PRIVATE

Code Assessment

of the Arbitrum v2 Smart Contracts

January 27, 2025

Produced for

USD₮0

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Arbitrum v2 according to Scope to support you in forming an opinion on their security risks.

In this project, the second version of the Arbitrum Extension of the TetherToken is implemented. This version of the token migrates the bridging functionalities from the Arbitrum Bridge to LayerZero.

The most critical subjects covered in our audit are functional correctness, access control, and upgradeability. No significant vulnerabilities were identified during this review, therefore 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	0

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 Arbitrum v2 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	22 Jan 2025	01cdf1d74c1bd4d9a664de4755ac5112f2a9988c	Initial Version

For the solidity smart contracts, the compiler version `0.8.4` was chosen. The following files in the folder `contracts` are in scope:

```
Wrappers/ArbitrumExtension.sol
Wrappers/OFTExtension.sol
```

2.1.1 *Excluded from scope*

Any file not listed explicitly above and any third-party library used in the codebase were not in scope of this review and are assumed to behave correctly and according to their specification. Furthermore, proxies that are already deployed, Arbitrum bridge, and LayerZero infrastructure are not in scope of this review, and we assume they work correctly and according to their specifications. Finally, the configuration of the LayerZero Endpoint is also excluded from this review.

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.

This project is an upgrade to the USDT contract on the Arbitrum blockchain that changes the token's bridge from the Arbitrum token bridge to LayerZero's Omnichain infrastructure. Additionally, EIP-3009 and EIP-1271 functionalities are added, and the token name can be changed (expected to be named "USDT0", which will be used for all USDT tokens external to mainnet). Furthermore, trusted accounts (mapping `isTrusted`) are no longer supported.

On mainnet, the non-upgradeable USDT contract will be used in conjunction with the `OAdapterUpgradeable` contract which locks tokens when they are bridged to other chains using the LayerZero infrastructure. On Arbitrum, the `OUpgradeable` contract is used as a counterpart. It holds special privileges on the Arbitrum USDT contract (`ArbitrumExtensionV2`) to mint / burn tokens when they are received / sent over the bridge.

The `ArbitrumExtensionV2` contract is an upgrade to the `ArbitrumExtension` contract currently deployed. During the proxy upgrade, the function `migrate()` will be called. It resets the native Arbitrum bridge that is currently in use and replaces it with the new LayerZero bridge. This is done by:

1. Sending a message to the L1 Arbitrum gateway that unlocks all tokens.
2. Transferring the tokens to the new `OAdapterUpgradeable` contract.
3. Setting the `ArbitrumExtensionV2.l1Address` to zero so that any remaining (or new) transfers from L1 to L2 are automatically returned back to L1 by the Arbitrum L2 gateway.

2.3 Trust Model & Assumptions

There are assumptions that are critical for the system to work as intended.

1. `USDT.basisPointsRate` fee is assumed to be 0. The `OAdapterUpgradeable._credit()` function relies on lossless transfers of tokens.
2. `USDT.deprecated` is assumed not to be set to `true`, as it might break other assumptions.
3. `OAdapterUpgradeable` is assumed not to be blacklisted in `USDT`, as it will break bridging functionality.
4. `OUpgradeable` is assumed to be set as `l2Gateway` in `ArbitrumExtensionV2`.
5. LayerZero contracts and off-chain infrastructure are assumed to be fully trusted and acting non-maliciously.
6. `OAdapterUpgradeable` is assumed to be deployed on mainnet.
7. `OUpgradeable` is assumed to be deployed on Arbitrum.
8. The `USDT` contract on Arbitrum is assumed to be upgraded to `ArbitrumExtensionV2`.
9. `ArbitrumExtensionV2.migrate()` is assumed to be called atomically during the proxy upgrade as it does not have access control.
10. LayerZero send confirmations from Mainnet to Arbitrum are set to a sufficiently high number until the tokens withdrawn from the Arbitrum bridge have been released to the `OAdapterUpgradeable` contract.

