PUBLIC

Code Assessment

of the Fund Distribution Smart Contracts

March 20, 2024

Produced for

OFFCHAIN LABS

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Offchain Labs team,

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

Offchain Labs implements a fund router to collect funds to the Arbitrum's DAO treasury deployed on Arbitrum One from various rollups and chains (Arbitrum Nova, or Orbit chain via Ethereum).

The most critical subjects covered in our audit are the correct use of Arbitrum's bridging mechanism, the safety of the funds and the correct implementation of the distribution intervals. No major issues were detected. Security regarding all the aforementioned subjects is high.

The general subjects covered are functional correctness, gas efficiency, specification and documentation. 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 do not 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 Fund Distribution 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	6 March 2024	5d945cf8d2d100928117e1b523257ebd7146fef1	Initial Version
2	19 March 2024	61f4f60384e2ecba8250287dfb2778ce30bd82b0	Final Version

For the solidity smart contracts, the compiler version `0.8.16` was chosen.

The contracts under the `src/FeeRouter/` directory are the following:

- `ChildToParentRewardRouter.sol`
- `DistributionInterval.sol`
- `ParentToChildRewardRouter.sol`

2.1.1 Excluded from scope

All contracts not mentioned in scope are considered out of scope. Some of the contracts are going to be executed by rollups different than Ethereum. The differences in the semantics of the EVM opcodes that could be introduced by the Sequencers are beyond the scope of this audit. The contracts in scope make use of Arbitrum's bridging mechanism. Bridging is assumed to work correctly. Moreover, tokens being transferred can specify their own gateways. Such gateways could be potentially malicious or malfunctioning. Such behaviors are considered out-of-scope for this review. Finally, tokens may not be able to be bridged in any direction due to mistakes in their implementation. Transferring tokens to the treasury could fail. We assume that only properly implemented tokens are going to be used. The examination of a token's malicious behavior focused solely on its potential to disrupt the bridge's functionality concerning other tokens or native assets.

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.

Offchain Labs implements a set of smart contracts that utilize Arbitrum's bridges to send ETH or ERC20 tokens from one chain to another. This functionality facilitates the collection of fees on other Arbitrum chains and forwards them to the DAO Treasury on ArbOne: Fees on the source Arbitrum chain get accumulated. Afterwards, they are firstly sent to L1, then from L1 to the DAO treasury on ArbOne.

As can be seen from the example above, in order to collect fees on other L2 chains and send them to ArbOne, two main functionalities are required:

1. Sending fees from the source L2 to L1: accomplished by `ChildToParentRewardRouter`
2. Forwarding them from L1 to the target L2: implemented by `ParentToChildRewardRouter` through retryable tickets

Both of which inherit from `DistributionInterval` to avoid spamming.

In this review, we usually refer to the parent chain as L1 and the child chain as L2, as the contracts under review use the same terminology. In general, these contracts should work for any parent-child chain pairing, where the child chain follows Arbitrum semantics. Hence, it also covers L2-L3 messaging, where the child (L3) uses Arbitrum One as the parent.

In what follows, we first introduce some basic concepts of Arbitrum chains and then delve into each concept separately.

2.2.1 ArbOS

Arbitrum extends EVM to provide the required functionalities for L2. One of the core functionalities of `ArbOS` is to support cross-chain messaging in both directions. Any account on any side can initiate a transaction to be executed on the other chain asynchronously. To facilitate cross-chain messaging, `ArbOS` provides:

1. **Address Aliasing**: A transaction sent from an address A on L1 to L2, using A as the `msg.sender` on L2 can be problematic as there could be already another contract deployed at the address A on L2. In this case, L2 cannot distinguish between them, it opens the window for impersonating attacks. To solve this problem, the address of an L1 sender A is represented as $A + 0x1111000000000000000000000000000000001111$. As the deployment address of a contract is calculated through a cryptographic hash of the sender it is computationally infeasible to find a collision between an aliased address and another L2 address.
2. **Delayed Inbox**: A set of L1 contracts which queues L1 messages. Then, the sequencer submits the batches of messages in the Delayed Inbox and writes them into the main Inbox. If the sequencer fails to include a message from the Delayed Inbox to the Main Inbox within a predefined interval, anyone can forcefully include the message in the Main Inbox. The force inclusion functionality guarantees liveness in the system.
3. **Outbox**: Like any other optimistic rollup, for L2 messages to be eventually executed on L1, the dispute period of the Rollup Block (RBlock) containing this message should be passed. Initiating L2 to L1 transactions starts from `ArbSys` precompile (which is part of the `ArbOS`). When an RBlock is confirmed, a user can call the Outbox contract with a Merkle proof of its message being included in the `RBlock`.

2.2.2 Retryable Tickets

When L1 contracts submit a transaction to L2, it runs essentially asynchronously on L2, so the submitter cannot theoretically make sure if the forwarded transaction is successfully run. This can cause issues. For example, in the case of depositing funds from L1 to L2, if the submitted transaction on L2 fails, the user's funds can be lost. To tackle this potential issue, Arbitrum introduced Retryable Ticket system. Being retryable here means if the execution of the submitted transaction on L2 fails, ArbOS creates a retryable ticket for it to be retried again. And if the transaction had callvalue attached to it, ArbOS escrows it. Later any user can redeem the ticket by providing enough gas to retry it. The retry will run with the original sender, callvalue, and data, with the only difference being the gas parameters and who is paying for the gas.

In this section we briefly look at the lifecycle of retryable tickets:

1. **Submission**: Inbox contract on L1 provides some functions to create retryable tickets. While some functions like `createRetryableTicket()` require the user to provide a reasonable amount of funds (to create the ticket and try its execution on L2 at least once), this contract also offers some functions that do not enforce this requirement like `unsafeCreateRetryableTicket()`. All of these functions create a retryable ticket with a unique `TicketID`. Creation of this ticket on L1 triggers `ArbRetryableTx` precompile on L2 to emit an event of `TicketCreated` with the ID set upon creation of the ticket on L1.
2. **Redemption**: As mentioned before creation of a ticket on L1 is asynchronous from its execution on L2, hence a successful L1 creation does not imply a successful redemption. Once `ArbRetryableTx` creates a new ticket, two conditions are checked to see if the creation can be followed by an immediate redemption. 1) the user's L2 balance can cover gas and 2) if the maximum fee determined by the submitter on L1 is at least `l2Basefee`. If both of the conditions are met, ticket submission is followed by an attempt to redeem it. It can either fail or succeed:

    - Success: It will execute with the sender, destination, callvalue, and calldata of the original submission. Excessive amount of fees are directed to the address set by the submitter.
    - Failure: The submission fee is collected on L2 to cover the resources required to temporarily keep the ticket in memory for one week. In this case, any user can try to redeem this ticket.

To manually redeem a ticket after failure, `ArbRetryableTx.redeem()` is called with the `TicketID` as its input. In this case, `ArbOS` internally enqueues the redemption and guarantees that this redeem transaction will be tried at least once before moving to the next non-redeem transaction in the current block. If this one week elapses without a successful redemption, the ticket expires and will be automatically discarded, meaning that any message and value (other than the escrowed callvalue by `ArbOS`) it carries could be lost without the possibility of being recovered.

2.2.3 Token Bridging

Provided with a cross-chain messaging protocol, Offchain Labs implements a "canonical" bridging mechanism. As bridging different tokens may need different logic to run on each chain, in the most general form, we see a pair of contracts on L1 and L2. These are called **Gateway** s. **Gateway Router** contracts are responsible for tracking the mapping of L1 tokens to their Gateways (which in turn map them to their L2 counterpart tokens). Apart from that, each Gateway Router has a default gateway, which will be used if, for a given token, no gateways are registered. If a token is bridged from L1 and its counterpart does not already exist on L2 a new proxy contract is deployed implementing a basic ERC20 functionality.

