OpenZeppelin | security

Blue UI Audit

Compound

June 19, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-05-15 To 2025-06-06
Total Issues	9 (9 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	2 (2 resolved)
Low Severity Issues	3 (3 resolved)
Notes & Additional Information	4 (4 resolved)

Scope

OpenZeppelin audited the paperclip labs/compound-blue repository at commit cac6102.

In scope were the following files:

```
src
├── actions
│   ├── data
│   │   ├── paraswap
│   │   │   ├── config.ts
│   │   │   ├── getParaswapExactBuy.ts
│   │   │   ├── getParaswapExactSell.ts
│   │   │   └── types.ts
│   │   └── rpc
│   │       ├── getAaveV3Positions.ts
│   │       ├── getIsContract.ts
│   │       └── getSimulationState.ts
│   ├── market
│   │   ├── marketLeverageBorrowAction
│   │   │   ├── computeLeverageValues.ts
│   │   │   └── index.ts
│   │   ├── marketRepayAndWithdrawCollateralAction.ts
│   │   ├── marketRepayWithCollateralAction.ts
│   │   └── marketSupplyCollateralAndBorrowAction.ts
│   ├── migration
│   │   ├── aaveV3MarketMigrationAction.ts
│   │   ├── aaveV3PortfolioMigrationToMarketAction.ts
│   │   ├── aaveV3PortfolioMigrationToVaultAction.ts
│   │   └── aaveV3VaultMigrationAction.ts
│   ├── rewards
│   │   └── merklClaimAction.ts
│   ├── subbundles
│   │   ├── aaveV3PortfolioWindDownSubbundle.ts
│   │   ├── inputTransferSubbundle.ts
│   │   ├── subbundleFromInputOps.ts
│   │   └── types.ts
│   ├── utils
│   │   ├── bundlerActions.ts
│   │   ├── math.ts
│   │   ├── positionChange.ts
│   │   ├── prepareBundle.ts
│   │   ├── subbundlesToAction.ts
│   │   └── types.ts
│   └── vault
│       ├── vaultSupplyAction.ts
│       └── vaultWithdrawAction.ts
└── components
```

```
    └── ActionFlowDialog
        └── ActionFlowProvider.tsx
```

This engagement was not a full penetration test of the broader infrastructure. It intentionally excluded deep assessments of hosting, CI/CD, DNS, wallet providers, or the Morpho/Compound Solidity codebase. The emphasis was functional correctness, user-experience safety, and code-quality concerns inside the React/Next.js application.

System Overview

Compound Blue is a Next.js frontend that enables Polygon users to interact with Morpho-powered lending vaults and markets curated for the Compound DAO. The initiative—introduced in proposal 413—originated from a collaboration between Compound, Morpho, and Polygon It establishes a suite of Morpho Vaults on Polygon where the Compound DAO acts as `Owner`, and Gauntlet serves as `Curator`, responsible for risk management and parameter optimization. This arrangement offers faster asset listings, immutable market parameters, and reduced governance overhead compared to traditional Compound markets.

The application is structured into four primary components:

1. **UI Layer** – React components render vault, market, and migration views.
2. **Data Layer** – Off-chain market intelligence is sourced from the Whisk API, with on-chain data verified through RPC providers.
3. **Action Layer** – Type-safe functions in `src/actions` construct and simulate transaction bundles before forwarding them to the user's wallet.
4. **Wallet Integration** – Handled via RainbowKit and wagmi for account connection and chain switching.

This review focused on the React + TypeScript frontend, specifically the Action layer, which handles blockchain transactions. The application features three primary user-facing actions: earn (lend), borrow, and migrate.

Earn

The Earn feature is Compound Blue's primary yield-earning interface, built around a set of Morpho vaults owned by the Compound DAO and curated by Gauntlet. Users select the asset-specific vault of their choice, deposit tokens, and immediately start accruing yield, eliminating the need for manual market selection, rebalancing, or strategy adjustments.

Each vault is a smart contract that automatically routes supplied liquidity across multiple lending markets within the Morpho ecosystem. The user's return is the sum of (i) the base APY generated by lending the asset in those underlying markets and (ii) any additional token incentives distributed by Morpho. By pooling deposits and handling allocation on-chain, the vault abstracts away operational complexity while aiming to deliver steadier, more competitive yields.

