. The coarse rescue path matters because native ETH can remain on `BridgeRouteReceiver` when `_refundExcessEth()` calls a non-payable `fallbackRecipient`. If more than one refund is stuck, or if the receiver also holds hub-native fee float for future route executions, `rescueETH(to)` cannot recover only the affected refund. The admin must sweep the full native balance and perform any recipient-level reconciliation off-chain.

Impact

Low. Funds are not permanently lost because the admin can recover the native balance, but the recovery primitive is all-or-nothing. This makes stuck refund handling operationally brittle: one rescue can also move unrelated hub float or other pending native balances, and the contract does not preserve recipient-level accounting for redistribution.

Recommendation

Add an amount-bounded native rescue function.

Developer Response

Fixed in 562a0b3a. `rescueETH(address to)` was replaced with `rescueETH(address to, uint256 amount)` to mirror `rescueTokens`.

2.8.2 Fee refund traps unspent fees when the fee token equals the routed token

Technical Details

`BridgeRouteReceiver.executeRoute()` snapshots `preCallFeeBalance` as `balanceOf(feeToken)` - `feeAmount` before the route runs. When the fee token is also the routed input token, this baseline still includes the bridged route amount. After `routeExternal` consumes the route amount and the second-hop bridge leaves `feeAmount` unspent, `_refundExcessFeeToken()` compares the current balance to the inflated baseline and returns without transferring the leftover fee. The contract explicitly supports fee forwarding and same-token approval accounting, but the refund baseline does not separate route principal from fee principal.

Impact

Low. Users lose access to unspent fee tokens for affected routes. The tokens remain trapped in `BridgeRouteReceiver` until an admin rescue, and repeated routes in this configuration accumulate stranded balances.

Recommendation

Separate fee-token accounting from routed-token accounting. When `feeToken` equals the input or bridge token, exclude both the delivered route amount and `feeAmount` from the baseline.

Developer Response

Fixed in d7512ccd. A new `FeeTokenConflictsWithRouteToken` revert was added at the `executeRoute` and `executeRouteWithNativeGasDropToken` entry points when `feeAmount > 0 && (feeToken == token || feeToken == params.nextHop.bridgeToken)`.

2.8.3 `BridgeRouteReceiver` failed ETH refunds lack recipient-level accounting

Technical Details

`BridgeRouteReceiver._refundExcessEth()` performs a raw `call` to `fallbackRecipient` and discards the return value.
The refund is invoked from both the success path and the fallback path. In both cases the route emits a high-level event afterwards:

- `BridgeRouteExecuted(fallbackRecipient, tokenIn, bridgeToken, amountIn, bridgeAmount`
- `BridgeRouteFallback(fallbackRecipient, token, fallbackAmount)`

Neither event carries any field that describes the outcome of the native refund. When `fallbackRecipient` is a contract without a payable `receive()` / `fallback()`, the refund silently fails and the leftover native value stays on `BridgeRouteReceiver`. From the outside,

the route looks fully successful because only the executed/fallback event is emitted; there is no signal that distinguishes "refund delivered" from "refund stuck on contract".
Because no refund-specific event is emitted, operators monitoring `BridgeRouteReceiver` cannot detect or reconcile silently failed refunds from emitted events alone; they have to infer them by replaying transactions and diffing `msg.value` against bridge consumption.

Impact

Low. The native ETH is not permanently lost because the admin can recover it, but users cannot claim their failed refunds and the admin has no accounting of how much is owed to each recipient. If multiple refunds fail, or if the receiver also holds hub-native fee float, correct reconciliation requires off-chain analysis and trusted manual redistribution.

Recommendation

Emit an explicit event when the refund call fails:

```
event RefundFailed(address indexed recipient, uint256 amount);

function _refundExcessEth(
    address recipient,
    uint256 baseBalance
) internal {
    uint256 currentBalance = address(this).balance;
    if (currentBalance <= baseBalance) return;

    uint256 amount = currentBalance - baseBalance;
    (bool success,) = recipient.call{ value: amount }("");
    if (!success) emit RefundFailed(recipient, amount);
}
```

This keeps the current "do not revert the route on refund failure" semantics but makes the failure auditable: operators can monitor `RefundFailed` and attribute the stuck amount to a specific recipient.

Developer Response

Fixed in 562a0b3a. `_refundExcessEth` now emits `RefundFailed(recipient, amount)` when the call fails.

2.9 Gas Savings Findings

None.

