yAudit

April 2026

Prepared for
VFAT

Audited by
fedebianu
adriro

Sickle Bridge Strategies

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 13 smart contracts totaling approximately 1600 lines of code across 6 days of review.

```
contracts
├── bridges
│   ├── BridgeSwapReceiver.sol
│   └── adapters
│       ├── AcrossDepositAdapter.sol
│       ├── AcrossSwapAdapter.sol
│       ├── BridgeAdapterBase.sol
│       ├── DeBridgeDepositAdapter.sol
│       ├── DeBridgeSwapAdapter.sol
│       ├── StargateDepositAdapter.sol
│       ├── StargateSwapAdapter.sol
│       ├── WormholeDepositAdapter.sol
│       └── WormholeSwapAdapter.sol
├── libraries
│   └── BridgeLib.sol
└── strategies
    ├── BridgeDepositStrategy.sol
    └── BridgeWithdrawStrategy.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	2
Low	1
Informational	12

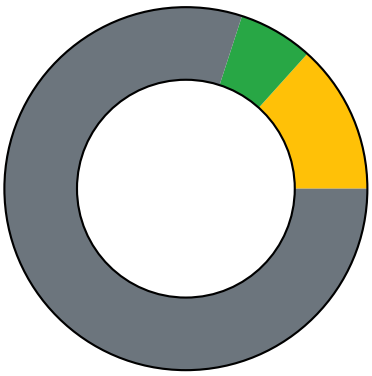

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The bridge extension is modular and reasonably easy to follow, with a clean adapter split between source-chain withdrawal/bridging and destination-chain execution. The main risk themes are bridge-specific trust assumptions, payload handling, and edge-case destination execution semantics, especially around legacy Wormhole integration, manual recovery flows, and protocol-specific callback behavior.

2 Audit Overview

2.1 Project Information

Protocol Name: VFAT
Repository: https://github.com/vfat-io/sickle-contracts
Commit Hash: 3608c27f504a2b328315989f151eefe64148c134

2.2 Audit Team

fedebianu, adriro

2.3 Audit Timeline

The audit was conducted from April 1 to 8, 2026.

2.4 Audit Resources

- Code repositories and documentation

Category	Mark	Description
Access Control	Good	The bridge layer uses sensible local access-control primitives such as whitelisted endpoints, whitelisted adapters, Sickle caller/target allowlists, and signature-gated destination deposits. The main caveat is that each bridge has a different trust model, so correctness depends on those assumptions being documented and enforced consistently.
Mathematics	Good	The code is not math-heavy, but several paths rely on balance-based accounting, amount caps, and callback-delivered values. The main issues are edge-case accounting and spend-bound semantics rather than complex formulas.
Complexity	Average	The adapter pattern keeps each bridge integration relatively isolated, but overall system complexity is increased by having multiple protocols with different callback models, payload formats, and fallback semantics. This is manageable but not trivial.
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, fee settings, and trusted bridge endpoint configuration. Safe operation also depends on external bridge infrastructure and offchain routing assumptions.
Code Stability	Average	Most contracts are compact and structurally clear, but some bridge paths are still evolving, especially around Wormhole migration and protocol-specific delivery semantics. This suggests moderate maturity rather than fully settled integration code.
Documentation	Average	The repository includes useful bridge flow documentation, but parts of the docs lag behind implementation changes and protocol migrations. Integrators should not rely on documentation alone without checking the current contracts.
Monitoring	Average	Events and fallback behavior provide a workable monitoring surface, but robust operational monitoring is still needed around bridge delivery failures, stale balances, manual recovery paths, and protocol-specific adapter configuration.
Testing and verification	Good	Test suite covers many adapter and strategy flows, including failure-path behavior and protocol-specific edge cases. Coverage is materially helpful, although some bridge assumptions are still validated with mocked callbacks rather than full protocol E2E environments.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

None.

2.7 Medium Findings

2.7.1 Wormhole adapters process relayer payloads without redeeming the bridged tokens

Technical Details

`WormholeDepositAdapter.receiveWormholeMessages()` and `WormholeSwapAdapter.receiveWormholeMessages()` decode `token` and `amount` directly from the relayed payload and immediately call `BridgeAdapterBase._forwardAvailable()` using the adapter's current balance. They do not inspect or redeem the token transfer VAA from `additionalMessages`.
In Wormhole's token-and-payload flow, the relayer callback delivers the application payload together with token-transfer VAAs in `additionalMessages`; the receiver is expected to verify/redeem the token transfer before using the funds. Wormhole's current token transfer examples use `TokenReceiver` with `receivePayloadAndTokens()` from Wormhole SDK, where the transferred tokens are materialized only after the token transfer is redeemed. See:

- https://docs.wormhole.com/products/messaging/tutorials/cross-chain-token-contracts/
- HelloToken example using SDK
- HelloToken example without SDK

Because these adapters ignore `additionalMessages`, the normal Wormhole token+payload route is incomplete in-scope: if the adapter is not already pre-funded with the payload-selected token, `BridgeAdapterBase._forwardAvailable()` forwards zero or only stale balance already resident in the adapter. The downstream deposit/swap logic then either no-ops or falls back, while the actual bridged transfer remains outside this execution path until redeemed separately.

