Ondo Bridge Registrar & USDon Converter

Security Review

Solo review by:
Sujith Somraaj, Security Researcher

October 9, 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

A security review is 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 the review 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 a 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 Oct 5th to Oct 6th the security researcher conducted a review of USDonConverter.sol and BridgeRegistrar.sol from rwa-internal on commit hash 18afc35a. A total of **8** issues were identified:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	2	1	1
Gas Optimizations	0	0	0
Informational	5	5	0
Total	**8**	**7**	**1**

3 Findings

3.1 Medium Risk

3.1.1 OFT cannot burn tokens during cross-chain transfers due to missing BURNER_ROLE

Severity: Medium Risk

Context: BridgeRegistrar.sol#L117

Description: The `BridgeRegistrar.register()` function only grants `MINTER_ROLE` to the newly deployed OFT contract but fails to grant the required `BURNER_ROLE`. This causes all outbound cross-chain token transfers to fail until the tokenAdmin manually grants the `BURNER_ROLE` to the OFT contract. When a user attempts to send tokens cross-chain via the OFT's `send()` function, the `_debit()` internal function is called:

```
function _debit(
    address _from,
    uint256 _amountLD,
    uint256 _minAmountLD,
    uint32 _dstEid
) internal returns (uint256 amountSentLD, uint256 amountReceivedLD) {
    (amountSentLD, amountReceivedLD) = _debitView(_amountLD, _minAmountLD, _dstEid);
    // Burns tokens from the caller.
    innerToken.burn(_from, amountSentLD);
}
```

The `innerToken.burn()` call attempts to burn tokens from the GMToken contract, which has the following access control:

```
function burn(address from, uint256 amount) external onlyRole(BURNER_ROLE) {
    _burn(from, amount);
}
```

Since the OFT lacks `BURNER_ROLE`, the burn operation will revert with an access control error, completely preventing users from bridging tokens out of the chain.

Recommendation: Consider implementing either of the following suggestions:

1. Replace the `burn` function with `burnFrom` function in the `OndoOFT.sol` contract:

```
  function _debit(
      address _from,
      uint256 _amountLD,
      uint256 _minAmountLD,
      uint32 _dstEid
  ) internal returns (uint256 amountSentLD, uint256 amountReceivedLD) {
    (amountSentLD, amountReceivedLD) = _debitView(_amountLD, _minAmountLD, _dstEid);
    // Burns tokens from the caller.
-    innerToken.burn(_from, amountSentLD);
+    innerToken.burnFrom(_from, amountSentLD);
  }
```

2. Grant both `MINTER_ROLE` and `BURNER_ROLE` to the OFT contract in the `register()` function:

```
  function register(address token) external override onlyRole(TOKEN_FACTORY_ROLE) whenNotPaused {
      // ....
+     IAccessControlEnumerable(token).grantRole(keccak256("BURNER_ROLE"), oft);
      // ....
  }
```

Ondo Finance: Fixed in PR 13 to use `burnFrom`.

Sujith Somraaj: Fix verified.

3.2 Low Risk

3.2.1 Precision loss in USDon to USDC redemption

Severity: Low Risk

Context: USDonConverter.sol#L143

Description: The `redeem()` function in the `USDonConverter.sol` contract suffers from precision loss when converting USDon (18 decimals) to USDC (6 decimals) due to Solidity's integer division truncation.

```
function redeem(
    uint256 rwaAmount,
    address receivingToken,
    uint256 minimumTokenReceived
) external override returns (uint256 receiveTokenAmount) {
    // ....
    usdon.safeTransferFrom(gmTokenManager, wallet, rwaAmount);
    receiveTokenAmount = rwaAmount / USDC_TO_USDON_CONVERSION_RATE; // --> precision loss
    // ...
}
```

The conversion divides rwaAmount (18 decimals) by `USDC_TO_USDON_CONVERSION_RATE` (`1e12`). Any remainder from this division is truncated due to Solidity's integer division behavior. This means any USDon amount with non-zero digits in the last 12 decimal places will lose that fractional value.

Recommendation: Consider one of the following approaches:

1. Require exact divisibility:

```
  function redeem(
      uint256 rwaAmount,
      address receivingToken,
      uint256 minimumTokenReceived
  ) external override returns (uint256 receiveTokenAmount) {
+     if (rwaAmount % USDC_TO_USDON_CONVERSION_RATE != 0) revert InvalidRedeemAmount();
    // ....
  }
```

2. Return dust to user:

```
  function redeem(
      uint256 rwaAmount,
      address receivingToken,
      uint256 minimumTokenReceived
  ) external override returns (uint256 receiveTokenAmount) {
    // ....
+     uint256 dust = rwaAmount % USDC_TO_USDON_CONVERSION_RATE;
+     rwaAmount = rwaAmount - dust;
    /// here the dust remains in the GMTokenManager contract; refunds should be handled there.
    // ....
  }
```

Ondo Finance: Acknowledged. We are aware of the truncation that will occur when decimals of the stablecoin < decimals of USDon; however, the truncation favors the protocol, so there is no additional risk here. Regardless, though, the dust that the user could be refunded would not be enough to offset the additional gas cost incurred from the additional refund logic.

Sujith Somraaj: Acknowledged.

3.2.2 Hardcoded conversion rate breaks converter functionality on chains with 18-decimal USDC

Severity: Low Risk

Context: USDonConverter.sol#L35

Description: The `USDonConverter.sol` contract uses a hardcoded `USDC_TO_USDON_CONVERSION_RATE` of `1e12`, which assumes USDC always has 6 decimals. However, on certain chains, such as BNB Chain (BSC), USDC has 18 decimals instead of 6.

