PUBLIC

Code Assessment

of the HyperLiquid and Stable Smart Contracts

February 25, 2025

Produced for

USD₮0

by

CHAINSECURITY

Contents

1 Executive Summary

Dear USDT0 team,

Thank you for trusting us to help USDT0 with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of HyperLiquid and Stable according to Scope to support you in forming an opinion on their security risks.

USDT0 implements contracts for the deployment of USDT on HyperLiquid and a "Stable" chain. These contracts leverage LayerZero's Omnichain Fungible Token (OFT) infrastructure and the composing feature and introduce actions that can be executed upon receiving transfers from other chains. For HyperLiquid, this is bridging the tokens from the HyperLiquid EVM chain to the L1, for the "Stable" chain, that is unwrapping the USDT0 tokens into native tokens.

The most critical subjects covered in our audit are the correctness of the integration with Layer Zero, the correct use of the system contracts of each chain, and cross contract interactions. No significant vulnerabilities were identified during this review, therefore security regarding all the aforementioned subjects is high.

Other general subjects covered are functional correctness, access control, and upgradeability, security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3
• Code Corrected	2
• Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the HyperLiquid and Stable repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

usdt0-tether-contracts-hardhat

V	Date	Commit Hash	Note
1	04 February 2025	9eb9ff6f018697ee4876247dd05e70e3b3bafba3	Initial Version
2	24 February 2025	c20b9a9a2a01fb10e40f3d39d2da2127555a0c52	Second Version

usdt0-oft-contracts

V	Date	Commit Hash	Note
1	15 February 2025	ca772d9b1ab1bd798761a7c366a3349f6819bd0f	Initial version
2	24 February 2025	a1a63ccd7fe10f6a38c5c066b553b4e16e98abb2	Second Version

For the solidity smart contracts, the compiler versions `0.8.4` and `0.8.22` were chosen. The following contracts are in scope:

usdt0-oft-contracts

- contracts/Wrappers/HyperLiquidComposer.sol
- contracts/Wrappers/OStableWrapper.sol
- contracts/Wrappers/StableComposer.sol

usdt0-tether-contracts-hardhat

- contracts/Tether/Wrappers/HyperliquidExtension.sol

2.1.1 *Excluded from scope*

Anything not listed in the scope section is considered out of scope for this assessment. This includes any third party libraries and external contracts the respective contracts interact with.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

The system consists of auxiliary contracts for USDT0 deployments on the HyperLiquid blockchain, as well as some other, unknown blockchain. The contracts are leveraging LayerZero's Omnichain Fungible

Token (OFT) infrastructure (in particular the composing feature) to introduce actions that can be executed upon receiving transfers from other chains.

- On HyperLiquid, this consists of automatically bridging USDT0 from the HyperLiquid EVM chain (HyperLiquid L2) to the HyperLiquid L1 upon receiving a transfer from another chain.
- For the "Stable" chain, this consists of unwrapping USDT0 into native tokens of equal value upon receiving a transfer from another chain.

2.2.1 HyperLiquid

2.2.1.1 HyperLiquidExtension

The contract extends the functionality of the `TetherTokenOFTExtension` contract, which is the shared implementation for the TetherToken on chains other than mainnet.

The shared implementation is a regular ERC20 token with permits, with the following additions:

- The token is upgradeable.
- It contains a blocklist of addresses that can't interact with the token.
- It contains a trusted `oftContract` role which is the only address that can `mint()` or `burn()` tokens.
- It implements `EIP3009`.
- It allows for `EIP1271` permit signatures.

On top of this, the HyperLiquid extension adds the following features:

- A role `isTrusted` can be given to and removed from addresses by the owner of the contract.
- Trusted addresses can call `transferWithHop()`, which allows them to transfer tokens from one address to another, and again from that address to a third address. No allowance is needed for this operation.

2.2.1.2 HyperLiquidComposer

The composer contract contains a single entry point, `lzCompose()`, which can only be called by the Layer Zero `EndpointV2` as part of a composed message coming from the USDT0 oApp (`OUpgradeable`). The composer is used to bridge USDT0 from the HyperLiquid EVM chain to the HyperLiquid L1 as soon as they are bridged to the EVM chain using LayerZero.

