Ondo PR 520

Security Review

Cantina Managed review by:
Giovanni Di Siena, Lead Security Researcher
Chinmay Farkya, Security Researcher

February 6, 2026

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 level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

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 are 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

Ondo's mission is to make institutional-grade financial products and services available to everyone.

From Jan 31st to Feb 2nd the Cantina team conducted a review of rwa-internal on commit hash e89c9020. The team identified a total of **6** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	2	1	1
Gas Optimizations	1	1	0
Informational	3	1	2
Total	**6**	**3**	**3**

2.1 Scope

The security review had the following components in scope for rwa-internal on commit hash e89c9020:

```
contracts
├── globalMarkets
│   └── tokenManager
│       └── IGMTokenManager.sol
├── limit-order
│   ├── GMTokenLimitOrder.sol
│   ├── IGMTokenLimitOrder.sol
│   ├── LimitOrderLib.sol
│   └── LimitOrderStorage.sol
└── xManager
    └── interfaces
        └── IOndoIDRegistry.sol
```

3 Findings

3.1 Low Risk

3.1.1 Batch execution of limit orders can be DOS'ed

Severity: Low Risk

Context: GMTokenLimitOrder.sol#L323-L327

Description: The `executeOrderBatch()` function is used to fill multiple orders at once, and it follows a complete-or-fail approach. If any of the orders' execution reverts, the whole batch reverts.

There are multiple ways in which users can purposefully DoS a batch execution by manipulating one of their orders included in that batch.

- User can frontrun `executeOrderBatch()` to cancel their own order, which marks it as `CANCELLED` and reverts.
- A token approval from user to the `LimitOrder` contract is required in order to proceed with order execution, but the tokens are not escrowed and remain in the user's wallet until execution. Thus, a user can frontrun `executeOrderBatch()` to revoke their token approval at the last moment, thus reverting the whole batch transaction.

The problem here is that the batch will also include orders from other users, whose execution can be repeatedly made to fail.

Recommendation: Consider implementing `executeOrderBatch()` in such a way that if any of the orders' execution fails due to any reason, the logic can continue to execute other orders successfully.

Ondo: Acknowledged: We intentionally chose atomic batch execution to avoid the indeterminate control flow introduced by a try/catch pattern. Monitoring and operational mitigations are sufficient from our view - failed batches can be resubmitted excluding the problematic order, and persistent bad actors can be removed via admin cancellation or compliance revocation. Additionally, executors can always revert to single-order execution as a fallback.

Cantina Managed: Acknowledged.

3.1.2 Precision loss due to sequential divisions in `calculateQuoteAmount()` can undercharge `BUY` orders

Severity: Low Risk

Context: LimitOrderLib.sol#L347-L354

Description: `LimitOrderLib::calculateQuoteAmount` suffers from precision loss due to intermediate truncation caused by performing two sequential divisions instead of a single combined division. `quantity * price` is first divided by 18, then further scaled up by the token decimals before the conditional rounding logic is applied; however, performing the computation in two division steps introduces intermediate truncation that can silently discard fractional value before the rounding logic executes. This edge case manifests for USDC when `(quantity * price) % 1e12` is zero and can result in `BUY` orders being undercharged by one wei.

Proof of Concept: The following test should be added to `GMTokenLimitOrder.t.sol`:

```
function test_calculateQuoteAmount_precisionLoss_USDC() public view {
  // Example values:
  //   quantity = 1e17 (0.1 GM token)
  //   price    = 1e18 + 1 ($1.000000000000000001)
  //   quantity * price = 1e35 + 1e17
  //   q = 1e17, r = 1e17
  //
  // Current Implementation:
  //   Step 1: usdValue = (1e35 + 1e17) / 1e18
  //           usdValue = floor(100000000000000000.1)
  //           usdValue = 1e17 [LOST: remainder 1e17]
  //
  //   Step 2: scaled = 1e17 * 1e6 = 1e23
```

```
  //
  //   Step 3: result = ceil(1e23 / 1e18)
  //           result = ceil(100000.0) ← EXACT INTEGER
  //           result = 100000
  //
  // Correct Implementation:
  //   numerator = (1e35 + 1e17) * 1e6 = 1e41 + 1e23
  //   result = ceil((1e41 + 1e23) / 1e36)
  //          = ceil(100000.0000000000001) ← HAS FRACTIONAL PART
  //          = 100001
  //
  // USDC Result: Current = 100000, Correct = 100001
  // LOSS OF 1 UNIT ($0.000001)

  uint256 quantity = 1e17;
  uint256 price = 1e18 + 1;

  IGMTokenManager.Quote memory quote = IGMTokenManager.Quote({
    chainId: block.chainid,
    attestationId: 1,
    userId: TEST_USER_ID,
    asset: address(gmToken),
    price: price,
    quantity: quantity,
    expiration: block.timestamp + 1 hours,
    side: IGMTokenManager.QuoteSide.BUY,
    additionalData: bytes32(0)
  });

  // Current implementation result
  uint256 currentResult = harness.calculateQuoteAmount(quote, address(usdc));

  // Correct implementation result
  uint256 numerator = quantity * price * 1e6;
  uint256 correctResult = (numerator + 1e36 - 1) / 1e36;

  assertEq(currentResult, 100000, "Current implementation should return 100000");
  assertEq(correctResult, 100001, "Correct implementation should return 100001");
  assertApproxEqAbs(correctResult, currentResult, 1, "Implementations should differ by 1
  ↪  wei");
}
```

Recommendation: Consider refactoring to perform all arithmetic in a single division operation, preserving full precision until the final rounding step. The corrected implementation combines the scaling factor into the numerator and uses a single denominator to ensure that no intermediate remainder is discarded before the rounding decision is made, resulting in correct ceiling behavior for `BUY` orders and floor behavior for `SELL` orders across all quote token decimal configurations.

