yAudit

June 2026

Prepared for
Blend

Audited by
panda
adriro

AllowlistedSwapAdapter Review

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 . . . 2
1.4 Key Findings . . . 3
1.5 Overall Assessment . . . 3

2 Audit Overview **3**
2.1 Project Information . . . 3
2.2 Audit Timeline . . . 3
2.3 Audit Resources . . . 3
2.4 Critical Findings . . . 4
2.5 High Findings . . . 4
2.6 Medium Findings . . . 4
2.7 Low Findings . . . 4
2.8 Gas Savings Findings . . . 4
2.9 Informational Findings . . . 4
2.9.1 Swap recipients receive stale adapter output balances . . . 4
2.9.2 Permissionless dusting can brick exact-input AllowlistedSwapAdapter swaps until owner sweep . . . 5
2.9.3 Persistent approvals to permissionless executors can expose later adapter balances . . . 7
2.9.4 Allowlisting Bundler3 lets any EOA bypass AllowlistedSwapAdapter's caller gate . . . 8
2.9.5 Allowlisting user Safes gives them arbitrary withdrawal power over adapter-held balances . . . 9

1 Review Summary

1.1 Protocol Overview

Blend is a non-custodial Yield Coordination Engine built on the Separately Managed Account (SMA) model. Users get their own isolated Gnosis Safe without shared pools or mixing of funds.

1.2 Audit Scope

This audit covers the AllowlistedSwapAdapter contract across half a day of review.

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	0
Informational	5

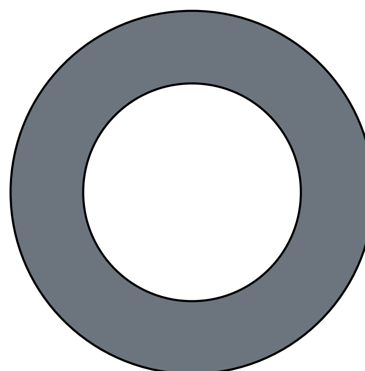


Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The review of Blend's AllowlistedSwapAdapter did not identify any critical, high, medium, or low-severity vulnerabilities. All findings were informational and centered on the adapter's zero-balance operating assumption. The team acknowledged these trust-model constraints and added NatSpec documentation clarifying the intended caller model and treatment of unsolicited balances.

2 Audit Overview

2.1 Project Information

Protocol Name: Blend
Repository: https://github.com/BlendMoney/contracts
Commit Hash: b1f6dcef09f8ff06fde708ff9715babc1ae4f9d6
Final Commit Hash:
Commit URL: https://github.com/BlendMoney/contracts/pull/192/changes/b1f6dcef09f8ff06fde708ff9715babc1ae4f9d6

2.2 Audit Timeline

The audit was conducted on June 26, 2026.

2.3 Audit Resources

- Code repositories and documentation

2.4 Critical Findings

None.

2.5 High Findings

None.

2.6 Medium Findings

None.

2.7 Low Findings

None.

2.8 Gas Savings Findings

None.

2.9 Informational Findings

2.9.1 Swap recipients receive stale adapter output balances

`SwapAdapter` measures swap output as a balance delta, but transfers the adapter's full post-swap output-token balance to the caller-selected recipient. Any output-token balance already held by the adapter is paid out with the fresh swap output.

Technical Details

In `swapToCollateral()`, the adapter snapshots `collateralToken.balanceOf(address(this))` before executing the caller-provided swap bundle. After the calls finish, it calculates `amountOut` as `collateralBalanceAfterSwap - balanceBefore`, which correctly excludes any pre-existing collateral balance from the slippage and event accounting.
The transfer does not use that delta. Instead, `swapToCollateral()` sends `collateralBalanceAfterSwap` to `recipient`, so the recipient receives both the new swap output and any collateral tokens that were already sitting on the adapter. The same pattern exists in `swapToLoanToken()`, where `amountOut` is calculated as `loanBalanceAfterSwap - balanceBefore` but `loanBalanceAfterSwap` is transferred.
This affects unsolicited donations, dust, or intermediate tokens left behind by prior routes if those tokens later become the output token of a successful swap. In `AllowlistedSwapAdapter`, any allowlisted swap caller can capture those balances through an otherwise valid swap, bypassing the owner-only `sweepToken()` recovery path for the affected token.