Borrow

Borrow is Compound Blue's collateralized lending hub. Users deposit an approved asset as collateral and, in the same streamlined flow, borrow another token from the corresponding Morpho market. The interface displays real-time health factors, liquidation thresholds, and interest rates, allowing borrowers to understand risk before executing a single batched transaction.

Behind the scenes, the DApp stitches together four audited market actions:

- `MarketSupplyCollateralAndBorrow`: Allows users to supply collateral and borrow against it in a single operation.
- `MarketRepayWithCollateral`: Allows users to repay loans using their existing collateral, eliminating the need to fund repayments directly from their wallet.
- `MarketRepayAndWithdrawCollateral`: Enables users to repay their debt and withdraw collateral simultaneously in one operation.
- `MarketLeverageBorrow`: Enables leveraged borrowing, allowing users to multiply their position size. This is achieved by borrowing tokens, swapping them for the collateral asset, and supplying both the initial and swapped collateral to the market while maintaining a safe loan-to-value (LTV) ratio.

Protocol Migration

The protocol migration feature allows users to transfer positions from Aave V3 on Polygon to Compound Blue. Users can migrate individual positions or a percentage of their entire AAVE portfolio in a single action. Migrated assets can be deposited into a vault for yield generation or into a market if borrowing is required.

When necessary, the migration process leverages a flash loan-based mechanism to efficiently transfer assets between protocols. It also internally handles any token swaps required to repay loans and complete the deposit into the designated vault or market.

Security Model and Trust Assumptions

Compound Blue has been designed as a thin client: all critical state changes occur on audited smart contracts, while the frontend merely prepares and forwards transactions signed by the user's wallet. It depends on the Whisk API to access real-time on-chain data, such as vault metrics and health-factor estimates.

The present assessment presumes that the Whisk API service remains available, tamper-resistant, and performant. It is also assumed that the Morpho contract suite has undergone comprehensive independent audits and is secure.

Automated Analysis

As a complement to the manual review, a full suite of automated security checks was executed over the frontend codebase to detect implementation bugs, insecure patterns, and dependency issues.

- **Source Code Analysis (SAST):** SonarQube, Semgrep, and Snyk Code scanned every file, flagging potential vulnerabilities, code smells, and duplicate blocks. All three engines completed their checks without reporting any critical or high-severity issues, and duplication and complexity metrics remained within the accepted thresholds.
- **Software Composition Analysis (SCA):** Snyk inspected the entire dependency tree, including transitive packages, and found no vulnerable components that required immediate remediation.

Since the application executes no off-chain Create/Update/Delete operations, its attack surface is narrower than that of a conventional web application. The remaining off-chain risks are chiefly supply-chain-related (e.g., malicious npm packages) or infrastructural (e.g., DNS hijacking of the hosting domain).

Medium Severity

M-01 Morpho Repayments Have Ineffective Slippage Protection

The `morphoRepay` function of the `GeneralAdapter1` contract accepts a `maxSharePriceE27` parameter, which is intended to represent the maximum amount of assets repaid per borrow share burned, thereby providing slippage protection.

However, throughout the codebase ([1], [2], [3]), `maxSharePriceE27` is incorrectly calculated using share units (via `market.toSupplyShares`) instead of asset units (as required by `GeneralAdapter1`). Since the number of shares typically exceeds the number of assets (as confirmed by querying the total assets and shares in a market), this results in the slippage condition always passing, rendering the protection ineffective.

To ensure proper slippage protection when repaying assets, consider updating the affected instances to use `market.toSupplyAssets` instead of `market.toSupplyShares` when computing `maxSharePriceE27`.

Update*: Resolved in pull request #55 at commit `63af2a1`. The slippage calculations were updated to follow the methodology currently used in the Morpho SDK repository.*

M-02 Minimum Output for Migration Is Miscalculated and Not Displayed to Users

During portfolio migrations, users can specify a maximum slippage tolerance `S_t`, meant to define the minimum acceptable output that will be deposited into Morpho once the migration is complete. Since each migration involves two or three layers of swaps, a per-swap tolerance `S` is derived from `S_t` to preserve the overall tolerance across the full migration.

