OpenZeppelin | security

USDT0 Audit

January 25, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-01-21 To 2025-01-24
Languages	Solidity

Total Issues	15 (0 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	0 (0 resolved)
Notes & Additional Information	15 (0 resolved)
Client Reported Issues	0 (0 resolved)

Scope

We audited the `Everdawn-Labs/usdt0-tether-contracts-hardhat` repository at the `01cdf1d` commit.

In scope were the following files:

```
contracts
└── Wrappers
    ├── ArbitrumExtension.sol
    └── OFTExtension.sol
```

In addition, we reviewed closely related files in the `contracts/Tether` directory of the repository including `TetherToken.sol`, `TetherTokenV2.sol`, and `WithBlockedList.sol`, as well as the `MessageHashUtils` and `SignatureChecker` libraries located in the `util` directory.

System Overview

This report presents the results of an audit for the upgrade to the existing Tether (USDT) implementation on the Arbitrum network, as well as the `TetherTokenOFTExtension` contract, which is intended for deployment on new chains.

The Arbitrum upgrade introduces support for the LayerZero-powered USDT0 token, an omnichain fungible token (OFT) that enables secure and efficient cross-chain transfers. The migration plan focuses on transitioning USDT holders to USDT0 with minimal disruption, using an upgradeable proxy pattern. A key element of this upgrade is the `ArbitrumExtensionV2` contract, which facilitates the migration of USDT on Arbitrum to the OFT standard powered by LayerZero.

The `ArbitrumExtensionV2` contract implements essential features such as token migration, ownership management, and compatibility with ERC-20, EIP-2612 (permit), and EIP-3009 (gasless transfers). These features ensure efficient and secure token operations across multiple blockchain environments.

During our review, the following main areas of the system were considered: - Integration of LayerZero's omnichain standard for cross-chain interoperability. - Compatibility with LayerZero endpoints and security mechanisms. - Accurate management of token ownership, balances, and cross-chain permissions. - Migration mechanism, upgradeability, and storage consistency.

Security Model and Trust Assumptions

To support our review of the migration process, we examined the steps outlined in the Playbook for USDT0 Migration and Integration on Arbitrum Mainnet. It is assumed that the steps of the process will be strictly adhered to.

Given that the `migrate` function is unpermissioned, it is imperative to follow the upgrade process as specified, using `upgradeToAndCall` to deploy and initialize the contract within a single transaction. Additionally, the deployed `OFT contract` will have the authority to mint

and burn tokens during cross-chain transfers. It is assumed that this contract functions as intended and is free from bugs.

Privileged Roles

The contracts in scope depend on access control mechanisms to manage and operate critical functionalities. Due to the sensitivity of these operations, the parties managing them are considered trusted entities.

For instance, the account assigned the Owner role in the Tether contracts has the authority to:
- Update the OFT contract responsible for minting and burning tokens.
- Modify the token's name and symbol.
- Block accounts to restrict token transfers.
- Unblock accounts.
- Burn tokens from its own balance.
- Burn tokens from blocked accounts.

Additionally, the Authorized Sender role, present in the `TetherTokenOFTExtension` and `ArbitrumExtensionV2` contracts, has the authority to:
- Mint any amount of tokens to any account.
- Burn any amount of tokens from any account.

This audit assumes that the accounts assigned these roles and responsibilities operate as intended. Consequently, vulnerabilities or attacks targeting these roles were outside the scope of this review.

Notes & Additional Information

N-01 Multiple Contract Declarations Per File

Within `ArbitrumExtension.sol`, there are multiple contracts and interfaces declared.

Consider separating the contracts and interfaces into their own files to make the codebase easier to understand for developers and reviewers.

N-02 Use `calldata` Instead of `memory`

When dealing with the parameters of `external` functions, it is more gas-efficient to read their arguments directly from `calldata` instead of storing them to `memory`. `calldata` is a read-only region of memory that contains the arguments of incoming `external` function calls. This makes using `calldata` as the data location for such parameters cheaper and more efficient compared to `memory`. Thus, using `calldata` in such situations will generally save gas and improve the performance of a smart contract.