Impact

Informational. A swap recipient can receive stale adapter balances in addition to the intended swap output. The value at risk is limited to tokens already held by the adapter outside the active swap, such as donated funds, dust, or stranded intermediate assets. Normal swap input funds are still subject to the route execution and post-swap invariant checks, so this primarily weakens balance recovery and isolation between users of a shared adapter.

Recommendation

Transfer only the computed output delta to `recipient`, not the full post-swap output-token balance. Alternatively, require the relevant output-token balance to be zero before executing the route, so stale balances remain recoverable only through the owner-controlled sweep path. Apply the same invariant to both `collateralToken` in `swapToCollateral()` and `loanToken` in `swapToLoanToken()`.

Developer Response

Acknowledged as informational. These findings share a common premise: tokens sitting on the adapter between transactions and the sweep function as a recovery path.
The adapter is designed to hold zero balance between transactions. sweepToken is a best-effort convenience for recovering unsolicited transfers, not a security boundary. Unsolicited transfers to the contract are outside the protocol's scope.
This is a non-custodial system. Users own their Safes and must be able to withdraw funds at any time, regardless of executor state. Safe owners interacting with the adapter directly is by design, not a flaw.
Every allowlisted address has gone through Blend's onboarding process. The allowlist is an operational gate for known parties, not a defense against anonymous adversaries.
Added trust model documentation to the contract NatSpec in PR #195.

2.9.2 Permissionless dusting can brick exact-input AllowlistedSwapAdapter swaps until owner sweep

Technical Details

`SwapAdapter` infers the swap input amount by reading `primary.balanceOf(address(this))` at the start of each swap operation rather than accepting an explicit `amountIn` parameter. The `checkSwapInvariants()` modifier enforces a strict postcondition requiring both primary and secondary token balances to be zero after swap completion. While `AllowlistedSwapAdapter` restricts who can call `swapToCollateral()` and `swapToLoanToken()`, it does not prevent unsolicited ERC20 transfers to the adapter contract.
The integrated flow prefunds the adapter before invoking swap entrypoints. When a third party sends dust of the primary token to the adapter, the balance-based input calculation in `SwapAdapter.sol:279-324` and `SwapAdapter.sol:344-389` treats that dust as part of the intended swap amount. Exact-input routes encode a fixed `amountIn` in their router calldata (e.g., Uniswap V3 `exactInputSingle`), so the router consumes only the original intended amount while the adapter expects the full balance—including dust—to be consumed.
This mismatch causes the transaction to revert with `AdapterNotEmpty()` when dust remains unswapped, or with `SlippageExceeded()` if the route attempts to consume the inflated balance

and produces output below the minimum threshold calculated from that larger input. Because the victim transaction reverts atomically, the dust persists on the adapter and blocks subsequent swaps until the owner calls `sweepToken()`. The attacker can re-poison the adapter at negligible cost before each execution.
The `checkSwapInvariants()` modifier at `SwapAdapter.sol:196-214` performs the zero-balance check that triggers the revert. `MorphoSeedingLib.sol:58-66,139-147` and `MorphoLeverageLib.sol:80-94,224-255` demonstrate the prefund-then-swap lifecycle where tokens are transferred to the adapter immediately before the swap call, creating a window during which the adapter holds both the intended balance and any unsolicited dust.

Impact

Informational. A public attacker can send 1 wei of the swap's primary token to the adapter and make later exact-input swap-driven rebalances or vault actions revert until the dust is swept or the route is rebuilt to consume the full live balance. This denial of service affects only routes whose calldata does not dynamically consume the adapter's full balance, yet those routes match the common integration pattern present in the codebase. The attack is repeatable and inexpensive, enabling persistent griefing that can delay time-sensitive deleveraging, rebalancing, or asset rotation operations during market moves.