Impact

Medium. Wormhole deposit and swap routes can fail to deliver the bridged funds into the intended destination flow even when the relayer callback succeeds. Users can end up with broken cross-chain execution, zero-amount fallbacks, or reliance on stale/pre-funded adapter balances instead of the actual Wormhole transfer.

Recommendation

As the Standard Relayer has been deprecated, take this in consideration when developing the new adapter:

The entry point changes; instead of receiveWormholeMessages, your contract now receives a VAA via executeVaaV1 from the Relay Provider and verifies it through parseAndVerifyVM on the Wormhole Core contract.

Here you can find an implementation example of `executeVaaV1()`.

Developer Response

Fixed in PR#682 - commit 8fbc2532. The adapter now redeems the Token Bridge VAA itself via `tokenBridge.completeTransferWithPayload()` before forwarding.

2.7.2 An attacker can sweep arbitrary adapter token balances from Wormhole adapter

Technical Details

`WormholeSwapAdapter.receiveWormholeMessages()` and `WormholeDepositAdapter.receiveWormholeMessages()` decode `token` and `amount` from the untrusted payload before checking whether `sourceChain` and `sourceAddress` are whitelisted emitters. If the emitter is not whitelisted, both functions enter a fallback branch that calls `BridgeAdapterBase._forwardAvailable(token, recipient, amount)`. `_forwardAvailable()` does not verify that `token` is the asset actually delivered by Wormhole. It only checks the adapter's current balance for the attacker-chosen token and transfers `min(balance, amount)`. As a result, any caller reaching the Wormhole relayer callback path with an unapproved emitter can name an arbitrary ERC20 already resident in the adapter and force the fallback branch to transfer it out.
Attack path:
The adapter enforces two layers of authentication:

1. `_onlyWhitelistedEndpoint()` — verifies `msg.sender` is the Wormhole relayer contract on the destination chain.
2. `whitelistedWormholeEmitters[sourceChain][sourceAddress]` — verifies the originating contract on the source chain is trusted.

The Wormhole relayer is a permissionless delivery service. An attacker can trigger the fallback path as follows:

1. Send a Wormhole message from any supported source chain targeting the adapter address on the destination chain, with a crafted payload. This only costs the Wormhole relay fee.
2. The Wormhole guardians sign the VAA. The `sourceAddress` and `sourceChain` are cryptographically authenticated and genuinely reflect the attacker's address.
3. The Wormhole relayer on the destination chain calls `adapter.receiveWormholeMessages(payload, [], sourceAddress, sourceChain, hash)`. Since `msg.sender` is the Wormhole Relayer, `_onlyWhitelistedEndpoint()` passes.
4. The attacker's `sourceAddress` is not in `whitelistedWormholeEmitters`, so the adapter enters the fallback branch.
5. The fallback decodes `token`, `amount`, and `recipient` from the attacker-controlled payload and calls `_forwardAvailable(token, recipient, amount)`, transferring `min(adapterBalance, amount)` of the chosen token to the attacker.

Impact

Medium. Residual ERC20 balances left in either Wormhole adapter can be stolen by unapproved Wormhole senders. Any dust, accidental transfers, or temporarily stranded user funds in the adapter become sweepable to an attacker-controlled address.

Recommendation

Consider removing the whitelisted emitter check as part of the Executor migration and use the fallback at strategy level.

Developer Response

Fixed in PR#682 in commit `8fbc2532` .

2.8 Low Findings

2.8.1 `BridgeDepositStrategy.depositFromSickle()` can be griefed by donating the input token when the same token is swept

Technical Details

`BridgeDepositStrategy.depositFromSickle()` is intended to cap consumption of the declared bridge token by measuring `balanceBefore - balanceAfter` on that token and reverting with `ManualDepositAmountExceeded` if the consumed amount exceeds the caller-supplied `amount` .
However, the function does not isolate user-supplied funds from third-party dust already present in the victim Sickle. If an attacker transfers even `1 wei` of the same `token` to the victim Sickle before the manual deposit executes, that dust is included in `balanceBefore` .
This becomes exploitable when the same `token` is also present in `params.sweepTokens` , because `TransferLib.transferTokensToUser()` transfers the full token balance of the Sickle to the owner. In that case the dusted `1 wei` is also consumed by the flow, so the post-check observes `consumed = amount + 1` and the transaction reverts even though the caller only intended to spend `amount` .

Impact

Low. An attacker can cheaply grief `depositFromSickle()` by dusting the victim Sickle with a negligible amount of the input token, causing otherwise valid manual bridge deposits to revert when that same token is swept. This is a DoS vector against the manual destination-side recovery path.
PoC

