Custom Fee Token

Security Assessment (Summary Report)

August 1, 2024

Prepared for:
Harry Kalodner, Steven Goldfeder, and Ed Felten
Offchain Labs

Prepared by: **Gustavo Grieco, Troy Sargent, and Kurt Willis**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits **1**

Notices and Remarks **2**

Table of Contents **3**

Project Summary **4**

Project Targets **5**

Executive Summary **6**

Summary of Findings **7**

Detailed Findings **8**

1. Double entrypoint or DeFi integrated ERC20 tokens should not be used 8

2. Token bridge will receive and lock ether 10

3. Cross-chain message out-of-order execution could affect correct token bridge deployment 11

A. Vulnerability Categories **12**

B. Development Practices **15**

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Kurt Willis, Consultant
kurt.willis@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 25, 2023	Pre-project kickoff call
October 2, 2023	Delivery of report draft
October 2, 2023	Report readout meeting
August 1, 2024	Delivery of summary report

Project Targets

The engagement involved a review and testing of the targets listed below.

nitro-contracts PR#19

Repository	https://github.com/OffchainLabs/nitro-contracts/pull/19
Version	`PR#19 (f23c15c...7dc1aa4)`
Type	Solidity
Platform	EVM

token-bridge-contracts PR#33

Repository	https://github.com/OffchainLabs/token-bridge-contracts/pull/33
Version	`PR#33 (8cb573a...6396a17)`
Type	Solidity
Platform	EVM

token-bridge-contracts PR#34

Repository	https://github.com/OffchainLabs/token-bridge-contracts/pull/34
Version	`PR#34 (32d00e7...9503d3c)`
Type	Solidity
Platform	EVM

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of the Custom Fee Token implemented in the PRs detailed in the Project Targets section. These changes allow new rollups to be deployed using a specific ERC20 implementation that will be used to pay transaction fees.

A team of three consultants conducted the review from September 21, 2023 to September 29, 2023, for a total of three engineer-weeks of effort. With full access to source code and documentation, we performed a manual review of the codebase.

Observations and Impact

Offchain Labs added a feature to allow rollup owners to select a specific ERC20 token on the parent chain that will be used to pay for the transaction fees, completely replacing the use of ETH in the child chain. This new feature requires changes in the token bridge and ArbOS, including new code to correctly deploy the token bridge in the chain.

We focused on the changes in each PR, but we have not performed a full review of the repositories involved. We also worked under the assumption that rollup owners will carefully review the available documentation before deployment, in order to avoid known issues with certain types of tokens (e.g., rebasing tokens).

This review uncovered two high-severity issues related to the assumptions about how ERC20 should behave (TOB-ARB-CFT-001) and how funds can flow into the token bridge contracts (TOB-ARB-CFT-002).

Recommendations

We recommend that Offchain Labs fix the reported issues and ensure that the token bridge documentation is up to date before deployment to ensure that rollup owners use suitable ERC20 as fee tokens.

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Double entrypoint or DeFi integrated ERC20 tokens should not be used	Access Controls	High
2	Token bridge will receive and lock ether	Undefined Behavior	High
3	Cross-chain message out-of-order execution could affect correct token bridge deployment	Undefined Behavior	Medium

Detailed Findings

1. Double entrypoint or DeFi integrated ERC20 tokens should not be used	
Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-ARB-CFT-001
Target: `src/bridge/ERC20Bridge.sol`	

Description

The use of ERC20 tokens with two or more entrypoints can allow an attack to drain the bridge.

The use of ERC20 tokens for paying fees in the rollup/bridge requires a number of checks and restrictions to avoid loss of funds. One of these checks is implemented in the bridge when a withdraw is executed:

```
    function _executeLowLevelCall(
        address to,
        uint256 value,
        bytes memory data
    ) internal override returns (bool success, bytes memory returnData) {
        // we don't allow outgoing calls to native token contract because it could
        // result in loss of native tokens which are escrowed by ERC20Bridge
        if (to == nativeToken) {
gvladika marked this conversation as resolved.
            revert CallTargetNotAllowed(nativeToken);
        }

        // first release native token
        IERC20(nativeToken).safeTransfer(to, value);
        success = true;
        …
```

Figure 1.1: Header of the `_executeLowLevelCall` function in `src/bridge/ERC20Bridge.sol`

Users are not allowed to directly call the native token address; otherwise, they could transfer funds out. However, this check will not be sufficient if the token has more than one entrypoint (e.g., when two different addresses can be used to execute ERC20 operations, such as transfer).