However, the current implementation contains several inconsistencies that can lead to misleading outcomes for users:

- The user-specified `S_t` is not reflected in simulation results because the `quotedOutputAssets` are added to the `GeneralAdapter1` holdings ([1], [2]) instead of the actual `minOutputAssets`. This causes the simulation to overestimate the guaranteed output after the migration.

- When the final output swap is needed, `minOutputAssets` is computed from a quote that treats `F_R` as exact input instead of using `minF_R`, so slippage in the first two layers is never factored in.

- The minimum output is calculated using the formula `minOut = quotedOut / (1 + S_t)`, which deviates from the more conventional expression `minOut = quotedOut * (1 - S_t)`. As a result, if a user specifies an `S_t` of 50% (the maximum allowed), the current calculation yields `minOut = quotedOut * 0.67`, allowing only 33% slippage instead of the intended 50%, assuming a conventional interpretation. While this benefits users and becomes more noticeable with higher slippage values, it may result in unexpected reverts in volatile markets.

To better align the system with user expectations, consider basing simulation outputs on `minOutputAssets` instead of `quotedOutputAssets`. In addition, consider updating the `minOutputAssets` calculation to reflect slippage in the first two swap layers and adopting the more conventional exact-input slippage formula. Ensure that all instances of the current formula—such as the post-quote sanity check and the `S` computation—are updated accordingly.

Update: *Resolved in pull request #62. The minimum output is now displayed to users and accurately reflects slippage in the first two swap layers. Although the exact-input slippage formula has not been updated, it is considered acceptable and consistent with practices in other dapps across the ecosystem. The team stated:*

> *We will not be addressing the slippage formula for `exactInput` swaps. We have followed the convention used by Uniswap in their SDK and their frontend, which is `minOut = quotedOut / (1 + S_t)`. This approach is more conservative, leading to tighter tolerances and more favorable worst-case swap prices for users. Note that for small `S_t` both approaches converge. While we agree with OpenZeppelin that `minOut = quotedOut * (1 - S_t)` is more intuitive, we now show users their minimum output amounts in the simulation review which should help avoid any confusion. Changing this formula would require re-derivation of the slippage math propagating through the 3 swap layers, along with associated testing.*

