Morpho SDKs
Security Review

Cantina Managed review by:

M4rio.eth, Lead Security Researcher
Slowfi, Security Researcher

June 26, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Morpho is a trustless and efficient lending primitive with permissionless market creation.

From May 20th to Jun 2nd the Cantina team conducted a review of morpho-sdks on commit hash 3ba8fd4a. The team identified a total of **17** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	0	1
Low Risk	11	4	7
Gas Optimizations	0	0	0
Informational	5	4	1
Total	**17**	**8**	**9**

3 Findings

3.1 Medium Risk

3.1.1 Missing version pinning could result in supply chain attacks

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: All `morpho-sdk` packages are published with caret (`^`) version ranges in `dependencies`, `peerDependencies`, and the `workspace:^` placeholders that are expanded during publishing:

```
"dependencies": { "mutative": "^1.1.0" },
"peerDependencies": {
  "@morpho-org/blue-sdk": "workspace:^",
  "@morpho-org/morpho-ts": "workspace:^"
}
```

Caret ranges allow any newer minor or patch release with the same major version to be installed. This means that:

- A fresh install, a partially-regenerated lock-file, or the use of npm/yarn instead of pnpm can silently pick up newer code.
- Down-stream projects that depend on these SDKs but maintain their own lock-files will resolve the latest matching versions.

The result is a supply-chain risk: a faulty or malicious minor update of `blue-sdk`, `morpho-ts`, or another dependency could change off-chain bundle logic, redirect approvals, or introduce other unsafe behavior without any code change in the integrator's repository.

Recommendation:

1. Replace caret (`^`) and `workspace:^` ranges with exact versions in `dependencies` and `peerDependencies`—for example:

```
"@morpho-org/blue-sdk": "2.0.0"
```

2. Bump these versions explicitly when a new release is reviewed and accepted.
3. Keep the lock-file committed, but treat it as a secondary safeguard.
4. Add a CI check that fails if a published `package.json` still contains `^`, `~`, or `workspace:` ranges.
5. Optionally include the `packageManager` field (`"pnpm@9.x"`) to discourage installation with other clients.

Pinning versions ensures every dependency update undergoes review, closing the silent supply-chain upgrade path.

Morpho: Acknowledged.

Cantina Managed: Acknowledged.

3.2 Low Risk

3.2.1 Missing checks throughout the `simulation-sdk`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `simulation-sdk` represents a dry-run of a suit of operations that would be bundled together in the `Bundler3`. In theory the simulator should mimic the onchain behavior, including all the checks plus should add extra checks that would protect the user from a mistake in his endavor to bundle multiple transactions at once. We've enumerated the checks as follow:

- The `slippage` parameter is used in various operations but we do not have any check to make sure this slippage is not negative or it's over the WAD value. E.g. SimulationState.ts#L358 in the `getBundleMaxBalance` where we apply the slippage when we convert to wrapped tokens. This is just one instance, the slippage parameter is used kinda everywhere across the simulation.

- When we borrow/withdraw/withdrawCollateral we should revert if the sender it's not authorized to modify `onBehalf`'s position, we only do this if the general adapter is the sender. E.g. borrow.ts#L24-L29.
- The `permit2` simulation does not check if the expiration is lower than `block.timestamp` permit2.ts#L14.

Recommendation: Consider adding these extra checks. Furthermore, while we tried to highlight some of the missing checks, we recommend to go over the onchain code and the simulation and make sure all the checks are implemented in the simulation as well.

Morpho: Fixed on PR 351.

Cantina Managed: Fix verified.

3.2.2 Reallocating assets should be conservative when we look at the vault's cap

Severity: Low Risk

Context: SimulationState.ts#L590-L594

Finding Description: The `getMarketPublicReallocations` function calculates the public reallocations required to reach the maximum available liquidity, based on a specific reallocation algorithm. This algorithm is implemented in `_getMarketPublicReallocations` and operates by iterating through available vaults and their withdrawal queues. It calculates the maximum amount that can be safely reallocated from each source market to a destination market, while respecting various constraints such as market caps, utilization targets, and configured limits. For each vault, the algorithm identifies the most liquid source market (i.e., the one with the highest available assets for withdrawal) and creates a reallocation operation to move those assets to the destination market.