This discrepancy will cause severe calculation errors and affect the smart contract functionality on those chains (or) force redeployment.

Recommendation: Consider querying decimals dynamically during deployment to set the `USDC_TO_USDON_CONVERSION_RATE` variable, instead of hardcoding it.

Ondo Finance: Fixed in PR 487.

Sujith Somraaj: Fix verified.

3.3 Informational

3.3.1 Use `encode` instead of `encodePacked` for `tokenId` generation

Severity: Informational

Context: BridgeRegistrar.sol#L106

Description: In the `register()` function in `BridgeRegistrar.sol` contract, uses abi.encodePacked() to generate a deterministic tokenId by hashing the token's symbol and name:

```
bytes32 tokenId = keccak256(
  abi.encodePacked(
    IERC20Metadata(token).symbol(),
    IERC20Metadata(token).name()
  )
);
```

abi.encodePacked() concatenates dynamic types (strings) without including length information or padding, which can lead to hash collisions. Different combinations of symbol and name can produce identical hashes.

Token A	Token B	Result
Symbol: ”ABC”, Name: ”DEF”	Symbol: ”AB”, Name: ”CDEF”	Same `tokenId`
Symbol: ”USDC”, Name: ”oin”	Symbol: ”USDCo”, Name: ”in”	Same `tokenId`

Using the same `tokenId` again will cause reverts in the `Messenger.sol` contract because each `tokenId` can only be registered once. This primarily prevents valid tokens with similar `tokenId`s (resulting from collisions) from being registered.

Recommendation: Replace `abi.encodePacked()` with `abi.encode()` to eliminate collision risks:

```
  bytes32 tokenId = keccak256(
-    abi.encodePacked(
+    abi.encode(
      IERC20Metadata(token).symbol(),
      IERC20Metadata(token).name()
    )
  );
```

Ondo Finance: Fixed in PR 488.

Sujith Somraaj: Fix verified.

3.3.2 Missing zero address validation for guardian

Severity: Informational

Context: BridgeRegistrar.sol#L69

Description: The constructor of `BridgeRegistrar.sol` takes a guardian parameter but does not verify that it isn't the zero address (i.e.,'address(0)') Deploying with address(0) may render the contract unusable, potentially requiring redeployment.

Recommendation: Consider adding a zero address check as follows:

```
  constructor(address guardian, address _ondoBridgeOwner) {
+    if (guardian == address(0)) revert OndoGuardianCantBeZero();
     /// rest of the code
  }
```

Ondo Finance: Fixed in PR 489.

Sujith Somraaj: Fix verified.

3.3.3 Missing oracle decimal validation

Severity: Informational

Context: USDonConverter.sol#L94

Description: The `USDonConverter.sol` contract relies on a Chainlink oracle to validate USDC pricing before allowing subscription and redemption operations.

The contract hardcodes the expected oracle price format to 8 decimals (MINIMUM_USDC_PRICE = 0.98e8) but fails to validate that the provided oracle actually returns prices in this format during deployment.

Recommendation: Add a validation check in the constructor to ensure the oracle returns prices with exactly 8 decimals:

```
+   error InvalidOracleDecimals();


  constructor(
    address _gmTokenManager,
    address _wallet,
    address _usdon,
    address _usdc,
    address _usdcOracle
  ) {
     /// rest of the code
+    if (usdcOracle.decimals() != 8) {
+      revert InvalidOracleDecimals();
+    }
  }
```

Ondo Finance: Fixed in PR 490.

Sujith Somraaj: Fix verified.

3.3.4 Missing natSpec documentation for constructor parameter

Severity: Informational

Context: USDonConverter.sol#L74

Description: The constructor's NatSpec documentation is incomplete and missing documentation for the `_usdcOracle` parameter.

Recommendation: Add the missing @param documentation for the `_usdcOracle` parameter:

```
  /**
     * @notice Constructs the USDonConverter contract
     * @param _gmTokenManager The GMTokenManager address authorized to call functions
     * @param _wallet         The wallet address that holds token reserves
     * @param _usdon          The USDon token address
     * @param _usdc           The USDC token address
+    * @param _usdcOracle     The Chainlink USDC/USD price feed address
     */
    constructor(
      address _gmTokenManager,
      address _wallet,
      address _usdon,
      address _usdc,
      address _usdcOracle
    ) {
      // ...
    }
```

Ondo Finance: Fixed in PR 487.

Sujith Somraaj: Fix verified.

3.3.5 Excessive oracle staleness threshold

Severity: Informational

Context: USDonConverter.sol#L39

Description: The `USDonConverter.sol` contract sets `MAX_ORACLE_DATA_AGE` to 30 hours to validate the freshness of Chainlink oracle data:

```
uint256 public constant MAX_ORACLE_DATA_AGE = 30 hours;

function _assertTokenMinimumUSDCPrice() internal view {
    (, int price, , uint256 updatedAt, ) = usdcOracle.latestRoundData();
    if (updatedAt < block.timestamp - MAX_ORACLE_DATA_AGE)
      revert OraclePriceOutdated();
    // ...
}
```

This 30-hour window is huge and defeats the purpose of Oracle staleness checks.

Recommendation: Reduce the `MAX_ORACLE_DATA_AGE` to a more reasonable value (< 24 hours).

Ondo Finance: Fixed in PR 491. This heartbeat can actually differ depending on the chain, so we decided to just parametrize this.

Sujith Somraaj: Fix verified.