According to the implementation of the oApp, the message received by the composer should be created such that:

- `amountLD` USDT0 was credited to the composer contract by the oApp in the regular LayerZero transfer.
- The `_receiver` can be any address.

Upon receiving a composed message, the composer will use `transferWithHop()` to:

1. Transfer the USDT0 to the `_receiver`.
2. Transfer the USDT0 from the `_receiver` to the `HL_NATIVE_TRANSFER` system address.

The system precompile can be seen as a bridge between the HyperLiquid EVM chain and the HyperLiquid `L1` chain. Upon receiving the ERC20 tokens, the system precompile will credit the same amount of native spot USDT0 to the `_receiver`. From `L1` to EVM, the process is reversed, it is hence important to note that the balance of the `HL_NATIVE_TRANSFER` address should always cover the native spot balance on the `L1`.

2.2.2 Stable

2.2.2.1 OStableWrapper

The contract is a wrapper that can wrap native tokens to USDT0 ERC20 tokens (`depositTo()`) or unwrap ERC20 tokens to native tokens (`withdrawTo()`). The native token has 18 decimals while the ERC20 token (wrapped) has 6 decimals. When converting from native to ERC20, only amounts that are multiples of 10^12 are allowed, to prevent rounding errors.

Depositing to the wrapper

Using the fallback, `deposit()`, `depositTo()`, or `depositToAndCall()`, users can "deposit" native tokens to get ERC20 tokens. Effectively, the native token is burned directly from the caller using `bank.burn()`, and the ERC20 tokens are sent to the receiver. This process does not mint new tokens, it is hence expected that sufficient token amounts are available in the wrapper.

Withdrawing from the wrapper

Using `withdraw()`, `withdrawToWithPermit()`, or `withdrawTo()`, users can "withdraw" native tokens against their ERC20 tokens. The wrapper will transfer the ERC20 tokens from the caller to itself and use `bank.mint()` to mint the native tokens to the receiver.

Sending tokens cross-chain

Given an amount of native tokens, the wrapper can also wrap and send the tokens to another chain in a single call. To do so, the function `send()` burns the given amount of native tokens from the caller, and calls `oft.send()` with the corresponding amount of ERC20 tokens and various Layer Zero parameters. The caller can also send an additional amount of native tokens to the function which act as fee payment for the LayerZero transfer.

2.2.2.2 StableComposer

The composer contract contains a single entry point, `lzCompose()`, which can only be called by the LayerZero `EndpointV2` contract as part of a composed message coming from the USDT0 oApp (`OUpgradeable`). The composer is used to unwrap USDT0 into native tokens as soon as they are bridged to the stable chain using LayerZero.

According to the implementation of the oApp, the message received by the composer should be such that:

- `amountLD` USDT0 was credited to the composer contract by the oApp.
- The `_receiver` can be any address.

Upon receiving a composed message, the composer will call `wrapper.withdrawTo()` to:

1. Have the ERC20 USDT0 transferred to the wrapper.
2. Have a corresponding amount of native tokens minted to the provided `_receiver`.

This is done using the `bank` contract which can be regarded as a precompile on the "Stable" chain that is able to mint and burn native tokens.

2.2.2.3 Changes in Version 2

In the second version of the system, the only changes made were fixes for issues found during the audit.

2.3 Trust Model

2.3.1 Upgradeable Contracts

Several contracts in the system are upgradeable. For this review, the implementation of such proxies is expected to be the contracts in scope of the assessment. The admin of the different proxies is fully trusted and can upgrade the contracts to any implementation. In the worst case, they could upgrade the contracts to a malicious implementation, which could lead to a loss of funds for the users of the system.