2.10 Informational Findings

2.10.1 Signed Sickle deposit amounts are not a global spend cap for farm and NFT execution

Technical Details

`BridgeDepositStrategy` defines `SICKLE_DEPOSIT_TYPEHASH` and binds `amount` into the keeper-driven `SickleBridgeDeposit` signature. `depositFromSickleWithSignature()` verifies the signature, checks `amount >= params.minAmount`, consumes the intent, resolves the user's Sickle, and calls `_executeDeposit(sickle, token, amount, params)`.
The comment above `SICKLE_DEPOSIT_TYPEHASH` states that binding `amount` makes the signed amount authoritative and that, if the Sickle balance is above `amount`, only `amount` is deposited while the excess remains in the Sickle. This is misleading as a general statement. Binding `amount` prevents the keeper from choosing an arbitrary scalar amount, but the user also signs `params`, and those params can intentionally encode balance-based farm or NFT behavior.
The signed `amount` is not enforced as a global spend cap over all balances that the downstream farm or NFT multicalls can consume. `_executeDeposit()` dispatches to either the farm path or the NFT path after validating only the signed zap shape.
`amount` is used as the signed entry amount, fee basis, and event amount, while parts of the downstream farm and NFT execution still use live Sickle balances. That can make an old still-valid signature interact with balances that arrived after the user signed the intent.

Impact

Informational.

Recommendation

Update the comment above `SICKLE_DEPOSIT_TYPEHASH` and document that the signed `amount` is not a full per-token spend cap for keeper-driven Sickle deposits.

Developer Response

Acknowledged. The comment above `SICKLE_DEPOSIT_TYPEHASH` overstated the guarantee: it implied that `amount` bounds total per-token spend ("only `amount` is deposited and the excess stays in the Sickle").
Binding `amount` makes it authoritative only as the entry, fee, and event amount, which is what prevents a keeper from substituting a different scalar (or calling with `amount = 0` to burn the `intentId`). It is not a global per-token spend cap: the signer also signs `params`, and the downstream farm/NFT zap multicalls execute against live Sickle balances according to those signed `params`. Authority over what downstream execution consumes comes from `params`, not `amount`.
The comment has been corrected to document this explicitly and to note that intent `deadline` values should be kept short to bound the stale-signature window. This was a comment-only change with no behavioral impact.

2.10.2 Generic bridge deposit intents can be consumed by front-running

Technical Details

`BridgeDepositStrategy` uses the generic `BRIDGE_DEPOSIT_TYPEHASH` for non-Stargate bridge deposit callbacks. The signed struct binds only `owner`, `token`, `paramsHash`,

`intentId`, and `deadline`. The signature verification path in `_tryVerifySignature()` does not bind the adapter, source chain, source sender, bridge order id, or any other provenance for the delivery that supplied the tokens.
Generic adapters decode the user payload and then call `executeDeposit()` with the amount they forwarded to the strategy. This applies to `AcrossDepositAdapter.handleV3AcrossMessage()`, `DeBridgeDepositAdapter.onERC20Received()`, and `WormholeDepositAdapter.executeVAAv1()`. If the signature is valid and the delivered amount is at least `params.minAmount`, `_finalizeVerifiedBridgeDeposit()` marks `usedIntentIds[sickleOwner][intentId] = true` before executing the deposit.
As a result, anyone who observes a still-valid generic bridge deposit payload and signature can fund their own delivery of the same token through a whitelisted generic bridge adapter, include the observed payload, and consume the victim's `intentId` first. The attacker cannot change the signed params and must deliver enough of their own tokens to satisfy `params.minAmount`; the forwarded funds are deposited according to the victim's signed instructions. The practical effect is that the victim's later intended bridge delivery reaches `executeDeposit()` after the `intentId` has already been consumed, causing `_tryVerifySignature()` to return false and the later delivery to fall back to the owner instead of completing the intended deposit.
The Stargate path has a separate typehash that binds `srcEid` and `composeFrom` in `STARGATE_BRIDGE_DEPOSIT_TYPEHASH`, so this observation is limited to the generic bridge deposit authorization model.

Impact

Informational.

Recommendation

If generic bridge deposits should be protected from this type of intent sniping, bind bridge provenance into the signed message wherever the underlying bridge exposes stable authenticated fields, such as adapter address, source chain identifier, source sender, or order/message id. Where stable provenance is unavailable, consider requiring a distinct per-delivery secret or nonce that is not revealed until the intended delivery is ready to execute. If the behavior is accepted, document that generic bridge deposit signatures authorize the first matching delivery that supplies enough token, regardless of who funded that delivery.

Developer Response

Acknowledged. It is detrimental only to the attacker and largely beneficial to the user.