```
function test_depositFromSickle_reverts_when_consumption_exceeds_amount_dos() public {
    uint256 balance = 1000e18;
    token.mint(address(ctx.sickle), balance);

    address attacker = makeAddr("attacker");
```

```
    token.mint(address(attacker), balance);
    vm.prank(attacker);
    token.transfer(address(ctx.sickle), 1);

    BridgeDepositStrategy.DepositParams memory params =
        _emptyDepositParams();
    params.sweepTokens = new address[](1);
    params.sweepTokens[0] = address(token);

    vm.prank(sickleOwner);
    vm.expectRevert(
        BridgeDepositStrategy.ManualDepositAmountExceeded.selector
    );
    strategy.depositFromSickle(address(token), 1000e18, params);

    assertEq(token.balanceOf(address(ctx.sickle)), balance + 1);
    assertEq(token.balanceOf(sickleOwner), 0);
}
```

Recommendation

Do not allow token sweeping in `depositFromSickle()`. Alternatively, since sweeping residual token balances is not a core functionality, clearly document this behavior so users can choose whether to include sweeping in the manual deposit flow.

Developer Response

Fixed in PR#680 (merge commit 298bdca2). The accepted final form is in commit 505d69fb — refactor: simplify `depositFromSickle` to just execute deposit and sweep leftovers — which removes the balance-before/after accounting (and the `ManualDepositAmountExceeded` error) entirely, so dust donations no longer cause a revert.

2.9 Gas Savings Findings

2.9.1 Redundant connector registry lookup in `_executeNftDeposit()`

In `_executeNftDeposit()`, the NFT liquidity connector is fetched from the registry a second time despite already being stored in a local variable.

Technical Details

At the start of `_executeNftDeposit()`, when `params.mode == DepositMode.NftDeposit`, the connector is looked up and stored:

```
_liqConnector = INftLiquidityConnector(
    connectorRegistry.connectorOf(address(nft.nft))
);
```

Later in Step 4 (still inside the `NftDeposit` branch), a new `liquidityConnector` variable is created with a second `connectorRegistry.connectorOf(address(nft.nft))` call. This is a redundant external call that wastes gas.

Impact

Gas Savings.

Recommendation

Reuse the existing `_liqConnector` variable instead of performing a second registry lookup.

Developer Response

Fixed in PR#674 - commit a05e9c72. Reuses `_liqConnector` in `_executeNftDeposit` instead of a second `connectorRegistry.connectorOf` call.

2.10 Informational Findings

2.10.1 Across relayer can spoof the bridge message payload

In the Across bridge integration, the relayer-delivered `message` is not integrity-checked by the SpokePool. According to the Across documentation, relayers can submit arbitrary message bytes when filling a relay. However, a relayer who spoofs a message **will not be repaid** by the Across protocol, making such an attack economically impractical under normal conditions.

Technical Details

The Across docs explicitly state the following security model for handler contracts:

> Handler contracts only uses the funds that are sent to it. That means that the message is assumed to only have authority over **those funds and, critically, no outside funds**. This is important because relayers can send invalid relays. They will not be repaid if they attempt this, but if an invalid message could unlock other funds, then a relayer could spoof messages maliciously. Message data should otherwise be treated as spoofable and untrusted for use **beyond directing the funds passed along with it**.

In the current implementation, both Across adapter paths satisfy the Across security model:

- **`AcrossSwapAdapter`**: The handler decodes `recipient`, `swapSteps`, and `minAmountOut` from the message. However, the funds operated on are strictly limited to the bridged tokens forwarded via `_forwardAvailable(tokenSent, address(receiver), amount)`, which caps the transfer to `min(balance, amount)`. Even if a relayer spoofs the `recipient`, they would lose their relay capital on the source chain without being repaid.
- **`AcrossDepositAdapter`**: The handler decodes `sickleOwner`, `params`, `intentId`, `deadline`, and `signature`. The deposit path requires a valid EIP-712 signature from the `sickleOwner`, so spoofing any of these fields would cause signature verification to fail. The fallback branch calls `_transferAvailable(token, sickleOwner, amount)`, which again only transfers up to `amount` of the bridged token held by the strategy

contract. A relayer redirecting the fallback to their own address would forfeit the relay cost on the source chain.

Since neither path grants access to any funds beyond the bridged amount, the economic disincentive (no repayment for spoofed relays) makes this attack impractical.

Impact

Informational. The Across message payload is not integrity-verified on-chain, but the current adapter implementations correctly follow the Across security model by restricting fund access to the bridged tokens only. The economic penalty for relayers submitting spoofed messages (no repayment) makes exploitation impractical.

Recommendation

Be aware that the Across message should be treated as untrusted. The current adapters are safe because they only direct the bridged funds and do not grant access to outside funds. If future callbacks or adapter modifications introduce access to additional funds beyond the bridged amount (e.g., funds already held in the handler, the Sickle, or any external contract), message integrity verification should be added — for example, by including a depositor signature over critical parameters in the message payload, as suggested by the Across documentation.

Developer Response

Acknowledged.

2.10.2 Wormhole adapters rely on fixed payload offsets for VAA parsing

Technical Details

`WormholeAdapterBase._redeemTransfer()` uses a sound security model: it snapshots the token balance, calls `tokenBridge.completeTransferWithPayload()`, and forwards only the balance delta actually received. However `_extractTokenFromVAA()` manually parses the raw VAA using hardcoded offsets.
This hand-rolled parsing with magic offsets is harder to review and maintain than using the parser functions already available in the Wormhole SDK. Any future adapter maintenance would require re-deriving and re-auditing all offsets manually.

Impact

Informational.

Recommendation

Consider replacing the manual offset-based parsing with Wormhole parsing functions:

1. Use Wormhole Core `parseVM(bytes)` to parse the VAA body instead of extracting fields from raw offsets.

2. Use Token Bridge `parseTransferWithPayload(bytes)` to decode the bridged token, token chain, and custom payload from the transfer message returned by `completeTransferWithPayload(bytes)`.
3. Derive the custom payload from parsed `TransferWithPayload.payload` instead of slicing at a hardcoded `133` offset.

Developer Response

Fixed in PR#699.

2.10.3 deBridge hook configuration should be carefully chosen for atomicity and failure behavior

When bridging tokens through deBridge DLN, the hook attached to the order can be configured with different atomicity and success modes that affect how failures are handled on the destination chain.

Technical Details

deBridge hooks support two independent configuration axes: atomicity (atomic vs non-atomic) and success requirement (success-required vs success-optional).