The algorithm sorts vaults by their reallocatable liquidity in descending order and recursively processes these reallocations, simulating each operation to ensure the system remains within its defined parameters. The process includes checks for market caps, utilization rates, maximum inflow/outflow limits, and accrued interest, ultimately optimizing liquidity distribution across the protocol. To conservatively compute the reallocatable assets, interest is accrued one hour in advance:

```
const suppliable =
  cap -
  data
    .getAccrualPosition(vault, marketId)
    .accrueInterest(this.block.timestamp + delay).supplyAssets;
```

We can see that we are missing an important aspect: if a `pendingCap` is valid within the next hour, it is not currently included in the algorithm and we are using the current cap that is active. A market may have a cap that is pending and set to be applied after a specific duration (MetaMorpho.sol#L472-L480).

This would cause the algorithm to inefficient calculate the assets that could be reallocated if a cap that is lower than the:

- `srcPosition.supplyAssets`.
- `targetUtilizationLiquidity`.
- `publicAllocatorConfig?.maxIn`.
- `publicAllocatorConfig?.maxOut`.

In the case of the bundler, the transaction would revert if the pending cap will be applied because the cap would be lower than what we try to reallocate, so no funds would be lost, but we must treat the simulation as an sdk that could be used in various usecase.

Recommendation: Consider using `min(cap, pendingCap)` if the `pendingCap` will be available within the next hour.

Morpho: Fixed on PR 351.

Cantina Managed: Fix verified.

3.2.3 The holdings do not exclude tokens that can not be transferred

Severity: Low Risk

Context: SimulationState.ts#L340

Finding Description: The `getHolding` function returns a snapshot of the current holdings of a specific token for a user. The `Holding` object includes the following field:

```
/**
 * Whether the user is allowed to transfer this holding's balance.
 */
public canTransfer?: boolean;
```

This indicates whether the user is allowed to transfer the balance of the holding. Currently, this field is not taken into account, and the holding is considered valid even if the user cannot transfer it. This may result in the simulation counting an underfunded balance for the user. This function is used across various logic, so the impact can vary depending on the use case.

Recommendation: Consider excluding from holdings any tokens marked as non-transferrable. After discussing with the team, it was acknowledged that this is typically used for permissioned tokens. However, we still recommend adding a flag to optionally exclude such tokens when a specific call requires it.

Morpho: Acknowledged.

Cantina Managed: Acknowledged.

3.2.4 Single slippage value used for different conversions

Severity: Low Risk

Context: SimulationState.ts#L472

Description: `SimulationState.getBundleAssetBalances` uses a single `slippage` value through every conversion that can occur while estimating the maximum amount of a target token. For example, when the target is `wstETH` the helper may chain up to four different transformations:

1. wstETH → wstETH (direct balance).
2. stETH → wstETH.
3. ETH → stETH → wstETH.
4. WETH → ETH → stETH → wstETH.

Each hop has its own exchange rate and liquidity profile, yet the same slippage tolerance used at every step. A single parameter cannot capture the risk distribution across:

- Protocol-native wraps / unwraps (stETH ↔ wstETH).
- Staking contracts (ETH → stETH).
- ERC-20 wrappers.

Consequences:

- Max-balance calculations may be overly optimistic on one leg and overly pessimistic on another, leading to:
  - Bundles that revert on-chain because the real output is smaller than simulated.
  - Missed liquidity opportunities when the simulator underestimates what can be supplied or withdrawn.
- Future support for ERC-4626 wrappers or other multi-hop paths would inherit the same inaccuracy.

Recommendation:

1. Replace the single `slippage` argument with a structure that lets callers specify per-hop tolerances, e.g.:

```
type HopSlippage = {
  unwrapNative?: bigint; // WETH -> ETH
  stake?: bigint;        // ETH -> stETH
  wrapStaked?: bigint;   // stETH -> wstETH
  generic?: bigint;      // fallback
}
```

2. Internally, apply the appropriate value at each conversion step. If none is provided, fall back to `generic` or a conservative default.
3. Expose a convenience helper that translates "overall" slippage to a safe set of hop-specific values for simple use-cases.

Morpho: Acknowledged. We will apply fixes to mitigate it but it is not a priority for now.

Cantina Managed: Acknowledged.