Recommendation

Separate intended swap amount from incidental token balance. Add an explicit `amountIn` parameter to `swapToCollateral()` and `swapToLoanToken()` entrypoints and verify the adapter received at least that amount before execution. Execute the swap against the specified `amountIn` and update the postcondition to ensure any remaining primary balance does not exceed the pre-swap balance plus a small tolerated dust threshold, or refund unexpected excess to the recipient.
Alternatively, if full-balance semantics are intentionally retained, snapshot the primary token balance before the swap and enforce cleanup relative to that baseline rather than requiring an absolute zero final balance. This approach prevents unsolicited transfers from changing swap semantics or triggering false `AdapterNotEmpty()` reverts while preserving the strict cleanup invariant for protocol-controlled flows.

Developer Response

Acknowledged as informational. These findings share a common premise: tokens sitting on the adapter between transactions and the sweep function as a recovery path.
The adapter is designed to hold zero balance between transactions. sweepToken is a best-effort convenience for recovering unsolicited transfers, not a security boundary. Unsolicited transfers to the contract are outside the protocol's scope.
This is a non-custodial system. Users own their Safes and must be able to withdraw funds at any time, regardless of executor state. Safe owners interacting with the adapter directly is by design, not a flaw.
Every allowlisted address has gone through Blend's onboarding process. The allowlist is an operational gate for known parties, not a defense against anonymous adversaries.
Added trust model documentation to the contract NatSpec in PR #195.

2.9.3 Persistent approvals to permissionless executors can expose later adapter balances

Technical Details

The `SwapAdapter` contract executes arbitrary call bundles through `_executeSwapCalls()` at `src/adapters/SwapAdapter.sol:405-423` without tracking or revoking token approvals created during those calls. Both `swapToCollateral()` and `swapToLoanToken()` delegate to this function, which mirrors the `Bundler3._multicall()` pattern by iterating over a `Call[]` array and performing raw external calls.
The adapter is designed to support sophisticated swap routes where an allowlisted caller can supply `extraData` containing arbitrary call sequences. One supported pattern demonstrated in the codebase involves the adapter first executing `IERC20.approve(bundler3Address, type(uint256).max)` to authorize a permissionless dispatcher contract, then calling that dispatcher's `multicall()` function to perform the actual swap logic.
Because `Bundler3.multicall()` at `lib/bundler3/src/Bundler3.sol:22-34` is permissionless and accepts arbitrary call bundles from any caller, the approval granted during the swap remains exploitable after the transaction completes. Any user can later call `Bundler3.multicall()` with a single instruction to execute `token.transferFrom(adapter, attacker, balance)`, and because the adapter previously approved Bundler3 as a spender, the transfer succeeds.
The `AllowlistedSwapAdapter` extends `SwapAdapter` and gates the swap entry points with `onlyAllowlisted` at `src/adapters/AllowlistedSwapAdapter.sol:231-255`, but once an allowlisted caller or trusted execution path submits swap calldata that includes such an approval, the resulting spend authority persists indefinitely. Normal swap flows are designed to prefund and empty the adapter atomically, enforced by `checkSwapInvariants`, so the main same-transaction balances are not exposed. The residual risk applies to any token balance that arrives on the adapter across transactions, including accidental transfers, intermediate tokens from multi-hop swaps, or other recoverable balances that `sweepToken()` is meant to reclaim.

Impact

Informational. Once a swap route leaves an approval from the adapter to a permissionless executor, later balances of that token can be drained by arbitrary third parties without requiring allowlist authorization or owner privileges. The owner-only `sweepToken()` function becomes ineffective for the approved token, as external users can front-run or independently drain those balances through the approved executor. The realistic value at risk consists of residual, stranded, or accidentally transferred tokens rather than the primary swap amounts, which are typically transferred out in the same transaction. Shared adapters reused across multiple strategies amplify the exposure, as a single unsafe route approval affects all future balances of that token on the adapter.

Recommendation

Prevent third-party spend approvals from persisting after a swap completes. Explicitly revoke any approval granted during the call bundle before returning from the swap entry point, or route spend authority through a helper contract that enforces exact-amount approvals and always resets them to zero, including on failure paths. If arbitrary call bundles remain necessary, reject approvals to generic permissionless executors such as `Bundler3` by maintaining an allowlist of approved call targets and validating each target before execution. At minimum,

document that any swap route approving a permissionless dispatcher must revoke that approval within the same transaction to avoid creating a permanent drain vector.