Another problematic type of ERC20 is tightly integrated in DeFi applications. For instance, the LUSD ERC20 token contains the following function:

```
    function burn(address _account, uint256 _amount) external override {
        _requireCallerIsBOorTroveMorSP();
        _burn(_account, _amount);
    }
```

Figure 1.2: Burn function from the LUSD token

This token can be minted or burned through a manager contract (which is different from the token contract itself), thereby bypassing the above check. In particular, this DeFi allows LUSD token owners to open, close, or repay vaults, so all of the bridge ERC20 LUSD could be easily manipulated using the low-level callback without requiring allowances to be set up.

Exploit Scenario
A user creates a rollup that uses a double entrypoint tokens for fees, allowing any user to drain the bridge contract.

Recommendations
Short term, clearly document this limitation to make sure of this potential security issue.

Long term, review the assumptions required by ERC20 tokens in order to be integrated in each component.

References

- Medium-severity bug in Balancer Labs

2. Token bridge will receive and lock ether	
Severity: **High**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-ARB-CFT-002
Target: `tokenbridge/ethereum/gateway/L1OrbitERC20Gateway.sol`	

Description

The token bridge's entrypoint for deposits can receive ether, but the token bridge cannot retrieve it in any way.

Users deposit ERC20 tokens using the `outboundTransfer*` functions from the token bridge. An example is shown below:

```
function outboundTransferCustomRefund(
     address _l1Token,
     address _refundTo,
     address _to,
     uint256 _amount,
     uint256 _maxGas,
     uint256 _gasPriceBid,
     bytes calldata _data
 ) public payable override returns (bytes memory res) {
 …
```

Figure 1.2: Header of the `outboundTransferCustomRefund` function in `src/tokenbridge/ethereum/gateway/L1OrbitERC20Gateway.sol`

This function will trigger the creation of a retryable ticket, so it needs funds to pay fees and gas. These fees can be paid using ether or some specific ERC20, but in different token bridge deployments that share the same interface. In the latter case, the entry function should not receive ether even though it is payable.

Exploit Scenario

A user accidentally provides ether to a token bridge associated with a rollup that uses a custom ERC20 token fee. The ether will be locked in the token bridge.

Recommendations

Short term, add a condition that checks the value provided into the `outboundTransfer` function, and have the function revert if the value is positive.

Long term, review how funds flow from the user to/from different components, and ensure that there are no situations where tokens can be trapped.

3. Cross-chain message out-of-order execution could affect correct token bridge deployment

Severity: **Medium**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ARB-CFT-002
Target: `tokenbridge/ethereum/L1AtomicTokenBridgeCreator.sol`	

Description

Out-of-order execution of outbox transactions on L1 and retryable tickets on L2 can lead to unexpected results when a token bridge is created. This issue relies on the specific ordering of retryable tickets.

The token bridge creation requires retryable tickets to be submitted and executed in a certain order:

```
    /**
     * @notice Deploy and initialize token bridge, both L1 and L2 sides, as part of
a single TX.
     * @dev This is a single entrypoint of L1 token bridge creator. Function deploys
L1 side of token bridge and then uses
     *      2 retryable tickets to deploy L2 side. 1st retryable deploys L2 factory.
And then 'retryable sender' contract
     *      is called to issue 2nd retryable which deploys and inits the rest of the
contracts. L2 chain is determined
     *      by `inbox` parameter.
     *
     *      Token bridge can be deployed only once for certain inbox. Any further
calls to `createTokenBridge` will revert
     *      because L1 salts are already used at that point and L1 contracts are
already deployed at canonical addresses
     *      for that inbox.
     */
    function createTokenBridge(
        address inbox,
        address rollupOwner,
        uint256 maxGasForContracts,
        uint256 gasPriceBid
    ) external payable {
    …
```

Figure 3.1: Header of the `createTokenBridge` function in `L1AtomicTokenBridgeCreator.sol`

However, a malicious user can leverage the out-of-order execution of retryable tickets to break the assumptions of the token bridge creator and produce a failed deployment.