2.10.3 Redundant `minAmount` check in `depositFromSickleWithSignature()`

The `amount < params.minAmount` check in `depositFromSickleWithSignature()` is redundant because both `amount` and `params.minAmount` are bound by the EIP-712 signature, so the only way the check can fail is if the signer produced an inconsistent payload.

Technical Details

`depositFromSickleWithSignature()` first calls `_tryVerifySickleSignature()`, which hashes the typed-data struct `SickleBridgeDeposit(address owner,address token,uint256 amount,bytes32 paramsHash,bytes3` and recovers the signer. The struct hash includes both `amount` and `keccak256(abi.encode(params))`, where `params` is the full `DepositParams` and therefore covers `params.minAmount`.
Once the signature is verified, the contract still performs:

```
if (amount < params.minAmount) {
    revert AmountBelowMinimum();
}
```

Since the caller cannot tamper with either value without invalidating the signature, this branch can only be reached when the original signed payload is internally inconsistent (the signer signed an `amount` that is below their own `minAmount`). In normal use the check is dead code, and on this path no funds sit in the strategy, so there is no slippage protection role for it to play beyond the signer's existing commitment.
Note that this differs from the callback path in `_finalizeVerifiedBridgeDeposit()`, where `amount` is the bridge-delivered amount and can legitimately be lower than the signed `minAmount`.

Impact

Informational.

Recommendation

Remove the `amount < params.minAmount` check from `depositFromSickleWithSignature()`, since the EIP-712 signature already binds both fields. Alternatively, document explicitly that the check exists only as a sanity guard against malformed signed payloads.

Developer Response

Intentionally retained as a sanity guard against signer-side bugs (the EIP-712 hash binds both `amount` and `params.minAmount`, so the branch only fires on internally inconsistent signed payloads). The existing rationale comment at `BridgeDepositStrategy.sol` lines 71-72 documents this; no code change.

2.10.4 `BridgeDepositStrategy.pause()` reimplements `onlyGuardianOrAdmin`

`BridgeDepositStrategy` already inherits `GuardianAdmin`, which exposes the `onlyGuardianOrAdmin` modifier and the canonical `NotGuardianOrAdminError`. The `pause()` function inlines the same access check and declares its own `NotGuardianOrAdmin` error instead of reusing the inherited modifier.

Technical Details

`pause()` is implemented as:

```
function pause() external {
    if (msg.sender != guardian && msg.sender != admin) {
        revert NotGuardianOrAdmin();
    }
    ...
}
```

`GuardianAdmin._onlyGuardianOrAdmin()` performs the exact same comparison and reverts with `NotGuardianOrAdminError`. Inlining the check in `pause()` duplicates code, introduces a second error type with a nearly identical name (`NotGuardianOrAdmin` versus `NotGuardianOrAdminError`), and makes the access surface harder to audit. Other contracts in the codebase such as `BridgeSwapReceiver` and `BridgeAdapterBase` already use the inherited `onlyGuardianOrAdmin` modifier directly.

Impact

Informational.

Recommendation

Replace the inline check with the `onlyGuardianOrAdmin` modifier from `GuardianAdmin` and remove the local `NotGuardianOrAdmin` error declaration. Off-chain consumers that decode the revert selector should be updated to expect `NotGuardianOrAdminError` to keep the revert surface consistent across the codebase.

Developer Response

Fixed in 562a0b3a. `pause()` now uses the inherited `onlyGuardianOrAdmin` modifier; the local `NotGuardianOrAdmin` error was removed (the revert selector becomes `NotGuardianOrAdminError`).

2.10.5 Bridge consumption check in `_callBridgeWithApprovals()` can be satisfied by `nativeValue` or fee token overlap

`BridgeRouteReceiver._callBridgeWithApprovals()` enforces that the bridge contract consumed at least `bridgeAmount` by comparing the receiver's pre and post balances of `bridgeToken`. When the bridged token is native ETH, the comparison also captures `nativeValue`, and when the bridged token equals the fee token, it also captures `feeAmount`. In both cases the check can pass even though the bridge under-consumed the principal.

Technical Details

The function builds `callValue` as `nativeValue + bridgeAmount` for native bridges and as `nativeValue` for ERC20 bridges, then snapshots the post-call balance:

```
uint256 balanceAfter = bridgeToken == address(0)
    ? address(this).balance
    : IERC20(bridgeToken).balanceOf(address(this));
if (balanceBefore - balanceAfter < bridgeAmount) {
    revert BridgeDidNotConsumeTokens();
}
```

Two collisions weaken this check:

1. When `bridgeToken == address(0)`, `balanceBefore` is taken before the call and `callValue` already includes both `bridgeAmount` and `nativeValue`. A bridge that retains the entire native fee but only consumes part of `bridgeAmount` can still produce `balanceBefore - balanceAfter >= bridgeAmount`, because the unspent portion of `bridgeAmount` is hidden behind the consumed `nativeValue`.
2. When `feeToken == bridgeToken`, `_setApprovals()` approves `bridgeAmount + feeAmount` of the same ERC20. Any portion of `feeAmount` that the bridge pulls counts toward the same balance delta, so the consumption check can be satisfied by fee tokens that were never meant to fund the bridged principal.

