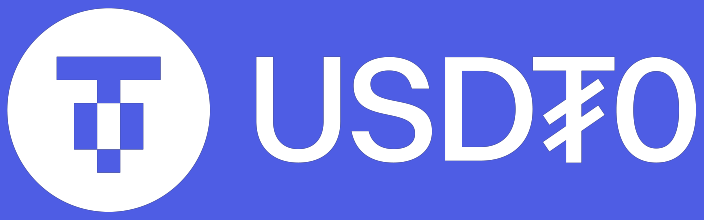
OpenZeppelin | security

USDT0 Transaction Helper Audit



April 28, 2025

Table of Contents

Summary

Type	DeFi/Stablecoin
Timeline	From 2025-04-03 To 2025-04-07
Languages	Solidity

Total Issues	15 (5 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	2 (2 resolved)
Low Severity Issues	5 (1 resolved)
Notes & Additional Information	8 (2 resolved)

Scope

We audited the Everdawn-Labs/usdt0-oft-contracts repository at commit 2ddcf81.

In scope were the following files:

```
contracts
├── helper
│   └── TransactionValueHelper.sol
└── mixins
    └── OwnableOperators.sol
```

System Overview

USDT0 is an omnichain stablecoin based on USDT. USDT0 is backed 1:1 by USDT that is issued on the Ethereum mainnet. It uses LayerZero for passing messages between chains. The USDT is locked on the Ethereum mainnet and an equivalent amount is minted on another chain. To bridge the USDT0 back to the Ethereum mainnet, it is burned on the source chain and USDT is unlocked on Ethereum.

The `TransactionValueHelper` contract allows users to pay the LayerZero bridging fees in USDT itself. For context, LayerZero allows fees to be paid in either the native token of the chain or in LayerZero's own token. The `TransactionValueHelper` contract pays the bridging fees for the user in the native token and then retrieves the equivalent from the user in USDT.

Security Model and Trust Assumptions

The following assumptions were made while reviewing these contracts:

- The `TransactionValueHelper` contract will have enough native tokens at all times to enable users to pay bridging fees.
- The privileged actors are non-malicious.

Privileged Roles

Throughout the in-scope files, the following privileged roles were identified:

- The owner can call the `setOperator`, `transferOwnership`, `setMaxGas`, `execute`, and `setPriceFactor` functions.
- An operator can call the `execute` function.

Medium Severity

M-01 `maxGas` Limit Is Easily Bypassable

In `TransactionValueHelper.sol`, the `maxGas` variable serves as the ceiling for the amount of ETH that can be sent to the OFT contract in a `send` function call. However, the `maxGas` amount is never checked against the actual amount of gas sent to the OFT contract but is instead checked against the `msg.value`. This enables users to easily bypass the set gas limit.

Consider checking the `maxGas` limit against the actual native tokens sent to the OFT contract to prevent users from being able to bypass the limit.

Update: *Resolved in pull request #87 at commit 4e2ea4e.*

M-02 `msg.value` Not Used in Price Value Accounting

In the `send` function of the `TransactionValueHelper` contract, the required ETH is sent to the OFT contract from the contract's balance. The amount of ETH that is used from the contract's balance is calculated by subtracting the balance after the call from the balance before the call. This is rather problematic as the balance before the call also includes `msg.value` which is not accounted for.

Hence, the amount of ETH calculated comes out wrong. This leads to the following situations:

- `msg.value` is the same as used ETH. In this case, even though the user paid for the complete requirement themselves, additional fees in `token` is charged to them.
- `msg.value` is less than the used ETH. In this case, the entirety of the used ETH will be charged to them even though they supplied part of it.
- `msg.value` is more than used ETH. In this case, the used ETH amount will be charged to them in addition to the `msg.value` they already sent to the contract instead of refunding them the excess.

Consider using `msg.value` in the accounting of the fees and refunds so that users do not pay extra.

Update: *Resolved in pull request #87 at commit 42dfde3.*

Low Severity

L-01 Possible Duplicate Event Emissions

When a setter function does not check whether the supplied value is different from the existing one, it opens the possibility of event spamming. Setting the same value repeatedly will emit the associated event even though the value has not changed, potentially confusing off-chain clients.