3.2.5 `Blue_Paraswap_BuyDebt` ignores `ParaswapAdapter` `limitAmount`

Severity: Low Risk

Context: buyDebt.ts#L50-L61

Description: In `handlers/blue/buyDebt.ts` the swap branch pulls `exactAmount` and `quotedAmount` from the ParaSwap calldata but completely skips the `limitAmount` field:

```
const exactAmountOffset   = Number(offsets.exactAmount);
const quotedAmountOffset  = Number(offsets.quotedAmount);
// limitAmountOffset is never read
// ...
amount        = hexToBigInt(slice(args.swap.data, exactAmountOffset,  exactAmountOffset  + 32));
quotedAmount  = hexToBigInt(slice(args.swap.data, quotedAmountOffset, quotedAmountOffset + 32));
```

On-chain, `ParaswapAdapter.swap()` treats `limitAmount` (taken from offsets.limitAmount) as the maximum srcToken the trade is allowed to spend and reverts if that cap is exceeded:

- `ParaswapAdapter.sol::buy`:

```
swap({
          augustus: augustus,
          callData: callData,
          srcToken: srcToken,
          destToken: destToken,
          maxSrcAmount: callData.get(offsets.limitAmount),
          minDestAmount: callData.get(offsets.exactAmount),
          receiver: receiver
});
```

By ignoring it the simulator:

- May treat swaps as valid even when the calldata would revert on-chain because the price has moved past the user's limit.
- Cannot warn when a bundle spends more `srcToken` than authorised.

Recommendation:

1. In `buyDebt.ts` parse `offsets.limitAmount` exactly as done for `exactAmount`/`quotedAmount`.
2. When you want to perform the synthetic buy use `limitAmount` as the max amount that gets "burned".

Morpho: Acknowledged.

Cantina Managed: Acknowledged.

3.2.6 Missing invariant checks after a bundler being successfully executed could result in dangling approvals or residual amounts

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `Bundler3` today guarantees that each encoded operation succeeds, but it does not enforce any post-execution invariant that user assets (balances + approvals) are actually flushed back to the user. If a bundle finishes while:

- An ERC-20 allowance to `bundler3` is still > `0`.
- Any token balance (ETH or ERC-20) involved in the current batch remains on `Bundler3` or on one of its adapters.

Then those approvals / tokens stay trapped until the user manually rescues them — a pattern that could produce losses for the user. Because we optimise the bundle off-chain, a faulty optimisation, an unexpected callback branch, or a new adapter that is forgotten in `finalizeBundle` could silently leave residual value.

Recommendation: * Add a "safety-mode" flag (e.g. `requireClean = true`) in the SDK. When set, the encoder should automatically append a single invariant-check call as the final step of the bundle.

- Implement a minimal "InvariantChecker" function (could live inside an adapter) that reverts unless:

```
// for every token involved in this bundle transaction
IERC20(tokenX).allowance(user, address(bundler3)) == 0 &&
IERC20(tokenX).balanceOf(address(bundler3))                == 0 &&
IERC20(tokenX).balanceOf(address(adapterX))                == 0 // for every known adapter
```

  and `address(bundler3).balance before - after == 0`.

Morpho: Acknowledged.

Cantina Managed: Acknowledged.

3.2.7 Incorrect Encoding When Both `assets` and `shares` Are Non-Zero in `morphoSupply`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `morphoSupply` function within `BundlerAction.ts` constructs calldata assuming that both `assets` and `shares` can be non-zero. However, according to the `GeneralAdapter1` implementation, providing both as non-zero results in a revert with `inconsistent input`. This inconsistency arises because the underlying Morpho protocol expects either `assets` or `shares` to be set, but not both. Currently, the SDK does not validate this constraint, which leads to malformed calldata and execution reverts during simulation or bundling.

Proof Of Concept: Given a test case with the following inputs:

```
{
  type: "Blue_Supply",
  sender: client.account.address,
  address: morpho,
  args: {
    idMarket,
    assets: amount,
    shares: amount,
    onBehalf: client.account.address,
    slippage: DEFAULT_SLIPPAGE_TOLERANCE,
  },
}
```

The resulting bundle leads to a revert with:

```
Execution reverted with reason: revert: inconsistent input
```

This occurs during a `morphoSupply` call to the `generalAdapter1` contract with both `assets` and `shares` set.