Developer Response

Acknowledged as informational. These findings share a common premise: tokens sitting on the adapter between transactions and the sweep function as a recovery path.
The adapter is designed to hold zero balance between transactions. sweepToken is a best-effort convenience for recovering unsolicited transfers, not a security boundary. Unsolicited transfers to the contract are outside the protocol's scope.
This is a non-custodial system. Users own their Safes and must be able to withdraw funds at any time, regardless of executor state. Safe owners interacting with the adapter directly is by design, not a flaw.
Every allowlisted address has gone through Blend's onboarding process. The allowlist is an operational gate for known parties, not a defense against anonymous adversaries.
Added trust model documentation to the contract NatSpec in PR #195.

2.9.4 Allowlisting Bundler3 lets any EOA bypass AllowlistedSwapAdapter's caller gate

Technical Details

`AllowlistedSwapAdapter` enforces access control using an owner-managed allowlist. Both `swapToCollateral()` and `swapToLoanToken()` apply the `onlyAllowlisted` modifier, which checks `allowlist[msg.sender]`. This authentication model works correctly for direct callers but fails when the adapter is invoked through a permissionless forwarder.
The protocol's leveraged rebalance paths in `MorphoLeverageLib` construct swap calls using `CallBuilder.swapToCollateral()` and `CallBuilder.swapToLoanToken()`, then execute them via `Bundler3.multicall()`. For an `AllowlistedSwapAdapter` deployment to function in these flows, the shared `Bundler3` contract must be added to the allowlist.
`Bundler3` is explicitly permissionless: any EOA can invoke `multicall()` and supply arbitrary call bundles. The contract stores the real caller in a transient `initiator` variable but never exposes it to downstream targets. When `Bundler3` forwards a call to `AllowlistedSwapAdapter`, `msg.sender` is the `Bundler3` address itself, not the original EOA.
Once `Bundler3` appears on the allowlist, any EOA can call `Bundler3.multicall()` with a bundle targeting `AllowlistedSwapAdapter.swapToCollateral()` or `swapToLoanToken()`. The adapter's `onlyAllowlisted` check sees `msg.sender == Bundler3`, which is allowlisted, and permits the call even though the EOA initiating the transaction was never approved. The adapter authenticates the forwarder contract, not the real user.
Because `SwapAdapter` accepts attacker-controlled `recipient`, `strategyData`, and `extraData` parameters—including an arbitrary `Call[]` bundle—an unauthorized caller can direct swap outputs to any address and execute arbitrary token transfers using balances held by the adapter.

Impact

Informational. If a deployment adds a permissionless forwarder such as `Bundler3` to the allowlist, any EOA can invoke the adapter's swap entrypoints with attacker-chosen parameters. This can drain token balances sitting on the adapter, including residual intermediate tokens or mistakenly prefunded amounts. The protocol's normal flows generally fund and empty the

adapter atomically within authorized transactions, so the practical victim is limited to balances that remain on the adapter outside those atomic operations. The adapter's intended access-control property—restricting swap execution to owner-approved addresses—is bypassed for any deployment that allowlists a permissionless forwarder.

Recommendation

Do not allowlist permissionless forwarders on this adapter. For execution paths that require `Bundler3`, use an adapter designed for execution-context gating, such as `WhitelistedSwapAdapter`, or add explicit trusted-forwarder handling that reads the forwarder's underlying initiator and requires that initiator to be allowlisted.
A robust fix would maintain a separate set of trusted forwarders. When `msg.sender` is a known forwarder, query its `initiator()` and require that address to be allowlisted; otherwise, require `msg.sender` itself to be allowlisted. As a defense-in-depth measure, reject known public forwarders such as `Bundler3` from `toggleAllowlist()` to prevent accidental misconfiguration. Add regression tests confirming that allowlisting `Bundler3` alone does not permit an arbitrary EOA to use the adapter, and that only the underlying initiator satisfies the allowlist when calls are forwarded.

Developer Response

Acknowledged as informational. We will never allowlist a permissionless forwarder such as Bundler3. The contract is purpose-built for direct callers, and this is documented in the trust model NatSpec added in PR #195.