Throughout the codebase, multiple instances of possible event spamming were identified:

- The `transferOwnership` sets the `owner` and emits an event without checking if the value has changed.
- The `_initializeOwner` sets the `owner` and emits an event without checking if the value has changed.
- The `setMaxGas` sets the `maxGas` and emits an event without checking if the value has changed.
- The `setPriceFactor` sets the `priceFactor` and emits an event without checking if the value has changed.
- The `setOperator` sets the `enable` status of an `operator` and emits an event without checking if the status has changed.

Consider adding a check statement to revert the transaction if the value remains unchanged.

Update: *Acknowledged, not resolved.*

L-02 Giving Approval to the OFT Contract Is Not Always Required

The OFT contract on the Ethereum mainnet locks the USDT token and then sends a message for its equivalent amount to be minted on the destination chain. To lock the tokens, the OFT contract needs the approval of the `msg.sender` to be able to transfer its tokens. This approval is given to the OFT contract in line 93 of the `TransactionValueHelper` contract. However, some OFT contracts can directly burn the tokens of the sender and hence no

approval needs to be given to it. If such an OFT contract is used with `TransactionValueHelper` then it would result in the given allowance to add up unused.

Consider adding a call to the `approvalRequired` function of the OFT contract to determine whether an approval needs to be given for sending tokens across chains.

Update: *Resolved in pull request #87 at commit 4f1fe96.*

L-03 Potential Loss of Tokens Due to Rounding

In the `OFT.send()` function, the caller specifies the amount of tokens they would like to send via the `_sendParam.amountLD` parameter. To account for the differences in token decimals between the source and destination chains, the OFT contract calculates the actual amount to be transferred to it (or burnt) by rounding `amountLD`. The adjusted value is denoted as `amountSentLD`.

However, in the `TransactionValueHelper` contract, when the user invokes the `send` function, the entire `amountLD` is transferred to the `TransactionValueHelper` contract. This leads to a discrepancy: although only the `amountSentLD` is ultimately transferred to the OFT contract, the full `amountLD` is deducted from the user.

Consider implementing a mechanism to refund the excess (`amountLD - amountSentLD`) back to the user.

Update: *Acknowledged, not resolved.*

L-04 Missing Zero-Address Checks

When operations with address parameters are performed, it is crucial to ensure that the address is not set to zero. Setting an address to zero is problematic because it has special burn/renounce semantics. Instead, this action should be handled by a separate function to prevent accidental loss of access during value or ownership transfers.

Within `TransactionValueHelper.sol`, multiple instances of operations missing a zero-address check were identified:

- The `_priceFeed` operation
- The `_to` operation
- `newOwner` in the `OwnableOperators::tranferOwnership` function

Consider adding a zero-address check before assigning a state variable.

Update: *Acknowledged, not resolved.*

L-05 Missing Validation of `_fee.lzTokenFee`

In `TransactionValueHelper.send()`, the user specifies the token amounts `_fee.nativeFee` and `_fee.lzTokenFee` that they are willing to pay as a fee for the OFT transfer in native and LayerZero tokens, respectively. However, the `_fee.lzTokenFee` amount is neither required to be sent to the `TransactionValueHelper` contract nor is it approved to be transferred to the OFT contract. Since the `TransferValueHelper` is not expected to hold LayerZero tokens, setting a non-zero `_fee.lzTokenFee` will always cause the transaction to revert.

Consider adding a check in `send()` which ensures that `_fee.lzTokenFee` equals zero, reverting otherwise with an informative error message. In addition, include a comment in the code to document this behavior. This will help improve code clarity and execution efficiency, saving users gas by reverting earlier in the execution flow.

Update: *Acknowledged, not resolved.*

Notes & Additional Information

N-01 Variable Visibility Not Explicitly Declared

Within `TransactionValueHelper.sol`, multiple instances of variables lacking an explicitly declared visibility were identified:

- The `PRICE_FACTOR_PRECISION` constant
- The `tokenPrecision` immutable

For improved code clarity, consider always explicitly declaring the visibility of variables, even when the default visibility matches the intended visibility.

Update: *Acknowledged, not resolved.*

N-02 Functions Updating State Without Event Emissions

Within `TransactionValueHelper.sol`, the `constructor` function sets the `priceFactor` variable without emitting any event.

Consider emitting events whenever there are state changes to improve the clarity of the codebase and make it less error-prone.