Recommendation: Consider enforcing validation in the SDK logic (e.g., in `populateSubBundle` or before encoding the `Blue_Supply` operation) to ensure that **either** `assets` or `shares` is zero before proceeding. An early guard clause or explicit invariant check can help prevent user or developer mistakes and reduce noise during testing. Alternatively, handle this condition gracefully by choosing one value to override (e.g., zeroing `shares` if `assets` is set) or returning a meaningful error from the SDK itself when both are non-zero.

Morpho: Acknowledged.

Cantina Managed: Acknowledged.

3.2.8 Long Fixed Deadline Used for Signature-Based Operations

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In `packages/bundler-sdk-viem/src/actions.ts`, the `deadline` for signature-based operations (e.g., Permit2) is hardcoded to 24 hours:

```
const deadline = Time.timestamp() + Time.s.from.h(24n);
```

This is used uniformly across the bundler SDK without consideration of the type of operation, user intent, or contextual sensitivity of the transaction. Setting such a long, fixed deadline may expose users to unnecessary risks in scenarios where:

- The user expects a short-lived permit or allowance.
- An operation could be front-run or reused within the deadline window.
- The user's risk profile would benefit from shorter expiry (e.g., interactive UIs or automated flows).

Recommendation: Consider to allow finer control over the `deadline` parameter:

- Let it be passed as an optional argument, defaulting to a conservative value (e.g., 5–15 minutes).
- Fine-tune deadlines based on the operation type (e.g., short for `transferFrom`, longer for batched `supply`).
- Optionally document the rationale behind any defaults chosen.

Reducing deadline duration or making it configurable may improve the security posture and flexibility of the SDK.

Morpho: Fixed on PR 351.

Cantina Managed: Fix verified. The issue reduced the 24 hours frame to a 2 hours maximum time as per use case design.

3.2.9 `Blue_Supply` Allows Arbitrary Target Address, Leading to Misleading Bundle Construction

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `Blue_Supply` operation constructed by the SDK allows setting an arbitrary `address` field in the input operation. In practice, this field is not enforced, and during bundling, it results in a call to `generalAdapter1.morphoSupply(...)` — which uses a hardcoded internal reference to the real Morpho address.

This causes a divergence between the declared `address` in the operation and the actual contract being interacted with. As a result:

- The final encoded call targets Morpho correctly via the general adapter.
- The operation in the bundle shows the incorrect address set by the user.
- Developers or downstream systems relying on the `address` field may be misled.

In the example below, the user-supplied address is `0x1234...5678`, yet the adapter still targets the correct Morpho address:

```
{
  type: 'Blue_Supply',
  sender: generalAdapter1,
  address: '0x1234567890abcdef1234567890abcdef12345678', // User-defined, incorrect
  args: { ... }
}
```

The transaction succeeds due to the adapter's internal use of the correct address, but this introduces an inconsistency that could be problematic in simulation, tracing, or signature logic.

Recommendation: Consider to:

- Either remove the `address` parameter from the Blue operations entirely during input processing.
- Or validate it to match the known Morpho address, and revert or warn if mismatched.

This would enforce consistency between declared and executed behavior and reduce the risk of accidental misuse or confusion when debugging or generating signatures.

Morpho: Acknowledged.

Cantina Managed: Acknowledged.

3.2.10 Unrevoked DAI Allowance via Permit

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: When using DAI's `permit` flow, the generated signature sets an unlimited allowance (`MAX_UINT256`) for the spender (typically `generalAdapter1`). However, the SDK does not revoke this allowance after usage. This leads to a persistent approval that remains active beyond the current bundle, which may not be the desired behavior in many contexts, particularly when building secure, limited-scope permissioned flows. DAI's `permit` standard only supports toggling between `MAX_UINT256` and `0`. Since the SDK does not append a follow-up signature to revoke the approval, the spender retains permanent allowance after the call.

Recommendation: Consider to support an option in the SDK (e.g. `revokeAfterUse: true`) that appends a second `permit` signature (with `allowed = false` or `amount = 0`) after the action completes. Alternatively, allow users to configure whether they want the default behavior to be ephemeral (i.e. revoke immediately after) or persistent.

Morpho: Fixed on PR 350.

Cantina Managed: Fix verified.

3.2.11 Incorrect `BalancerV2` offsets in the Paraswap helper

Severity: Low Risk

Context: paraswap.ts#L4

Description: `helpers/paraswap.ts` currently defines the offsets for `swapExactAmountOutOnBalancerV2` as `{ exactAmount: 4 + 32 × 0, limitAmount: 4 + 32 × 1 }`. This is inverted.