2.2.4 `DistributionInterval`

This contract offers a function `canDistribute()`. This function takes as input the address of a token or ETH (encoded as `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`) and checks whether since the last distribution of the token at least `minDistributionIntervalSeconds` has passed.

2.2.5 `ParentToChildRewardRouter`

This contract is responsible for receiving ETH/ERC20 tokens and forwarding them to a predefined `destination` on the child chain. For example, if the destination chain is Arbitrum One, the destination should be the Arbitrum DAO treasury. The contracts have two public functions. One is designated for forwarding ETH and the other one for ERC20 tokens.

1. `receive()` funds willing to be sent to the other chain should be firstly forwarded to this contract.
2. `routeNativeFunds()` After collecting ETH, anyone can call this function to forward the ETH to the destination address on the target chain. The caller should provide it with extra fees to make the

creation and keeping of retryable tickets on L2 possible. To create retryable tickets, this function does the following:

- ETH can be distributed
- gas limit and its price are higher than the minimum
- `msg.value` covers the required gas to submit the retryable ticket and try to execute it once
- finds `excessFeeRefundAddress` which is the aliased address on L2, if the caller is a contract. Otherwise, the address of the caller
- calls `unsafeCreateRetryableTicket()` on the inbox with the predefined `destination`, while forwarding all balances of this contract to the inbox.

Please note, that the whole balance of this contract is forwarded to the inbox. Part of it (msg.value) covers the fee to create a retryable ticket. The rest is the callvalue on L2 (the actual amount being sent to `destination` on L2)

3. `routeToken()` sends all the balance of `parentChainTokenAddr` belonging to this contract to the corresponding gateway and finally makes a call to `outboundTransferCustomRefund()` on this gateway. This function assures that `msg.value` is enough to cover the fees and creates a retryable ticket. It also checks that there exists a peer ERC20 on L2. It is worth mentioning that down the call path of `outboundTransferCustomRefund()`, `createRetryableTicket()` is called which assures that `msg.value` is enough to cover the fees to create a retryable ticket and L2 call value. Excessive fees are refunded to `msg.sender` or its L2 alias, depending on whether `msg.sender` is an EOA or a contract. While `callValueRefundAddress` (i.e., the L2 address to which the l2CallValue is credited if the ticket times out or gets canceled) is L2 alias of `ParentToChildRewardRouter`.

When sending native funds or tokens from L1 to L2, the same least gas is required `_minGasLimit`, although sending ERC20 tokens are expected to consume more, due largely to their extra transfer logic. We assume that Upon deployment, a value greater than the greater of the two values (expected to be the value for a token transfer) should be set.

2.2.6 `ChildToParentRewardRouter`

As sending funds/tokens from L2 to L1 does not involve creating and keeping retryable tickets, this contract has no checks on `msg.value` being enough to cover the creation of tickets and their execution. Hence, if there is no value to be sent or the trial to send is earlier, in contrast to `ParentToChildRewardRouter`, it just skips and does not revert. Likewise, however, it has two main functions:

1. `routeNativeFunds()`: if there is a non-zero balance of ETH and if it can be distributed (i.e., time has elapsed), it forwards all of its balance to the `ArbSys.withdrawEth()` precompile at `0x0000000000000000000000000000000000000064` which sends the ETH to `parentChainTarget` on L1.
2. `routeToken` firstly, finds the respective gateway for the token, approves its balance for the gateway, and calls `outboundTransfer()` on the gateway router. Consequently, an outbound transaction is created which calls `sendTxToL1()` on the `ArbSys` precompile.

2.2.7 *Trust Models and Assumptions*

`ParentToChildRewardRouter` is considered trustless. Any user can interact with its functions to send ETH and ERC20 to the `destination` (DAO treasury) on L2.

`ChildToParentRewardRouter` is also treated as trustless. Any user can call its functions.

We assume for each token to be bridged the correct gateway is set and that the bridge functions properly. Also, we assume that the contracts will be correctly configured.