2.9.5 Allowlisting user Safes gives them arbitrary withdrawal power over adapter-held balances

Technical Details

`AllowlistedSwapAdapter` restricts its swap entrypoints using only `allowlist[msg.sender]` at `src/adapters/AllowlistedSwapAdapter.sol:225-255`. In the repository's direct-call integrations (`MorphoSeedingLib` and `VaultToVaultAction`), swap operations execute via `delegatecall` inside user-controlled Safe contracts. This makes the Safe itself the immediate caller at runtime, requiring each participating Safe to be added to the adapter's allowlist for swaps to function.
Once a user Safe is allowlisted, its owners can submit direct transactions to the adapter outside the intended `StrategyManager` flow. The adapter's `swapToCollateral()` and `swapToLoanToken()` functions decode oracle configuration and maximum slippage from caller-supplied `strategyData`, then execute arbitrary `Call[]` bundles from caller-supplied `extraData`. Because `MAX_SLIPPAGE_BPS` permits up to 10,000 basis points (100%), an allowlisted Safe owner can set both maximum and runtime slippage to 100%, reducing `amountOutMin` to zero. The attacker can then craft a swap bundle that calls the held token contract directly to transfer the adapter's entire balance to an arbitrary recipient. Post-swap invariants in `SwapAdapter.checkSwapInvariants()` at `src/adapters/SwapAdapter.sol:184-213` pass as long as the primary token balance becomes zero and the secondary token balance remains zero.
A simpler drain path exists when `primary == secondary`: the `checkSwapInvariants` modifier immediately transfers the adapter's full balance of that token to the caller-specified recipient

and emits `SwapSameAsset`, bypassing swap execution entirely. Both paths convert the adapter's generic swap interface into a user-controlled withdrawal primitive.

Impact

Informational. Any allowlisted Safe owner can extract the adapter's entire balance of a given token, bypassing the owner-only `sweepToken()` recovery mechanism. Because a single `AllowlistedSwapAdapter` instance can be shared across multiple Safes and strategies, this lets one Safe owner claim balances that may have originated from other Safes, such as accidentally transferred tokens or leftover intermediate-hop tokens. Normal successful swaps are designed to empty the adapter of source and destination tokens atomically, so the exploit does not steal in-flight user funds during legitimate operations. However, any value resting on the adapter between transactions becomes accessible to all allowlisted Safe owners, effectively granting them a privileged sweep capability over a shared resource.

Recommendation

Do not allowlist user-controlled Safes or other untrusted callers on `AllowlistedSwapAdapter`. For Safe-based integrations like `MorphoSeedingLib` or `VaultToVaultAction`, use `WhitelistedSwapAdapter` instead, which gates by execution context rather than `msg.sender`. Alternatively, deploy a thin protocol-owned wrapper contract that is the sole allowlisted address and enforces that swaps are callable only from trusted controllers or modules, not from user-controlled Safes.

If direct Safe callers must remain supported, store oracle references and maximum slippage in protocol-controlled configuration rather than accepting them from caller-supplied `strategyData`. Additionally, validate and sanitize the `Call[]` array from `extraData` to ensure swap bundles cannot perform arbitrary token transfers. Disabling same-asset swaps alone is insufficient, as the broader issue stems from unrestricted call data and slippage control.

Developer Response

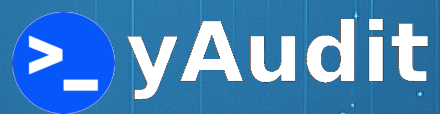
Acknowledged as informational. These findings share a common premise: tokens sitting on the adapter between transactions and the sweep function as a recovery path.

The adapter is designed to hold zero balance between transactions. sweepToken is a best-effort convenience for recovering unsolicited transfers, not a security boundary. Unsolicited transfers to the contract are outside the protocol's scope.

This is a non-custodial system. Users own their Safes and must be able to withdraw funds at any time, regardless of executor state. Safe owners interacting with the adapter directly is by design, not a flaw.

Every allowlisted address has gone through Blend's onboarding process. The allowlist is an operational gate for known parties, not a defense against anonymous adversaries.

Added trust model documentation to the contract NatSpec in PR #195.



AllowlistedSwapAdapter Review

Completed 2026-06-26