In `ArbitrumExtension.sol`, the `signature` parameter of the `permit`, `transferWithAuthorization`, and `receiveWithAuthorization` functions should use `calldata` instead of `memory`. This applies to the same functions in `TetherTokenV2.sol` as well.

Consider using `calldata` as the data location for the parameters of `external` functions to optimize gas usage.

N-03 `isValidSignatureNow` Reverts on Invalid ECDSA Signatures

The `isValidSignatureNow` function in the `SignatureChecker` library supports seamless verification of both ECDSA signatures from externally owned accounts and ERC-1271 signatures from smart contract accounts. This implementation is adapted from the one in OpenZeppelin Contracts, introducing a subtle difference in how invalid signatures are handled.

The original OpenZeppelin implementation ensures that the function always returns a boolean without reverting, even for invalid ECDSA signatures. In contrast, the adapted implementation reverts on invalid ECDSA signatures during the call to `recover`. While invalid signatures are never handled in a way other than reverting, this behavior could lead to unexpected error messages being returned. For instance, the `_permit` function in `TetherTokenV2` could throw the error `"ECRecover: invalid signature length"` instead of `"EIP2612: invalid signature"`.

To maintain consistency in handling invalid signatures, consider modifying the `isValidSignatureNow` function to ensure it never reverts, aligning with the behavior of the original implementation.

N-04 Inconsistency on Transfers to Blocked Recipients

The `WithBlockedList` contract extended by `TetherToken` allows the owner of the token to block specific accounts. Blocked accounts cannot get their tokens transferred to others or execute transfers on behalf of others, but they can still receive token transfers.

However, if a user authorizes a blocked recipient to receive a token transfer using the `receiveWithAuthorization` function, the recipient would be unable to execute the function due to the `onlyNotBlocked` modifier. This creates an inconsistency, as other methods of transferring tokens to the blocked recipient are permitted.

To ensure consistency among transfer methods, consider allowing blocked recipients to receive token transfers via `receiveWithAuthorization`. Alternatively, if the intention is to prevent blocked accounts from executing any kind of transfer (even self-received transfers via `receiveWithAuthorization`), consider documenting this edge case to clarify the behavior.

N-05 Multiple Optimizable State Reads

Throughout the codebase there are multiple optimizable storage reads:

- The `_newName` storage read in `ArbitrumExtension.sol`.
- The `_newSymbol` storage read in `ArbitrumExtension.sol`.
- The `_newName` storage read in `OFTExtension.sol`.
- The `_newSymbol` storage read in `OFTExtension.sol`.

Consider reducing SLOAD operations that consume unnecessary amounts of gas by caching the values in a memory variable before using them.

N-06 File and Contract Names Mismatch

The `OFTExtension.sol` file name does not match the `TetherTokenOFTExtension` contract name.

To make the codebase easier to understand for developers and reviewers, consider renaming the file to match the contract name.

N-07 Multiple Functions With Incorrect Order of Modifiers

Function modifiers should be ordered as follows: `visibility`, `mutability`, `virtual`, `override` and `custom modifiers`.

Throughout the codebase, there are multiple functions that have an incorrect order of modifiers:

- The `_EIP712NameHash` function in `ArbitrumExtension.sol`.
- The `domainSeparator` function in `EIP3009.sol`.
- The `_EIP712NameHash` function in `OFTExtension.sol`.

To improve the project's overall legibility, consider reordering the modifier order of functions as recommended by the Solidity Style Guide.

N-08 Inconsistent Order Within Contracts

Throughout the codebase, there are multiple contracts that deviate from the Solidity Style Guide due to having inconsistent ordering of functions:

- The `TetherTokenV2Arbitrum` contract in `ArbitrumExtension.sol`.
- The `ArbitrumExtensionV2` contract in `ArbitrumExtension.sol`.
- The `TetherTokenOFTExtension` contract in `OFTExtension.sol`.
- The `TetherTokenV2` contract in `TetherTokenV2.sol`.

To improve the project's overall legibility, consider standardizing ordering throughout the codebase as recommended by the Solidity Style Guide (Order of functions).

N-09 Lack of Indexed Event Parameters

Throughout the codebase, several events do not have indexed parameters:

- The `LogUpdateNameAndSymbol` event of `ArbitrumExtension.sol`.
- The `LogUpdateNameAndSymbol` event of `OFTExtension.sol`.