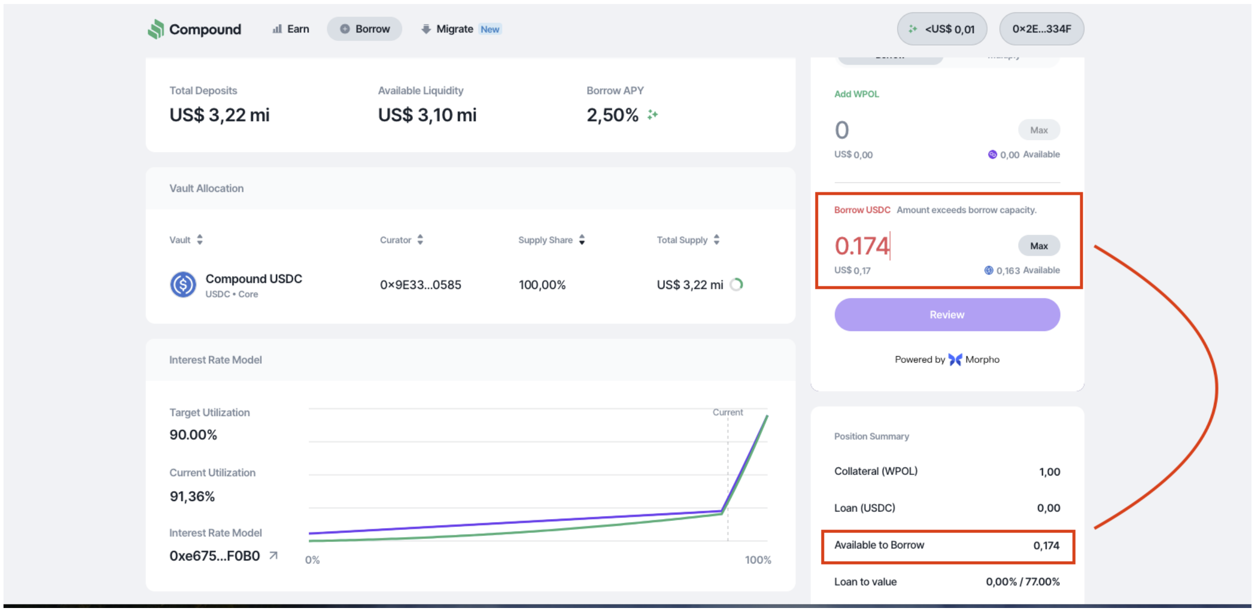
Low Severity

L-01 Inconsistent “Available to Borrow” Amount Between Borrow Input and Position Summary

Within the Borrow flow, the application displays two different values for the maximum amount that a user can borrow:

- The **Borrow input field** uses a helper that subtracts a 5% safety margin from the market’s LLTV before returning the amount.
- The **Position Summary** shows an “Available to Borrow” value calculated without applying this margin.

Since each component relies on a different formula, the value presented in the input (what the user can actually request) differs from the one shown in the Position Summary (what the UI suggests is still borrowable). This inconsistency may confuse users and undermine trust in the accuracy of the interface.



Total Deposits	Available Liquidity	Borrow APY
US$ 3,22 mi	US$ 3,10 mi	2,50%

Vault Allocation

Vault	Curator	Supply Share	Total Supply
Compound USDC USDC • Core	0×9E33...0585	100,00%	US$ 3,22 mi

Interest Rate Model

Target Utilization 90.00%

Current Utilization 91,36%

Interest Rate Model 0xe675...F0B0

Add WPOL 0 US$ 0,00 0,00 Available

Borrow USDC Amount exceeds borrow capacity.

0.174 US$ 0,17 0,163 Available

Position Summary

Collateral (WPOL)	1,00
Loan (USDC)	0,00
Available to Borrow	0,174
Loan to value	0,00% / 77.00%

Consider standardizing both calculations to apply the same margin logic and presenting a single, consistent "Available to Borrow" value throughout the Borrow flow.

Update*: Resolved in pull request #63 at commit `52317fc`. The team stated:*

> *Included the 5% borrow origination margin in the position summary's "available to borrow" field to match the borrow flow form, using the same utility function.*

L-02 No Warning and Inconsistent Handling When Supplying All Network Tokens

The current UX around supplying WPOL/POL (the network fee token) exposes users to several interconnected issues:

- **Lack of upfront warning**: Users can input a supply amount that is 100% of their WPOL/POL balance without being alerted that doing so will leave them without funds to cover future gas fees.
- **Silent simulation failure on manual input**: If a user manually inputs an amount that leaves less than the internal gas buffer (`MIN_REMAINING_NATIVE_ASSET_BALANCE_AFTER_WRAPPING`), the bundle rejects the transaction with a generic "Insufficient wallet balance" error. The UI provides no indication that the real issue is gas reservation, leaving users confused.
- **Misleading success path with the “Max” button**: Pressing “Max” triggers backend logic that subtracts the gas buffer and simulates the transaction accordingly. However, the UI and review screen still display the full wallet balance (e.g., `12.9 POL`), even though only the buffered amount (e.g., `≈ 12.76 POL`) will be deposited. This leads to a mismatch between what users see and what is actually supplied on-chain.

To reduce confusion and improve the user experience, consider implementing the following changes:

- **Gas-reserve prompt**: If a supply action would leave the wallet balance below the required gas threshold, display a clear warning such as: "Supplying this amount will leave you without enough POL to pay network fees for future transactions".
- **Max button logic**: Pre-deduct the gas buffer before populating the input field. The displayed value should match the actual deposit amount, ensuring consistency between the input, the review screen, and the executed transaction.
- **Manual input guard:** Block or automatically adjust manual inputs that would leave the wallet below the gas buffer, instead of throwing a cryptic “insufficient balance” error.
- **Accurate review screen:** To prevent misleading users, when using "Max", reflect the precise `ga1AssetBalance` that will be deposited instead of showing the raw wallet balance.

Implementing these changes will ensure that simulations, UI behavior, and on-chain results are consistently aligned.