2.3.2 LayerZero

The configuration required for managing the LayerZero applications on multiple chains is considered out of scope for this review and should be performed by the `delegate` for the oApps in the Layer Zero endpoint, that includes:

- Correctly setting peers to whitelist on each chain.
- Correctly setting the `enforcedOptions` to ensure users pay a predetermined amount of gas for delivery on the destination transaction. It should be computed such that messages sent from a source have sufficient gas to be executed on the destination chain. Setting a gas limit too small could mean that no executor has an incentive to pay for the delivery of the message at the destination, and the message should either be dropped by the admin, or some executor should execute it at a loss to resume message handling.
- Correctly setting a DVN configuration, including optional settings such as block confirmations, security threshold, the Executor, max message size, and send/receive libraries. If no send and receive libraries are explicitly set, the oApps will fall back to the default settings set by LayerZero Labs. In case LayerZero Labs changes the default settings, the oApps will be impacted and use the new default settings which implies a trust in LayerZero Labs.

In case non-EVM chains are to be supported at some point in time, token supply should adhere to LayerZero's default limit of `(2^64 - 1)/(10^6)` and use a shared decimal value of 6. Currently, no such limits are in place.

It should be noted that the owner of the oApps can set arbitrary `peers` or `delegates` which could lead to draining or losing all funds.

2.3.3 HyperLiquid

- The owner of the `HyperLiquidExtension` is fully trusted. Iin the worst case, they could update `oftContract` to a malicious contract, and mint to or burn tokens from any address.
- The `HyperLiquidExtension` is expected to have only one address which `isTrusted`. This address is assumed to be the `HyperLiquidComposer`.
- The `oftContract` of the `HyperLiquidExtension` is assumed to be the `OUpgradeable` contract of the USDT0 oApp.
- Similarly, the oApp of the `HyperLiquidComposer` is assumed to be the `OUpgradeable` contract of the USDT0 oApp.
- After deployment of the token, the link should be made between the L1 and the EVM L2 such that the ERC20 is linked to a native spot asset on the L1.
- It is assumed that at the time the link is made, the total supply of the token on the L1 is 0, and the balance of `0x2222222222222222222222222222222222222222` on the L2 is null.

2.3.4 Stable

- The `oft` of the `OStableWrapper` is assumed to be the `OUpgradeable` contract of the USDT0 oApp.
- Similarly, the oApp of the `StableComposer` is assumed to be the `OUpgradeable` contract of the USDT0 oApp.
- The `wrapper` of the `StableComposer` is assumed to be the `OStableWrapper` contract of the USDT0 oApp.
- Both `token` of the `OStableWrapper` and the `token` of the `StableComposer` are assumed to be the same token; the USDT0 token.
- Native tokens on the "Stable" chain are assumed to be of equal value as the token in `OStableWrapper` and `StableComposer`.
- The `bank` of the `OStableWrapper` is assumed to be a system contract with the following properties:

    1. The bank can burn native tokens from any account by calling `bank.burn(address, amount)`. The `OStableWrapper` is authorized to call `bank.burn()` for an arbitrary amount and address.
    2. The bank can mint native tokens to any account given `bank.mint(address, amount)`. The `OStableWrapper` is authorized to call `bank.mint()` for an arbitrary amount and address.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	**Impact**		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Locked Funds Acknowledged

5.1 Locked Funds

Design Low Version 1 Acknowledged

CS-USDT0_HYPER_STABLE-002

In both `StableComposer` and `HyperLiquidComposer`, the function `lzCompose()` is payable, but the function does not handle the value sent with the call. Native tokens transferred to the contract will be locked.

Acknowledged:

The client acknowledges the issue.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2

- Incorrect Value in I677.Transfer Code Corrected
- Missing Sanity Check Code Corrected

Informational Findings	4

- Function Visibility Too Broad Code Corrected
- Inconsistent Use of safeERC20 Code Corrected
- Incorrect Documentation Code Corrected
- Unused Imports, Interfaces and Code Code Corrected

6.1 Incorrect Value in `I677.Transfer`

Design Low Version 1 Code Corrected

CS-USDT0_HYPER_STABLE-001

The event `I677.Transfer` emitted in `OStableWrapper.depositToAndCall()` is expected to log the amount of ERC20 tokens transferred to the recipient. In `depositToAndCall()` this is not exactly the case, as it logs `msg.value`, which is the amount sent to the receiver multiplied by `1e12`.