Certain roles within the system have privileged access to critical functions and are thus considered trusted:

1. The owner of the `ArbitrumExtensionV2` contract is assumed to be trusted, as they might set a malicious `l2Gateway` to freely mint new tokens.
2. The owner of `OUpgradeable` is assumed to be trusted, as they can set a malicious `peer` address.
3. The owner of `OAdapterUpgradeable` is assumed to be trusted, as they can set a malicious `peer`.
4. The delegate of `OUpgradeable` is assumed to be trusted, as they can set a malicious `SendLib` contract.
5. The delegate of `OAdapterUpgradeable` is assumed to be trusted, as they can set a malicious `SendLib` contract.

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 Findings

In this section, we describe our findings. The findings are split into these different categories:

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	0

6 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.

6.1 Compiler Version Outdated

Informational Version 1 Acknowledged

CS-USDT0-Arbv2-001

The solidity compiler is fixed to version `0.8.4`. A more recent, non-breaking version is available. Known bugs in version `0.8.4` are: https://github.com/ethereum/solidity/blob/9b97e52d45b4d94df22b3599dc0411317b056668/docs/bugs_by_version.json#L1922.

More information about these bugs can be found here: https://docs.soliditylang.org/en/latest/bugs.html

6.2 Contract Version in Domain Separator Not Upgraded

Informational Version 1 Acknowledged

CS-USDT0-Arbv2-002

The contract `ArbitrumExtensionV2` is used as the new implementation contract of a proxy contract which is already deployed. The function `migrate()` allows the caller to specify a new `name` and `symbol` for the token. However, it does not allow upgrading the `version` specified in EIP-712 and used by `domainSeparator()`.

It is worth highlighting that if the `name` of the token, which is also used by `domainSeparator()`, changes, any existing signatures become invalid even without a corresponding change of the version.

6.3 Known Issues in OpenZeppelin Dependency

Informational Version 1 Acknowledged

CS-USDT0-Arbv2-003

The codebase uses version `4.2.0` of the OpenZeppelin contracts. This version has known issues that are listed on the following page: https://github.com/OpenZeppelin/openzeppelin-contracts/security.

6.4 Misleading State Variable Names

Informational Version 1 Acknowledged

CS-USDT0-Arbv2-004

The contract `ArbitrumExtensionV2` introduces a new state variable `isMigrating` which is used to ensure that `migrate()` is executed only once:

```
function migrate(
    ...
) public {
```

```
    require(!isMigrating, "ALREADY_MIGRATED");
    ...
}
```

However, the variable name is potentially misleading as it hints that its value is `true` only while `migrate()` is executing, which is not the case.

Similarly, the variable name `l2Gateway` now stores the address of `OUpgradeable` contract on Arbitrum and is, therefore, misleading.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Excess Amount in Arbitrum L1 Gateway

Note Version 1

The function `migrate()` in ArbitrumExtension initiates an outbound transfer on Ethereum mainnet to move locked funds from the Arbitrum L1 Gateway into the LayerZero adapter contract (`OAdapterUpgradeable`):

```
function migrate(
    string memory _name,
    string memory _symbol,
    address _oftContract
) public {
    ...
    ARBITRUM_L2_GATEWAY_ROUTER.outboundTransfer(l1Address, USDT0_L1_LOCKBOX, totalSupply(), bytes(""));
    ...
}
```

The amount specified for the transfer is the `totalSupply` of USDT on Arbitrum. However, this `totalSupply` is smaller than the USDT balance of the L1 Gateway on Ethereum mainnet. Therefore, the L1 Gateway might still have a significant USDT balance after the migration. This excess amount corresponds to the pending withdrawals according to the dev team of USDT.

7.2 Upgrade and Migrate Should Be Executed Atomically

Note Version 1

The function `ArbitrumExtensionV2.migrate()` is permissionless and of high importance as it sets the new address for `l2Gateway` which can call `mint()`. Therefore, the upgrade of the proxy to the new implementation and the migration should be executed atomically, otherwise, an adversary receives the minting rights.