In both cases, the receiver treats the bridge as having consumed the full principal even though some of it is left behind in the receiver and effectively credited elsewhere in the accounting.

Impact

Informational.

Recommendation

Track the principal separately from the relay fee and the same-asset fee. For native bridges, snapshot the pre-call balance after subtracting `nativeValue` (or compare against `bridgeAmount + nativeValue`). For the `feeToken == bridgeToken` case, snapshot the receiver's balance using only the principal portion, or assert against `bridgeAmount + feeAmount` while the second-hop bridge call is in flight, so partial consumption of either component cannot offset the other.

Developer Response

Fixed in d7512ccd. Native consumption check now requires `balanceBefore - balanceAfter >= bridgeAmount + nativeValue`. The `feeToken == bridgeToken` overlap is closed by a separate fix (same commit), which rejects the collision at entry and makes the `feeIsBridgeToken` branch in `_setApprovals` dead.

2.10.6 Intermediate swap deadline is hardcoded to `block.timestamp`

`BridgeRouteReceiver._swapToBridgeToken()` calls `MultiSwapRouter.swap()` with `block.timestamp` as the deadline, which silently disables the deadline check for the intermediate swap that connects bridge hops.

Technical Details

In `_swapToBridgeToken()` the swap is invoked as:

```
amountOut = msr.swap(
    swapSteps, amount, minSwapOut, address(this), block.timestamp
);
```

Passing `block.timestamp` makes the deadline equal to the current block, so the underlying router cannot reject a stale execution. Because this swap runs on the destination chain after a bridge delivery, the user has no direct control over when it executes, and the gap between intent submission and execution can be arbitrarily long. Without a user-supplied deadline, the only protection against unfavorable execution is `minSwapOut` (already passed via `params.minSwapOut`), which alone does not bound how long the route can sit before being executed.

Impact

Informational.

Recommendation

Expose the swap deadline as a field on `RouteParams` and forward it to `msr.swap()`. Callers that intentionally want to skip the check can set it to `type(uint256).max`, but the default should let the source-chain caller bound how long the destination-side swap remains executable.

Developer Response

Fixed in 562a0b3a. The new `uint256 swapDeadline` field on `RouteParams` is forwarded to `msr.swap()`. Callers that intentionally skip the check pass `type(uint256).max`.

2.10.7 Fallback rewrap in `executeRoute()` is unreachable

`BridgeRouteReceiver.executeRoute()` calls `_rewrapNativeFallbackToken()` from the `catch` branch to rewrap unspent native ETH back into WETH for the user. Because the `catch` branch is entered only after `routeExternal()` has reverted and rolled back its state changes, this rewrap path can never perform useful work.

Technical Details

`routeExternal()` is invoked through `this.routeExternal{ value: msg.value }(...)`. If it reverts, all state and balance changes performed inside that subcall (the swap that turned WETH into ETH, the `_executeBridge()` call, the approvals, and any partial native ETH spent) are reverted along with it. Once execution reaches the `catch` block, the only ETH the receiver holds beyond `preCallEthBalance` is `msg.value`, which `_rewrapNativeFallbackToken()` already excludes via the `currentBalance <= baseBalance + msg.value` short circuit.

As a result, `wrapAmount` is always zero and the WETH `deposit` call is never reached. The function is dead code in the current control flow, but it still adds gas overhead, an extra external `wrappedNative()` lookup, and a misleading suggestion that partial fallbacks are handled.

Impact

Informational.

Recommendation

Remove `_rewrapNativeFallbackToken()` from the `catch` path or, if the intent is to recover from a future code path that performs an irreversible WETH unwrap before the bridge call, document that scenario explicitly and gate the rewrap on it. As written, the helper can be deleted along with its call sites.

Developer Response

Fixed in 562a0b3a. The `_rewrapNativeFallbackToken()` helper and both catch-side call sites were removed (state inside the external self-call already rolls back on revert).