Ondo: Fixed in commit f760a19.

Cantina Managed: Verified. The computation has been combined into single division to avoid intermediate truncation precision loss.

3.2 Gas Optimization

3.2.1 `ReentrancyGuardTransient` can be used to save gas on all nonReentrant calls

Severity: Gas Optimization

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

Description: The `GMTokenLimitOrder.sol` contract uses `ReentrancyGuard` from OpenZeppelin. There is a more gas efficient library for blocking reentrancies: ReentrancyGuardTransient.

Recommendation: Consider using `ReentrancyGuardTransient`.

Ondo: Fixed in commit 201e466.

Cantina Managed: Verified.

3.3 Informational

3.3.1 `LimitOrderLib::calculateQuoteAmount` can round down to zero for small buy orders

Severity: Informational

Context: LimitOrderLib.sol#L347

Description: The calculation of `usdValue` within `LimitOrderLib::calculateQuoteAmount` can round down to zero for a sufficiently small `quote.quantity * quote.price` dividend. This can result in the entire function returning zero, as the intended rounding up behavior for buy side orders is not applied. In the absence of `minimumDepositUSD` validation within `GMTokenManager`, this could be abused to systematically extract value from the protocol without paying any quote token.

Recommendation: Ensure that `minimumDepositUSD` is always configured to be non-zero.

Ondo: Acknowledged: The `minimumDepositUSD` parameter in `GMTokenManager` will always be configured to a non-zero value in production. There is no operational reason to ever allow ~zero-value deposits.

Cantina Managed: Acknowledged.

3.3.2 Partial fills of buy orders can result in user losses due to the absence of refunds

Severity: Informational

Context: LimitOrderLib.sol#L322-L336

Description: `LimitOrderLib::validateExecution` performs validation to protect users from overspending the GM token allowance intended for `EXACT_QUOTE` sell orders; however, there is no equivalent validation for `EXACT_QUOTE` buy orders. The earlier price check ensures `quote.price <= limitPrice`, but there is no check that the user receives a minimum amount of GM tokens. Without it, an executor can provide a quote with a valid price (at or below the limit) but a `quote.quantity` much lower than expected. Consider the following scenario:

1. `LimitOrderLib::executeBuy` pulls `order.exactAmount` (e.g., 100 USDC) from the user.
2. Calls `gmTokenManager.mintWithAttestation(quote, sig, USDC, 100e6)`.
3. `GMTokenManager` calculates `mintUSDonValue = quote.quantity * quote.price` (e.g., $5 \times \$10 = \$50$).
4. Converts all 100 USDC $\rightarrow$ 100 USDon.
5. Uses only 50 USDon for minting, refunds 50 USDon to `GMTokenLimitOrder`.
6. Mints 5 GM tokens to `GMTokenLimitOrder`.
7. `GMTokenLimitOrder` transfers 5 GM to the user.

To summarise, the user pays 100 USDC but receives only 5 GM tokens worth ~$50 while the excess 50 USDon remains stuck in `GMTokenLimitOrder`.

It is understood that this scenario can, in practice, occur only if the market gaps down, in which case the protocol still acts as expected despite being potentially unfavorable for users. In any case, the question remains whether excess funds should be refunded to the user. Currently, it is intended to avoid returning small balances of a separate token to avoid confusion, e.g. `createSellOrderExactOut()` will return exactly the USDC specified, even if the swap itself results in slightly more being received. However, given that there is nothing strictly preventing partial fills in the above edge case, the discrepancy between the returned and refunded amounts could be significant.

Recommendation: Consider returning refunded tokens to the user.

Ondo: Acknowledged. This contract is purpose-built for a streamlined UX flow. Under nominal market conditions, any USDon refund would be negligible (well under 1 cent), and automatically returning a secondary token on every trade would only confuse users. For the unlikely edge case of a significant market gap on these tradfi-backed assets, any material excess can be retrieved via `retrieveTokens()` and returned to the affected user manually. We accept this operational trade-off to preserve clean UX for the common case.

Cantina Managed: Acknowledged.

3.3.3 Incorrect mocked quote typehash

Severity: Informational

Context: GMTokenLimitOrder.t.sol#L49

Description: While it does not affect the existing tests, the mocked quote typehash is incorrect and should be updated to match the actual implementation.

Recommendation: Use `bytes32 additionalData` in place of `bytes additionalData`.

Ondo: Fixed in commit dd6e841.

Cantina Managed: Verified.