For `swapExactAmountOut` the calldata layout of `BalancerV2Data` is:

Word	Field
0	`fromAmount` (max src, *limit amount*)
1	`toAmount` (exact dest, *exact amount*)
2	`quotedAmount`
3	`metadata`
4	`beneficiaryAndApproveFlag`

Consequently:

```
exactAmount  = toAmount   $\rightarrow$ 4n + 32n * 1n
limitAmount  = fromAmount $\rightarrow$ 4n + 32n * 0n
```

Currently this is not used in the simulation but as being a helper, in the future it can decode the calldata wrongfully. Furthermore, consider changing the comment on the `paraswap.ts` to the latest version of the Augustus Router which is 6.2 (see address 0x6A000F20005980200259B80c5102003040001068).

Recommendation: Consider modifying the offsets as follows:

```
swapExactAmountOutOnBalancerV2: {
  exactAmount: 4n + 32n * 1n,  // ← toAmount
  limitAmount: 4n + 32n * 0n,  // ← fromAmount
  quotedAmount: 4n + 32n * 2n,
},
```

Morpho: Acknowledged.

Cantina Managed: Acknowledged.

3.3 Informational

3.3.1 The simulation does not take into consideration special tokens when simulates an approval

Severity: Informational

Context: approve.ts#L10

Finding Description: The simulation does not account for tokens where the allowance must be set to zero before setting a new value. Theoretically, this would revert onchain, but the simulation does not take this into consideration.

Recommendation: Theoretically, it should be fine not to simulate this, but consider adding a comment noting this behavior, as it is handled in the bundler.

Morpho: Fixed in PR 351.

Cantina Managed: Fix verified.

3.3.2 Missing `MAX_UINT256` handling in several Blue-handlers inside the `simulation-sdk`

Severity: Informational

Context: withdrawCollateral.ts#L41

Description: The on-chain adapters treat `2**256 - 1` as a special flag meaning "use the whole balance / debt / collateral". Several handlers in the simulation layer ignore this flag, so when `assets` or `shares` is set to `MathLib.MAX_UINT_256` the dry-run diverges from real execution:

If the value is taken literally, balances under- or overflow and the local simulation reverts although the real call would succeed which means the dry-run diverges from on-chain behaviour and can mask fund-draining mistakes.

File	Parameter
`src/handlers/blue/withdrawCollateral.ts`	`assets`
`src/handlers/blue/withdraw.ts`	`shares`
`src/handlers/blue/repay.ts`	`assets` / `shares`

Recommendation: For every handler that forwards Blue contract calls:

1. Detect `if (assets == MathLib.MAX_UINT_256)` (and the equivalent for `shares`).
2. Replace it with the correct *current balance*, *full collateral*, or *full debt* depending on the action, exactly like the on-chain adapters do.

Morpho: Fixed in PR 351.

Cantina Managed: Fix verified.

3.3.3 Union technique can be bypassed via object spreading

Severity: Informational

Context: operations.ts#L65-L66

Description: The SDK encodes mutually-exclusive variants with a union such as:

```
type BlueSupplyArgs =
  | { assets: bigint; shares?: never }
  | { assets?: never; shares: bigint };
```

This catches literals like:

```
{ assets: 100n, shares: 200n } // compile-time error
```

but it does not catch objects produced via spread/merging:

```
const base = { assets: 100n };
const op = {
  type: "Blue_Supply",
  sender: user,
  args: { ...base, shares: 200n }   //  passes TS
};
```

The spread causes `args` to be inferred as `{ assets: bigint; shares: bigint }`, which is compatible with the *union* even though it violates the intended XOR rule. At runtime the handler reaches `exactlyOneZero(assets, shares)` *after* several other steps. Depending on the numbers supplied it may:

- Revert with a misleading error (e.g. "insufficient balance"), or.
- Proceed with an unexpected branch (only one of the values is used), causing the simulation to diverge from on-chain behaviour.

Because these objects can be generated programmatically, the bug is easy to miss in testing.