Given that the `DeBridgeDepositAdapter` and `DeBridgeSwapAdapter` contracts execute on-chain actions (deposits, swaps) that depend on the order's output tokens being available in the same transaction, the hooks should be configured as **atomic**. Non-atomic hooks allow solvers to fill the order without executing the hook, leaving the order's output tokens in the DLN intermediary contract until a separate transaction triggers the hook or the authority cancels it. This introduces unnecessary complexity and a dependency on an external party to trigger execution.
For the success requirement, two valid configurations exist:

1. **Success-optional**: If the hook reverts, the order is still filled and the output tokens are sent to the fallback address specified in the hook envelope. This is the safer option for user experience, as the user always receives their tokens. The fallback address should be set to the Sickle owner's address so funds are recoverable. The current adapter logic in `onERC20Received()` already has internal fallback handling via the signature verification path in `BridgeDepositStrategy.executeDeposit()`, but the deBridge-level fallback provides an additional safety net at the protocol layer.

2. **Success-required**: If the hook reverts, the entire order fulfillment transaction reverts, and the order remains unfilled. In this case, the order's authority on the destination chain must be properly configured to allow cancellation, so the user can recover funds on the source chain. Without a proper cancellation path, user funds could become stuck.

Recommendation

Configure deBridge hooks as **atomic** to ensure execution happens within the order fulfillment transaction. For the success mode, prefer **success-optional** with the fallback address set to the Sickle owner, or if **success-required** is chosen, ensure the order's destination chain authority is properly configured to support cancellation. Document the chosen configuration and its implications for the frontend and order creation flow.

Developer Response

Fixed in commit 264509cf (PR#680, merge commit 298bdca2). @dev NatSpec was added to the deBridge adapters documenting the required hook configuration (atomic, success-optional with fallback to token recipient).

2.10.4 deBridge adapters do not use the `(bool, bytes)` callback result to report unexpected execution failures

Technical Details

Both `DeBridgeDepositAdapter.onERC20Received()` and `DeBridgeSwapAdapter.onERC20Received()` implement `IDeBridgeExternalCallExecutor`, whose callback interface explicitly returns `(bool callSucceeded, bytes memory callResult)`.
However, both adapters currently call into the downstream strategy/receiver and unconditionally return `(true, "")` on the success path, while any unexpected downstream revert simply bubbles up and reverts the whole callback.
The adapters bypass `deBridge`'s structured failure channel and bubble hard reverts instead, reducing compatibility with `deBridge`'s error handling.

Impact

Informational.

Recommendation

Wrap downstream `strategy.executeDeposit()` and `receiver.executeSwap()` calls in `try/catch` and return `(true, "")` on success and `(false, reason)` on failure, instead of bubbling hard reverts. This better matches the `IDeBridgeExternalCallExecutor` interface and lets the deBridge caller enforce `requireSuccessfullExecution` based on an explicit callback result.
On the source side, Sickle should also set `ExternalCallEnvelopV1.requireSuccessfulExecution = true` in the optional `OrderCreation._externalData` payload in order to make the deBridge flow revert.
To be fully compliant, it is recommended to add a fallback transfer in the new catch block. So, in case deBridge doesn't revert because `ExternalCallEnvelopV1.requireSuccessfulExecution` is `false`, tokens are still forwarded.

Developer Response

Fixed in PR#680 (merge commit 298bdca2). The accepted final form is in commit f8311a3c — fix: remove try catch pattern, always return true on success — try/catch was removed (a failed call would leave tokens stuck outside the adapters) but adapters still return (true, new bytes(0)) on success to match the IDeBridgeExternalCallExecutor interface.

2.10.5 `FarmDeposit` and `FarmIncrease` deposit modes are treated identically

Both `DepositMode.FarmDeposit` and `DepositMode.FarmIncrease` follow the exact same code path in `_executeDeposit()`, making the distinction unnecessary.

Technical Details

In `_executeDeposit()`, the condition `params.mode == DepositMode.FarmDeposit || params.mode == DepositMode.FarmIncrease` routes both modes into the same `_executeFarmDeposit()` function with no differentiation. In the original `FarmStrategy`, `deposit()` also handles Sickle deployment via `getOrDeploySickle()`, which is not relevant here since `bridgeDepositExternal()` already requires the Sickle to exist. The two enum variants add complexity without providing any behavioral difference.

Impact

Informational.

Recommendation

Consider merging `FarmDeposit` and `FarmIncrease` into a single `FarmDeposit` mode, or add a comment explaining why the distinction is preserved (e.g., for future differentiation or frontend clarity).

Developer Response

Fixed in PR#680 - commit 264509cf. The fix adds inline DepositMode documentation explaining that `FarmDeposit` and `FarmIncrease` share the same execution path in the bridge context (Sickle pre-exists, position settings are managed source-side).

2.10.6 `bridgeDepositExternal()` duplicates `getSickle()` logic

`bridgeDepositExternal()` manually looks up the sickle address and checks for zero instead of using the inherited `getSickle()` helper.

Technical Details

The function performs:

```
address sickleAddr = address(factory.sickles(sickleOwner));
if (sickleAddr == address(0)) revert NoSickle();
```

The `StrategyModule.getSickle()` function already encapsulates this exact pattern with its own `SickleNotDeployed` error.

Impact

Informational.

Recommendation

Replace the manual lookup with `Sickle sickle = getSickle(sickleOwner);` and remove the `NoSickle` custom error.

Developer Response

Fixed in PR#674 - commit `a05e9c72`. Removed duplicated `getSickle()` logic from `bridgeDepositExternal()`.

2.10.7 `_tryVerifySignature()` return logic can be simplified

The signature verification function uses an `if-return false-return true` pattern that can be expressed more concisely.

Technical Details

In `BridgeDepositStrategy._tryVerifySignature()`, the final check is:

```
if (err != ECDSA.RecoverError.NoError || recovered != owner) {
    return false;
}
return true;
```

This is equivalent to:

```
return err == ECDSA.RecoverError.NoError && recovered == owner;
```

Impact

Informational.

Recommendation

Replace the conditional with:
`return err == ECDSA.RecoverError.NoError && recovered == owner;`

Developer Response

Fixed in PR#674 - commit a05e9c72. Simplified `_tryVerifySignature()` return logic.

2.10.8 Intent ID not marked as used when `_bridgeDepositSafe()` fails

In `executeDeposit()`, the `intentId` is only marked as used when `_bridgeDepositSafe()` succeeds, leaving it reusable after a fallback transfer.

Technical Details

When `_bridgeDepositSafe()` catches a revert, it sends the bridged tokens directly to the sickle owner as a fallback. However, `usedIntentIds[sickleOwner][intentId]` is not set to `true` in this case. Since the tokens have already been bridged and transferred (either deposited or sent via fallback), the intent has effectively been consumed. Leaving it unmarked means the same `intentId` could theoretically be replayed if the owner signs the same parameters again, though in practice the tokens would no longer be available in the contract.

Impact

Informational. The intent should be considered consumed once tokens are transferred regardless of the deposit outcome. In practice, re-execution is unlikely to cause harm because the contract will not hold the same tokens again, but the state is semantically incorrect.

Recommendation

Mark `usedIntentIds[sickleOwner][intentId] = true` before calling `_bridgeDepositSafe()`.

Developer Response

Fixed in PR#674 - commit a05e9c72. Intent ID is now marked as used when `_bridgeDepositSafe()` fails.

2.10.9 Bridge withdraw strategies always charge fees regardless of swap presence

`BridgeWithdrawStrategy` unconditionally charges `BridgeWithdrawStrategyFees.Withdraw` fees during withdrawals, diverging from the behavior of the non-bridge `FarmStrategy` and `NftFarmStrategy` which only charge fees when zap swaps are present.

Technical Details

In `_farmWithdraw()`, fees are always charged via `IFeesLib.chargeFees()` with the `BridgeWithdrawStrategyFees.Withdraw` fee type regardless of whether `params.zap.swaps.length` is zero. The same pattern appears in `_nftZapOutAndFees()`, which always charges fees without checking `nftWithdraw.zap.swaps.length`.
In contrast, `FarmStrategy` only charges the withdraw fee when swaps are present in the zap, and `NftFarmStrategy` similarly gates fee charging behind

`nftWithdraw.zap.swaps.length > 0`. This means users performing a bridge withdrawal without swaps are charged a fee they would not incur through the non-bridge path.

Impact

Informational.

Recommendation

Align the fee-charging logic with `FarmStrategy` and `NftFarmStrategy` by only calling `chargeFees()` when `swaps.length > 0`, or document the intentional behavioral difference.

Developer Response

Fixed in PR#678 in commit `44aa74fe` — withdraw fees are now only charged when swaps are present, aligned with FarmStrategy/NftFarmStrategy behavior.

2.10.10 `NftHarvest` sweep tokens silently ignored in `_nftHarvestForBridge()`