Exploit Scenario

Alice starts the deployment of a canonical token bridge for a new rollup. Eve notices this deployment and spams the rollup bridge with transactions to increase the L2 gas cost, and the tickets are not auto-redeemed. Later, Eve can trigger the tickets out of order to produce a broken deployment. Alice will not be able to redeploy, and no canonical deployment of the token bridge can be used.

Recommendations

Short term, consider migrating part of the deployment steps to L2 and require a single retryable ticket to be executed.

Long term, review all possible ways in which the out-of-order execution of retryable tickets may affect each component and document.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Development Practices

In this section, we provide best practices regarding code complexity management.

- When designing smart contracts with the purpose of reuse, try to minimize inheritance whenever possible, as it often can lead to the creation of multiple levels of indirection and make the execution flow very hard to follow. While some amount of inheritance is expected, it can easily be abused.

  An example of a complex inheritance structure can be seen when tracing the internal function calls when creating deposits to `L1OrbitERC20Gateway`:

  - `L1GatewayRouter.outboundTransferCustomRefund`
  - `super (L1ERC20Gateway).outboundTransferCustomRefund`
  - `super (L1ArbitrumGateway).outboundTransferCustomRefund`
  - `L1ArbitrumGateway._parseUserEncodedData` (overloaded in L1OrbitERC20Gateway)
  - `L1ArbitrumGateway.calculateL2TokenAddress` (overloaded in `L1ERC20Gateway`)
  - `L1ArbitrumGateway.getOutboundCalldata` (overloaded in `L1ERC20Gateway`)
  - `L1ArbitrumGateway._initiateDeposit` (overloaded in `L1OrbitERC20Gateway`)
  - `L1ArbitrumMessenger.sendTxToL2CustomRefund`
  - `L1ArbitrumMessenger._createRetryable` (overloaded in `L1OrbitERC20Gateway`)

  As shown in figure B.1, the calls jump back and forth between four contracts using the `super` keyword and function overloading. Abstraction is useful for separating concerns and reducing code duplication; however, it should not be overused. Too much abstraction can make following execution traces difficult and introduce significant mental overhead.