Update*: Resolved in pull request #68 at commit `f7ab639`. The team stated:*

> *Added a gas reserve warning before simulation and on the review screen to let the user know when they may not be left with enough gas for future transactions. Fixed the form to respect the internal gas buffer, this includes both the zod input validation, and the max button behavior.*

L-03 Precision Loss When Parsing Supply Amount

The `VaultSupply` component (part of the "Earn" flow) parses the user-entered supply amount, validates it with Zod, and forwards the value to a simulation + action bundle that ultimately builds the on-chain transaction. Internally, the code coerces the raw text amount from **string → JavaScript `number` → string** before calling `parseUnits`.

Since a JavaScript `number` is a 64-bit IEEE-754 floating-point value, any token amount requiring more than ≈ 15-17 significant digits—or exceeding $2^{53} - 1$—loses precision. As a result, users who supply a fractional amount pay a few extra base units (wei) per transaction. For example, a deposit of `1.000000000000669900` is rounded internally and finally encoded as `1000000000000067`, charging **100 additional token units**. While negligible per deposit, this hidden rounding is poor practice and could become material in aggregate.

Consider keeping all monetary values as `BigInt` or arbitrary-precision decimal objects from the moment they enter the app. Replace the `z.coerce.number()` step with a `z.string()` validated by regex, and perform arithmetic via libraries such as `bignumber.js` or native `BigInt` alongside `parseUnits`. This guarantees that the exact amount a user sees is the amount they sign on-chain, eliminating silent rounding errors.

Update*: Resolved in pull request #69. The team stated:*

> *Addressed by: (i) Keeping all numeric form inputs in string form to prevent precision loss, (ii) using bigint raw values for validation instead of parsed values, (iii) only converting to `Number` when displaying formatted data (for `NumberFlow` animations), (iv) limiting asset value input fields to the precision of the underlying asset decimals (both regex, and zod validation).*

Notes & Additional Information

N-01 Misleading Documentation

Within `computeLeverageValues.ts`, multiple instances of misleading documentation were identified:

- In line 15, LLTV should be defined as the **liquidation** loan-to-value ratio of the market.
- In line 16, the units for `P_market` are inconsistent. Based on the surrounding logic, the correct expression should be `1 collateralAsset = loanAssets / P_market`.
- Between lines 35 and 36, the transition could benefit from additional explanation to clarify how one line leads to the next. Additionally, both lines should use the same comparison operator (`<` or `<=`) for consistency.

Consider updating the documentation accordingly to improve the clarity and maintainability of the codebase.

Update: *Resolved in pull request #64 at commit `6e03ef7`.*

N-02 Migration UI Allows Higher Slippage Than the Underlying Actions

The `aaveV3PortfolioWindDownSubbundle` function restricts the user-specified maximum slippage tolerance to 50%. However, the UI currently allows users to input up to 100%, which results in a simulation error.

Consider updating the UI to enforce the same slippage limit as the action to prevent user confusion and simulation failures.

Update: *Resolved in pull request #65 at commit `0eb5034`. The team stated:*

> *Addressed by clamping zod input validation for slippage to 50% (same as action limit), and adding better user facing error messages.*

N-03 Typographical Errors

Throughout the codebase, multiple instances of typographical errors were identified:

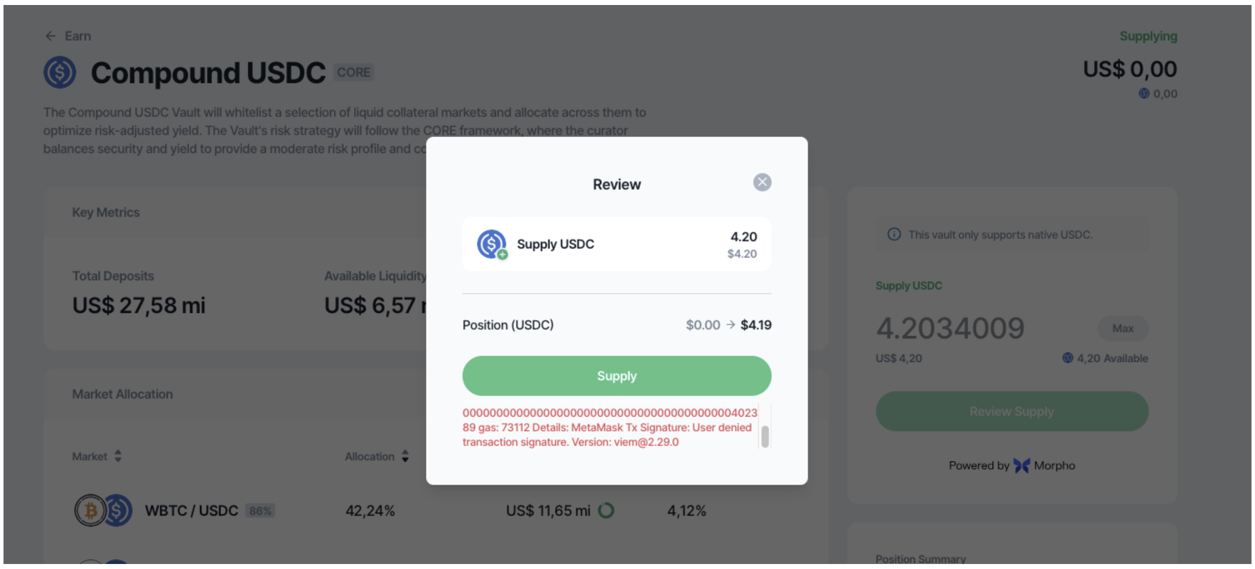
- In line 82 of `ActionFlowProvider.tsx`, "automaitcally" should be "automatically".
- In line 83 of `marketRepayWithCollateralAction.ts`, "accured" should be "accrued".
- In line 5 of `math.ts`, `TOKEN_REBASEING_MARGIN` should be `TOKEN_REBASING_MARGIN`.
- In line 60 of `inputTransferSubbundle.ts`, "Mofify" should be "Modify".
- In line 28 of `computeLeverageValues.ts`, "the" is repeated twice at the end.
- In line 124 of `positionChange.ts`, `sharedAfter` should be `sharesAfter`.
- In line 20 of `ProtocolMigratorValueHighlight.tsx`, "Migrabable" should be "Migratable".
- In line 136 of `getSimulationState.ts`, "markets and markets" should be "vaults and markets".

Consider fixing these typographical errors to improve the readability of the codebase.

Update*: Resolved in pull request #66 at commit `15f71b9`.*

N-04 Unhandled Error When User Cancels Transaction

When a user initiates a transaction in the app (e.g., supply or borrow) and then cancels the signature request in their wallet, the application displays a raw error message directly to the user:



```
00000000000000000000000000000000000000004023
89 gas: 73112 Details: MetaMask Tx Signature: User denied
transaction signature. Version: viem@2.29.0
```

This behavior is not ideal from a user-experience perspective. It exposes low-level internal details (like gas estimate and library version) that are irrelevant to the user, and may be confusing or alarming. Such an event (user cancellation) is expected and should be handled explicitly with a clean and informative message, such as "Transaction cancelled".

Consider catching the specific error that is returned and displaying a user-friendly message.

Update: *Resolved in pull request #67 at commit `9c638a7`. The team stated:*

> *Addressed by adding the error message "Transaction Cancelled" when users reject a wallet action. Note that this works for wallets which conform to EIP-1193 or include "user reject" in their error responses. This should work for the majority of wallets, with Family wallet via Wallet Connect being the only known exception.*

Conclusion

The Compound Blue UI offers a clean and intuitive interface for interacting with Morpho vaults and markets, enabling users to earn yield, borrow assets, and migrate AAVE positions. The codebase is well-structured and modular, with on-chain action logic neatly decoupled from UI components, facilitating both review and testing.

The audit identified two medium-severity issues related to slippage calculations during loan repayments and portfolio migrations. While these issues could lead to unexpected user outcomes, they do not compromise the robustness of the overall design. Additional recommendations were provided to enhance UI clarity and code consistency.

The development team is appreciated for their responsiveness and collaboration throughout the audit process. They engaged with the audit team's feedback constructively and showed a strong commitment to delivering a secure and reliable application.