The `NftHarvest` struct's `sweepTokens` field is silently ignored when harvesting for a bridge operation, which could lead to unexpected behavior if a caller provides non-empty sweep tokens.

Technical Details

In `BridgeWithdrawStrategy._nftHarvestForBridge()`, the `NftHarvest` parameter contains a `sweepTokens` array that is never referenced. This is intentional because the harvest result stays in the Sickle to be bridged rather than swept to the user. However, a caller could mistakenly pass non-empty `sweepTokens` expecting them to be transferred, and the function would silently succeed without sweeping.

Impact

Informational.

Recommendation

Add a check that `h.sweepTokens.length == 0` at the start of `_nftHarvestForBridge()` and revert if it is not, making the expectation explicit.

Developer Response

Fixed in PR#678 - commit 44aa74fe. As discussed in this thread, the final behavior sweeps harvestParams.sweepTokens to the user (matching the non-bridge _nftHarvest) instead of reverting.

2.10.11 BridgeLib inherits unused DelegateModule mixin

`BridgeLib` inherits `DelegateModule` but does not use any of its functionality.

Technical Details

`BridgeLib` inherits `DelegateModule`, but never calls `_delegateTo()`, the only function provided by the mixin.

Impact

Informational.

Recommendation

Remove the `DelegateModule` inheritance from `BridgeLib`.

Developer Response

Fixed in PR#674 - commit `a05e9c72`. Removed unused `DelegateModule` inheritance from `BridgeLib`.

2.10.12 Migrate Wormhole adapters from Standard Relay to Executor Relay

Technical Details

The deployment plans still configure `WormholeDepositAdapter` and `WormholeSwapAdapter` to trust the legacy `WormholeRelayer` address `0x27428DD2d3DD32A4D7f7C497eAaa23130d894911` as their whitelisted bridge endpoint.
Examples:

- `script/deployments/1/plan.json`
- `script/deployments/42161/plan.json`

Wormhole announced on March 17, 2026 that Standard Relaying was deprecated and that automatic delivery shuts down on April 1, 2026 (see here). Integrations built around `receiveWormholeMessages` and the Standard Relayer need to migrate to the newer Executor-based flow.
As configured today, the Wormhole adapter paths in this repository remain tied to the legacy relayer endpoint.

Impact

Informational.

Recommendation

Migrate Wormhole integrations away from the Standard Relayer to Wormhole's Executor flow. Update deployment plans, trusted endpoint configuration, and any receiver-side assumptions tied to `receiveWormholeMessages()` and Standard Relayer delivery semantics:

> Key steps for GMP integrators:
>
> - Review your source-chain contract. Does it call IWormholeRelayer.sendPayloadToEvm? You'll need to replace this with the Executor's requestExecution pattern.
> - Review your destination-chain contract. The entry point changes; instead of receiveWormholeMessages, your contract now receives a VAA via executeVaaV1 from the Relay Provider and verifies it through parseAndVerifyVM on the Wormhole Core contract.
> - Implement quote fetching. Before sending, your client (or contract, if using on-chain quotes) must fetch a signed execution quote and include it in the transaction.
> - Generate relay instructions. These define destination gas limits, message value, and any gas drop-off behavior. They're serialized using the Wormhole TypeScript SDK and passed to the Executor contract.

Developer Response

Fixed in PR#682 - commit `8fbc2532`. Adapters now implement `executeVAAv1` instead of the deprecated `receiveWormholeMessages` entry point.

2.11 Final Remarks

The bridge extension is broadly well-structured, but its safety depends heavily on getting bridge-specific trust boundaries exactly right. The strongest concerns are concentrated in integration edges rather than in core strategy logic: legacy Wormhole assumptions, callback payload coupling, manual destination-side recovery behavior, and mismatches between protocol callback semantics and local adapter expectations.

yAudit

Sickle Bridge Strategies

Completed 2026-04-08