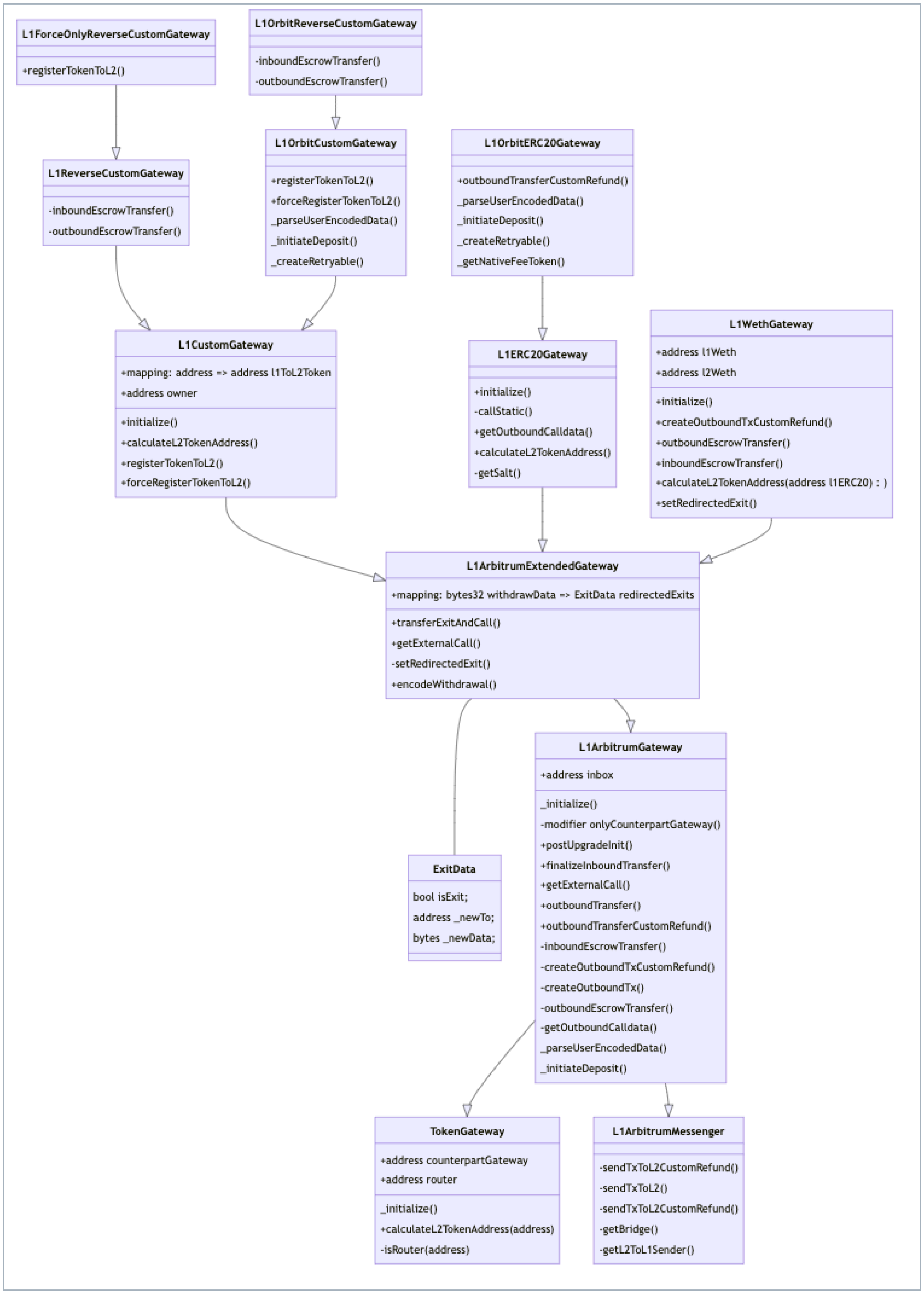

Figure B.1: Arbitrum Gateway's complex inheritance structure

- When overloading functions, aim to place optional parameters at the end if possible. This makes it clearer what parameters are optional and reduces cognitive load by not shifting positions of other parameters too much.

```
// ...
function registerTokenToL2(
    address _l2Address,
    uint256 _maxGas,
    uint256 _gasPriceBid,
    uint256 _maxSubmissionCost,
    uint256 _feeAmount
) external returns (uint256) {
    return
        registerTokenToL2(
            _l2Address,
            _maxGas,
            _gasPriceBid,
            _maxSubmissionCost,
            msg.sender,
            _feeAmount
        );
}

// ...
function registerTokenToL2(
    address _l2Address,
    uint256 _maxGas,
    uint256 _gasPriceBid,
    uint256 _maxSubmissionCost,
    address _creditBackAddress,
    uint256 _feeAmount
) public returns (uint256) {
    return
        _registerTokenToL2(
            _l2Address,
            _maxGas,
            _gasPriceBid,
            _maxSubmissionCost,
            _creditBackAddress,
            _feeAmount
        );
}
```

Figure B.2: Overloading `registerTokenToL2` in `L1OrbitCustomGateway`

- Aim to keep the same order when passing on function parameters.

```
function createOutboundTxCustomRefund(
    address _refundTo,
    address _from,
    uint256, /* _tokenAmount */
```

```
    uint256 _maxGas,
    uint256 _gasPriceBid,
    uint256 _maxSubmissionCost,
    bytes memory _outboundCalldata
) internal virtual returns (uint256) {
    // We make this function virtual since outboundTransfer logic is the same for
many gateways
    // but sometimes (ie weth) you construct the outgoing message differently.

    // msg.value is sent, but 0 is set to the L2 call value
    // the eth sent is used to pay for the tx's gas
    return
        sendTxToL2CustomRefund(
            inbox,
            counterpartGateway,
            _refundTo,
            _from,
            msg.value, // we forward the L1 call value to the inbox
            0, // l2 call value 0 by default
            L2GasParams({
                _maxSubmissionCost: _maxSubmissionCost,
                _maxGas: _maxGas,
                _gasPriceBid: _gasPriceBid
            }),
            _outboundCalldata
        );
}
```

Figure B.3: L2GasParams are not in order with function parameters. (`L1ArbitrumGateway`)