To improve the ability of off-chain services to search and filter for specific events, consider indexing event parameters.

N-10 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, there are contracts that do not have a security contact:

- The `ArbitrumExtensionV2` contract.
- The `TetherTokenOFTExtension` contract.

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

N-11 Non-explicit Imports Are Used

The use of non-explicit imports in the codebase can decrease code clarity and may create naming conflicts between locally-defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity file or when inheritance chains are long.

Throughout the codebase, global imports are being used:

- The import "../Tether/TetherToken.sol" import in `ArbitrumExtension.sol`.
- The import "../Tether/EIP3009.sol" import in `ArbitrumExtension.sol`.

- The import "../Tether/util/SignatureChecker.sol" import in `ArbitrumExtension.sol`.
- The import "../Tether/TetherTokenV2.sol" import in `OFTExtension.sol`.

Following the principle that clearer code is better code, consider using the named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

N-12 Duplicated Code

The `TetherTokenV2Arbitrum` and `TetherTokenV2` contracts currently contain identical code.

Consider refactoring `TetherTokenV2Arbitrum` to use `TetherTokenV2` as a base. This approach would eliminate duplicate code while preserving the desired storage layout through inheritance. For example, `TetherTokenV2Arbitrum` could implement `IArbToken`, extend `TetherTokenV2`, and directly define the state variables of `ArbitrumExtension` without inheriting from it.

N-13 Inconsistency Between `bridgeMint` and `bridgeBurn`

In the `ArbitrumExtensionV2` contract, the `bridgeMint` function reverts with a `NotImplemented` error. However, the similarly unused function `bridgeBurn` has an empty body and does not revert.

For improved clarity, consider having both functions revert with the same error, or clearly document the rationale behind the empty body in `bridgeBurn`.

N-14 Missing Docstrings

Throughout the codebase, multiple instances of missing docstrings were identified:

- In `ArbitrumExtension.sol`
  - The `IArbToken` interface
  - The `ArbitrumExtension` abstract contract
  - The `l1Address` state variable
  - The `TetherTokenV2Arbitrum` abstract contract

    - The `IArbL2GatewayRouter` interface
    - The `outboundTransfer` function
    - The `ArbitrumExtensionV2` contract
    - The `LogSetOFTContract` event
    - The `Burn` event
    - The `LogUpdateNameAndSymbol` event
    - The `USDT0_L1_LOCKBOX` state variable
    - The `migrate` function
    - The `oftContract` function
    - The `mint` function
    - The `burn` function
    - The `setOFTContract` function
    - The `updateNameAndSymbol` function
- In `OFTExtension.sol`
    - The `TetherTokenOFTExtension` contract
    - The `LogSetOFTContract` event
    - The `Burn` event
    - The `LogUpdateNameAndSymbol` event
    - The `oftContract` state variable
    - The `mint` function
    - The `burn` function
    - The `setOFTContract` function
    - The `updateNameAndSymbol` function

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

N-15 Misleading Documentation

Throughout the codebase, multiple instances of misleading documentation were identified:

- The comments next to the `_newName` and `_newSymbol` state variables in the `TetherTokenOFTExtension` contract suggest that these variables are unused. However, they are used within the `name` and `symbol` functions of the contract, respectively.

- The documentation for `_transferWithAuthorizationValidityCheck` and `_receiveWithAuthorizationValidityCheck` incorrectly states that these functions execute a transfer. In reality, they only validate the transfer without performing its execution.

Consider revising these comments to accurately reflect the functionality of the code, thereby improving overall clarity and readability.

Conclusion

The audited codebase introduces the `ArbitrumExtensionV2` and `TetherTokenOFTExtension` contracts, which extend the original Tether token implementation to enable LayerZero's Omnichain Fungible Token (OFT) functionality. These extensions serve as an interface between Tether tokens and the OFT infrastructure, facilitating cross-chain operations.

This report highlights opportunities to enhance the code for improved clarity and readability, which would facilitate future audits, integrations and development. The Everdawn team has demonstrated great diligence in sharing their codebase and providing comprehensive details, which facilitated a thorough and efficient audit process.