Update: *Acknowledged, not resolved.*

N-03 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, multiple instances of contracts without a security contact were identified

- The `OwnableOperators` contract
- The `TransactionValueHelper` 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.

Update: *Acknowledged, not resolved.*

N-04 Naming Suggestions

Throughout the codebase, multiple opportunities for improved naming were identified:

- In the `send` function of the `TransactionValueHelper` contract, instead of reusing the `nativeAmount` variable to calculate the used ether amount after the call, a new variable called `usedNativeAmount` can be used.
- Similarly, in the same function, instead of calling the equivalent ETH amount in USD as `priceValue`, it could be called `equivalentTokenAmount`.

Consider making the above-mentioned changes to improve the readability of the codebase.

Update: *Resolved in pull request #87 at commit f8438f6.*

N-05 Unused Internal Function

The `_checkOwner` `internal` function of the `OwnableOpeartors` contract is defined but never invoked anywhere.

Consider removing the `_checkOwner` function to increase code clarity.

Update: *Acknowledged, not resolved.*

N-06 `setPriceFactor` May Allow Unintended Discounts

In the `setPriceFactor` function, the only validation performed is that `_newPriceFactor` is non-zero. However, there is no check to ensure that `_newPriceFactor` is not less than `PRICE_FACTOR_PRECISION`. If such a value is set, then in the `send` function, a discount is applied when calculating the equivalent token amount of the native tokens paid.

Consider adding a check that `_newPriceFactor` is greater than `PRICE_FACTOR_PRECISION` to limit the ability of the `owner` to mistakenly give unintended discounts.

Update: *Resolved at commit 49f99c5.*

N-07 Stablecoin Depeg Risk in `TransferValueHelper` Fee Conversion

The `TransferValueHelper` contract lets users pay for cross-chain transfers of a token using the same token instead of the native token. To do this, it converts the required native fee into an equivalent amount of the token using price data from an oracle.

To compute this equivalent amount (this is done in `send` and `quoteSend`), the contract fetches the USD price of the native token from the oracle, but it assumes that the target token (which is expected to be USDT) has a constant value of 1 USD. A configurable `priceFactor` is then applied to the conversion (which is normally expected to have a value greater than 1), which acts as a protective margin in favor of the system.

This approach introduces a potential vulnerability if the stablecoin were to lose its peg to the USD (i.e., it depegs). In such a case, if the `priceFactor` does not sufficiently account for the depeg, users may exploit the contract by effectively purchasing native tokens below market value, paying with depreciated stablecoins. This can be done by calling the `send` function using an `_oft` address under their control.

Consider using oracle prices for the stablecoin instead of assuming that it holds its peg. Alternatively, consider monitoring the stablecoin's price off-chain and regularly updating the `priceFactor` if needed. Consider also adding a comment which states that the `TransferValueHelper` contract is strictly intended for use with stablecoins.

Update: *Acknowledged, not resolved.*

N-08 Unspent Token Approvals Could Lead to Delayed Transfers

In the `send` function, the amount of tokens specified in `_sendParams` is transferred from the caller to the `TransferValueHelper` contract. The contract then grants approval for this amount to the `OFT` contract (specified by the caller), expecting that the `OFT` contract will spend the allowance during its execution (i.e., in this same transaction).

However, since the `OFT` address is provided by the user, a malicious user could supply a contract that does not spend the allowance immediately, intentionally delaying the token transfer. While this does not allow the attacker to obtain more tokens than what they had initially sent, it nonetheless enables them to transfer these tokens at a later time, outside the

expected execution flow. This can cause confusion for anyone monitoring the `TransferValueHelper` contract's token balances.

At the end of the `send` function, consider explicitly revoking the approval given to the `OFT` contract. Doing this will help prevent any misuse that could lead to confusion.

Update: *Acknowledged, not resolved.*

Conclusion

USDT0 is an omnichain stablecoin that is backed 1:1 by USDT on the Ethereum Mainnet. The audited contracts will enable users to pay the LayerZero bridging fees using the stablecoin itself. A few issues affecting the accounting were identified during the course of the review, with appropriate fixes suggested for the same. In addition, the codebase could benefit greatly from thorough testing of the `send` functionality. The protocol team is appreciated for being responsive and promptly answering any questions that the audit team had.