Code corrected:

The event now contains the actual amount that has been transferred.

6.2 Missing Sanity Check

Design Low Version 1 Code Corrected

CS-USDT0_HYPER_STABLE-007

Using the `HyperLiquidComposer`, it is possible to call `token._transfer(intermediate, HL_NATIVE_TRANSFER, amt)` from an arbitrary `intermediate` address. It could be that the `intermediate` address is the `HL_NATIVE_TRANSFER` itself. Depending on the implementation of the native transfer precompile, which is closed source at the time of writing the report, this could be misinterpreted as tokens to be bridged. If funds to be bridged are accounted for by looking for the `Transfer` event, it could be possible that funds in the bridge contract are double accounted for.

Code corrected:

`HyperLiquidExtension.transferWithHop()` now reverts if the `intermediate` is equal to the `recipient`. Thus, the mentioned problem is no longer present in the `HyperLiquidComposer`.

6.3 Function Visibility Too Broad

Informational Version 1 Code Corrected

CS-USDT0_HYPER_STABLE-004

In the `OStableWrapper` contract, the following functions are `public` but could be restricted to `external`:

- `depositToAndCall()`
- `withdraw()`
- `withdrawToWithPermit()`

Code corrected:

The mentioned functions are now defined as `external`.

6.4 Inconsistent Use of `safeERC20`

Informational Version 1 Code Corrected

CS-USDT0_HYPER_STABLE-005

In the `OStableWrapper` contract, the `SafeERC20` library is used whenever interacting with the `token`. This is not the case for the call to `token.approval()` made in the constructor of the `StableComposer`.

Additionally, using `SafeERC20` might be redundant in case the contract is only used in conjunction with the token `TetherTokenOFTExtension`.

Code corrected:

The token approval is now created with `SafeERC20.forceApprove()`.

6.5 Incorrect Documentation

Informational Version 1 Code Corrected

CS-USDT0_HYPER_STABLE-006

The following documentation is misleading or incorrect:

HyperLiquidComposer:

1. The Natspec of the constructor mentions `StableComposer`, it should be `HyperLiquidComposer`.
2. The Natspec of `lzCompose()` mentions the contract being a mock, although it is not.

3. The Natspec of `lzCompose()` mentions the amount to be in the compose message while it is only in the delivery message. 3. The Natspec of `lzCompose()` mentions a token swap, although technically only a (bridge) transfer occurs.

StableComposer:

1. The Natspec of `lzCompose()` mentions the contract being a mock, although it is not.
2. The Natspec of `lzCompose()` mentions the amount to be in the compose message while it is only in the delivery message.

OStableWrapper:

1. The Natspec of `depositToAndCall()` mentions ETH and WETH10 while it is assumed that other tokens are handled by the function.

Code corrected:

All mentioned problems have been fixed.

6.6 Unused Imports, Interfaces and Code

Informational Version 1 Code Corrected

CS-USDT0_HYPER_STABLE-008

The following code is not used in the system:

HyperLiquidComposer:

1. `IOAppCore`.
2. `OStableWrapper`.
3. The event `SetTrusted` from `IHyperliquidExtension`.
4. The function `setTrusted()` from `IHyperliquidExtension`.
5. `SafeERC20`.

StableComposer:

1. `IOAppCore`.
2. `SafeERC20`.

Code corrected:

All mentioned points apart from the event and function in `IHyperliquidExtension` have been removed. `SafeERC20` in `StableComposer` has not been removed but is now in use.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 EIP-677

Informational Version 1 Acknowledged

CS-USDT0_HYPER_STABLE-003

The function `depositToAndCall()` in the `OStableWrapper` emits the following event before performing an `onTokenTransfer()` callback:

```
...
emit I677.Transfer(msg.sender,recipient, msg.value, data);
return ITransferReceiver(recipient).onTokenTransfer(msg.sender, amountOut, data);
```

It should be noted that:

1. `EIP677` is not an existing EIP and hence does not define a standardized interface.
2. The now closed proposal for EIP677 mentions the interface to be `transferAndCall()` and not `depositToAndCall()`.

Acknowledged:

The client acknowledges the issue.