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 any open findings. Findings that have been resolved have been moved to the Resolved Findings section. 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 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the 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	0
Informational Findings	2

- Reseting Approvals Code Corrected
- Missing Checks Code Corrected

6.1 Missing Checks

Informational Version 1 Code Corrected

CS-OCFD-002

Sanity checks are missing for the following values:

- `ParentToChildRewardRouter.constructor`: `minGasPrice`, `minGasLimit`, and `destination`
- `ChildToParentRewardRouter.constructor`: `parentChainTarget`

Moreover, in `ParentToChildRewardRouter.routeToken()`, when querying the gateway for a token, there is a case where the zero address is returned e.g., when the token is `DISABLED`. In such a case, `routeToken()` will fail late during `outboundTransferCustomRefund()` consuming more gas than it should. Note that `routeToken()` could already fail during the approve call to `address(0)`, since most token implementations prevent that. However, the ERC20 standard doesn't explicitly disallow giving approval to `address(0)`.

Code corrected:

Offchain Labs has addressed `destination` and `parentChainTarget`, but left the others, mentioning:

```
We've added sanity checks for designation and parentChainTarget.
We have not included sanity checks for minGasPrice or minGasLimit because although for
the Nova-to-Treasury-Router and Orbit-Chains-to-Treasury-Router deployments non-zero
values will be used, we believe the contracts should be unopinionated about this for other,
future use cases.
```

About handling the case of a disabled token:

```
We think the cost of adding in additional check in the normal case
(token isn't disabled) outweighs the benefit of gas savings from reverting earlier in the rare
case (token is disabled).
```

6.2 Reseting Approvals

Informational Version 1 Code Corrected

CS-OCFD-001

When transferring from the parent chain an approval must given to the gateway of the specific token. The allowance is specified to be equal to the amount to be transferred. This means that by the end of the transaction, the full allowance will be consumed and set to 0 releasing the repsective storage slot. In case, the allowance was set to a non-zero value the slot would be retained and thus the transaction would be cheaper.

Code corrected:

The allowance is set to `value + 1` therefore the slot is not released. Note that in case the allowance was always reset to the same value e.g., `1` the transaction would be even cheaper. In the current implementation the allowance always increases by one at the end of the transaction.

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 Contract Check

Note Version 1

In order to check whether an aliasing should be used for the `excessFeeRefundAddress`, `Address.isContract()` is used. The function checks whether there is a bytecode deployed in this address. However, the call returns false if the contract is not yet deployed but it made the call during its construction. This could potentially set the `excessFeeRefundAddress` to an already deployed contract on L2 which might be unexpected.

7.2 Native Assets From `ChildToParentRewardRouter`

Note Version 1

Each `ChildToParentRewardRouter` is dedicated to exactly one token. Moreover, one can create a dedicated `ChildToParentRewardRouter` for native ETH by setting the `parentChainTokenAddress` to `address(1)`. However, everyone can still use any of the token child-to-parent routers to bridge native tokens. Using a dedicated native funds gateway removes the logic of supporting native funds on each implementation.

Acknowledged:

Offchain Labs acknowledged the existence of this possibility:

```
Explicitly disabling tokens with address(1) was only
introduced to avoid the awkwardness of using an arbitrary non-token address when only
native funds are intended (i.e., to route a chain's native fees). We don't think it's necessary
to provide the option of explicitly disabling native fees.
```

7.3 No Fallback on Cancellation

Note Version 1

During the transfer of native ETH from parent chain to child, a low level call is created on L2 with sender the original msg.sender (maybe aliased) on L1 and receiver the destination contract. If the ticket is redeemed successfully then the call will trigger the `receive()` function of the destination contract on L2. In case the ticket is expired, the escrowed amount, namely the amount specified from `l2CallValue`, will be credited to the destination address without triggerring the fallback. In the current scope this is not an issue. However, if the logic of the destination address is updated, one should take into account this behavior.