- Aim to be consistent with function variable names when possible, or declare temporary variables and add comments explaining variable name switching.

```
function _initiateDeposit(
    address _refundTo,
    address _from,
    uint256, // _amount, this info is already contained in _data
    uint256 _maxGas,
    uint256 _gasPriceBid,
    uint256 _maxSubmissionCost,
    uint256 tokenTotalFeeAmount,
    bytes memory _data
) internal override returns (uint256) {
    return
        sendTxToL2CustomRefund(
            inbox,
            counterpartGateway,
            _refundTo,
            _from,
            tokenTotalFeeAmount,
            0,
            L2GasParams({
```

```
                _maxSubmissionCost: _maxSubmissionCost,
                _maxGas: _maxGas,
                _gasPriceBid: _gasPriceBid
            }),
            _data
        );
}
```

Figure B.4: `tokenTotalFeeAmount` is mapped to `_l1CallValue`. (`L1OrbitERC20Gateway`)

```
function sendTxToL2CustomRefund(
    address _inbox,
    address _to,
    address _refundTo,
    address _user,
    uint256 _l1CallValue,
    uint256 _l2CallValue,
    L2GasParams memory _l2GasParams,
    bytes memory _data
) internal returns (uint256) {
    // ...
}
```

Figure B.5: `tokenTotalFeeAmount` is mapped to `_l1CallValue`. (`L1ArbitrumMessenger`)

- Use named parameters or temporarily declare named variables when passing in unnamed constants as function parameters.

```
function _initiateDeposit(
    address _refundTo,
    address _from,
    uint256, // _amount, this info is already contained in _data
    uint256 _maxGas,
    uint256 _gasPriceBid,
    uint256 _maxSubmissionCost,
    uint256 tokenTotalFeeAmount,
    bytes memory _data
) internal override returns (uint256) {
    return
        sendTxToL2CustomRefund(
            inbox,
            counterpartGateway,
            _refundTo,
            _from,
            tokenTotalFeeAmount,
            0,
            L2GasParams({
                _maxSubmissionCost: _maxSubmissionCost,
                _maxGas: _maxGas,
                _gasPriceBid: _gasPriceBid
            }),
```

```
            _data
        );
}
```

Figure B.6: `tokenTotalFeeAmount` is mapped to `_l1CallValue`. (`L1OrbitERC20Gateway`)

```
function _initiateDeposit(
    address _refundTo,
    address _from,
    uint256, // _amount, this info is already contained in _data
    uint256 _maxGas,
    uint256 _gasPriceBid,
    uint256 _maxSubmissionCost,
    uint256 tokenTotalFeeAmount,
    bytes memory _data
) internal override returns (uint256) {
    // The `_l2CallValue` is set to `0` when bridging ERC20 tokens.
    uint256 _l2CallValue = 0;

    return
        sendTxToL2CustomRefund(
            inbox,
            counterpartGateway,
            _refundTo,
            _from,
            tokenTotalFeeAmount,
            _l2CallValue,
            L2GasParams({
                _maxSubmissionCost: _maxSubmissionCost,
                _maxGas: _maxGas,
                _gasPriceBid: _gasPriceBid
            }),
            _data
        );
}
```

Figure B.7: `tokenTotalFeeAmount` is mapped to `_l1CallValue`. (`L1OrbitERC20Gateway`)

- Be consistent with a function naming convention of prepending an underscore "_" for internal functions. This can help detect which functions are important for access control checks.

```
function inboundEscrowTransfer(
    address _l1Token,
    address _dest,
    uint256 _amount
) internal virtual {
    // this method is virtual since different subclasses can handle escrow
differently
    IERC20(_l1Token).safeTransfer(_dest, _amount);
}
```

```
/**
 * @dev Only excess gas is refunded to the _refundTo account, l2 call value is
always returned to the _to account
 */
function createOutboundTxCustomRefund(
    address _refundTo,
    address _from,
    uint256, /* _tokenAmount */
    uint256 _maxGas,
    uint256 _gasPriceBid,
    uint256 _maxSubmissionCost,
    bytes memory _outboundCalldata
) internal virtual returns (uint256) {
```

Figure B.8: `inboundEscrowTransfer` and `createOutboundTxCustomRefund` do not follow the convention seen for internal functions. (`L1ArbitrumGateway`)