2.10.8 `BridgeRouteReceiver` accepts unexpected native principal from adapters

Technical Details

`BridgeRouteReceiver.executeRoute()` accepts an adapter-supplied `token` and `amount` without rejecting `token == address(0)`. If `params.nextHop.bridgeToken == address(0)` and there are no swap steps, `_resolveBridgeTokenAmount()` treats that native token route as valid and forwards the nominal `amount` into `_callBridgeWithApprovals()`. In that native-bridge branch, the receiver sends `nativeValue + bridgeAmount` from its own ETH balance.
This shape is not expected from the current route adapters. Stargate wraps native ETH to WETH before calling the receiver, then calls `executeRoute(token, forwarded, address(0), 0, params)` with `token = weth`. The DeBridge route adapter reverts on native ETH delivery, AggLayer rejects native origin tokens, CCIP delivers ERC20 `tokenAmounts`, Wormhole resolves to a local ERC20/wrapped asset, and Across does not provide a native ETH delivery path to the receiver.
Because no current adapter should pass `token == address(0)` as the delivered route asset, accepting that shape in `BridgeRouteReceiver` is a defensive-hardening gap rather than a currently reachable exploit. If a future or misconfigured whitelisted adapter passed `token == address(0)` with a positive `amount` without actually delivering matching native principal, the receiver could spend existing ETH balance as route principal instead of only as the separately capped `nextHop.nativeValue` fee.

Impact

Informational.

Recommendation

Reject unexpected native principal at the receiver boundary unless and until a specific adapter is designed to deliver native ETH principal safely.

Developer Response

Fixed in 562a0b3a. Both `executeRoute` and `executeRouteWithNativeGasDropToken` now revert with `NativePrincipalNotSupported` when `token == address(0)`.

2.10.9 `BridgeAdapterBase._forwardAvailable()` comment misleadingly suggests fee-on-transfer tokens are supported

Technical Details

`BridgeAdapterBase._forwardAvailable()` is documented as handling fee-on-transfer tokens and bridge under-delivery:

```
/// @dev Forward up to `amount` tokens to `to`, capped to actual balance.
/// Handles fee-on-transfer tokens and bridge under-delivery.
function _forwardAvailable(
```

The function caps the attempted transfer only by the adapter's current balance and returns that nominal `sent` value. It does not measure the recipient's post-transfer balance delta. For a fee-on-transfer or tax token, the recipient can receive less than `sent`, while downstream code still treats `sent` as the delivered amount.

Impact

Informational.

Recommendation

Update the comment.

Developer Response

Fixed in 562a0b3a. The NatSpec for `BridgeAdapterBase._forwardAvailable()` was updated to reflect that it only caps to the adapter's balance and does not measure the recipient's balance delta.

2.10.10 Native bridge consumption check can be satisfied by the relay fee

When bridging native ETH, the post-bridge consumption check in `BridgeLib.bridge()` measures the change in the Sickle's full ETH balance, which also includes the forwarded relay fee. As a result the check can pass even when the bridge contract did not actually consume the requested `amount` of bridged tokens.

Technical Details

In `BridgeLib.bridge()` the native branch records `balanceBefore = address(this).balance` after unwrapping WETH and then calls the bridge contract with `callValue = amount + msg.value`, where `msg.value` is the relay fee. The success check is:

```
if (balanceBefore - balanceAfter < amount) {
    revert BridgeDidNotConsumeTokens();
}
```

Because `callValue` includes both the bridged `amount` and the relay fee, `balanceBefore - balanceAfter` reflects whatever portion of `amount + msg.value` the bridge contract kept. A bridge that under-consumes the bridged amount but fully retains the relay fee, or that internally accounts for the relay fee against the bridged amount, can still satisfy `balanceBefore - balanceAfter >= amount` while bridging less than `amount` of native ETH. The ERC20 branch is not affected because the relay fee is paid separately as `msg.value` and does not move ERC20 balances.

Impact

Informational.

Recommendation

Track the bridged amount and the relay fee separately for native ETH. For example, snapshot `balanceBefore = address(this).balance - msg.value` before the call, or compare `balanceBefore - balanceAfter` against `amount + msg.value` so that any unconsumed relay fee is not credited toward the bridged amount.

Developer Response

Fixed in d7512ccd. `BridgeLib.bridge()` snapshots `address(this).balance - msg.value` after the unwrap; bridges that refund any portion of the relay fee now revert.

2.11 Final Remarks

Overall, the Stargate update is well structured: adapters authenticate the bridge endpoint and Stargate source context, delivered amounts are forwarded through receiver/strategy boundaries, and failure paths usually preserve user recoverability. The strongest remaining concerns are edge cases at integration boundaries, not broad code quality issues.

Sickle Bridge Strategies Update

Completed 2026-05-14