Proof of Concept:

```
test("should demonstrate assets XOR shares safeguard bypass by spread technique", () => {
    // Create a base object with assets for the args property
    const baseArgs = {
      id: marketA1.id,
      onBehalf: userA,
      assets: 100n,
    };

    // Create an operation with both assets and shares by using spread
    const invalidOperation = {
      type: "Blue_Supply",
      sender: userA,
      args: {
        ...baseArgs,
        // Adding shares to an object that already has assets
        shares: 200n,
      }
    };

    console.log(invalidOperation); // This will print you the invalid operation like this
    /*
    {
      "type": "Blue_Supply",
      "sender": "0xaAaAaAaaAaAaAaaAaAAAAAAAaaaAaAaAaaAaaAa",
      "args": {
        "id": "0x042487b563685b432d4d2341934985eca3993647799cb5468fb366fad26b4fdd",
        "onBehalf": "0xaAaAaAaaAaAaAaaAaAAAAAAAaaaAaAaAaaAaaAa",
        "assets": "100n",
        "shares": "200n"
      }
    }
    */

    // TypeScript allows this invalid operation to be created
    expect(invalidOperation.args).toHaveProperty("assets");
    expect(invalidOperation.args).toHaveProperty("shares");

    try {
      const simState = dataFixture;
      handleOperation(invalidOperation as Operation, simState);
    } catch (error: unknown) {
      console.log(error); // This will fail with insufficient balance (because we do not have balance in the
      ↪  users) instead of invalid input
    }
```

```
  });
});
```

Recommendation: * Stronger compile-time XOR: replace the current union with a utility type that cannot be widened, e.g.:

```
type XOR<A, B> =
  | (A & { [K in keyof B]?: never })
  | (B & { [K in keyof A]?: never });

type BlueSupplyArgs = XOR<
  { assets: bigint },
  { shares: bigint }
>;
```

or make sure you always check the objects that use the union technique for assets/shares are always checked to not contain both fields.

Morpho: Fixed in PR 351.

Cantina Managed: Fix verified.

3.3.4 Supply Input Allows Both `assets` and `shares` to Be Set Simultaneously

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the Blue supply handler, the current validation only checks if both `assets` and `shares` are zero:

```
if (assets === 0n && shares === 0n) throw new BlueErrors.InconsistentInput();
```

This allows a case where both `assets` and `shares` are non-zero, which may lead to ambiguity or unintended behavior during supply execution.

Proof Of Concept: The following logic bypasses the check and may incorrectly interpret dual input values:

```
const assets = parseEther("100");
const shares = 100n;

await handler({
  assets,
  shares,
  // ...
});
```

This passes validation despite both values being defined, which may not be an intended use case.

Recommendation: Consider to replace the condition with:

```
if ((assets === 0n && shares === 0n) || (assets !== 0n && shares !== 0n)) {
  throw new BlueErrors.InconsistentInput();
}
```

This ensures that exactly one of `assets` or `shares` is provided.

Morpho: Fixed on PR 351.

Cantina Managed: Fix verified.

3.3.5 Lack of Documentation Across SDK Packages

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The SDK, including packages such as `bundler-sdk-viem`, `simulation-sdk`, and associated type definitions, lacks comprehensive documentation across both code-level interfaces and high-level usage flows. Core components such as `Action`, `ActionBundle`, `populateBundle`, simulation handlers (e.g.,

`blue/supply.ts`), and the various `OperationType`-based abstractions are exposed without sufficient inline documentation or developer-facing references.

This absence increases onboarding time and raises the likelihood of incorrect usage. Developers must often reverse-engineer behavior by inspecting source code, especially when dealing with intricate constructs such as nested callback flows, signature-based permissioning, or assumptions in simulation input/output consistency.

Recommendation: Consider to:

- Add inline documentation (JSDoc or TypeScript comments) for exported classes, interfaces, and utility functions across all public-facing packages.
- Provide detailed reference or example usage for complex features such as:
  - Signature flows (Permit, Permit2, Dai-specific permits).
  - `skipRevert` semantics.
  - Nested callbacks and reallocation flows.
  - How simulation output maps to bundling input.
  - Document known assumptions or constraints (e.g., expected simulation freshness, address validation, or bundler chainId alignment).
  - Include a package-level README or developer guide outlining intended workflows and module responsibilities.

Improving documentation across packages would reduce developer friction and help ensure correct integration of the SDK components across consumer apps and audits.

Morpho: Acknowledged.

Cantina Managed: Acknowledged.