Arbitrum Token Bridge Creator

Security Assessment (Summary Report)

July 29, 2024

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

Prepared by: **Jaime Iglesias, Troy Sargent, Gustavo Grieco, 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

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

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:

Troy Sarget, Consultant
troy.sargent@trailofbits.com

Kurt Wilis, Consultant
kurt.wilis@trailofbits.com

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Project Timeline

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

Date	Event
December 4, 2023	Pre-project kickoff call
December 18, 2023	Delivery of report draft
December 18, 2023	Report readout meeting
July 29, 2024	Delivery of summary report

Project Targets

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

token-bridge-contracts-private PR#16

Repository	https://github.com/OffchainLabs/token-bridge-contracts-private/pull/16
Version	`PR#16 (ca71072...ed24e1e)`
Type	Solidity
Platform	EVM

token-bridge-contracts-private PR#21

Repository	https://github.com/OffchainLabs/token-bridge-contracts-private/pull/21
Version	`PR#21 (b2e62e1...3c90260)`
Type	Solidity
Platform	EVM

nitro-contracts PR#100

Repository	https://github.com/OffchainLabs/nitro-contracts/pull/100
Version	`PR#100 (b455c31...645e51d)`
Type	Solidity
Platform	EVM

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of its Token Bridge Creator. From December 4 to December 12, 2023, a team of three consultants began a security review of the Token Bridge Creator's source code. From December 13 to December 15, 2023, a team of three consultants began a security review of a PR to the Nitro contract's codebase.

Observations and Impact

Token Bridge Creator is a set of smart contracts used for the atomic and permissionless deployment of token bridge contracts on top of an existing rollup. Its main use-case is to make deployment and/or configuration of Orbit L3 chains more convenient. The contract can be used for both L1<>L2 and L2<>L3 setups.

Additionally, we reviewed a number of small changes to the Nitro Contracts codebase.

In total, we uncovered eight issues, most of which are related to assumptions made by the design of the protocol. One high-severity issue, TOB-ARB-TBC-002, could allow a malicious actor to grief the deployment of a bridge. This issue and others were resolved during the fix review.

Summary of Findings

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

ID	Title	Type	Severity
1	L2 runtime code does not contain constructor code	Access Controls	**Informational**
2	L2 token bridge contract deployment can be griefed	Denial of Service	**High**
3	Incorrect L2 Multicall address predicted	Configuration	**Low**
4	Rollup owner is assumed to be an EOA	Data Validation	**Informational**
5	Depositing before the token bridge is fully deployed can result in loss of funds	Data Validation	**Medium**
6	Dangerous aliasing assumption	Data Validation	**Low**
7	Unclear decimal units of provided amounts	Data Validation	**Low**
8	Token values in DeployHelper are not adjusted to token decimals	Data Validation	**Medium**

Detailed Findings

1. L2 runtime code does not contain constructor code	
Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-ARB-TBC-001
Target: `contracts/tokenbridge/arbitrum/L2AtomicTokenBridgeFactory.sol`	

Description

Contracts that are being deployed to L2 via retryable tickets on L1 do not include information on their creation code. This can be dangerous and lead to inconsistencies in state.

The contracts deployed to L2 are encoded in the `L1TokenBridgeRetryableSender`'s `sendRetryable` function.

```
bytes memory data = abi.encodeCall(
    L2AtomicTokenBridgeFactory.deployL2Contracts,
    (
        L2RuntimeCode(
            l2.routerTemplate.code,
            l2.standardGatewayTemplate.code,
            l2.customGatewayTemplate.code,
            l2.wethGatewayTemplate.code,
            l2.wethTemplate.code,
            l2.upgradeExecutorTemplate.code,
            l2.multicallTemplate.code
            ),
        l1.router,
        l1.standardGateway,
        l1.customGateway,
        l1.wethGateway,
        l1.weth,
        l2StandardGatewayAddress,
        rollupOwner,
        aliasedL1UpgradeExecutor
    )
);
```

Figure 1.1: The L2 contracts' template code is included in a retryable TX.
(`L1TokenBridgeRetryableSender.sol`)

In order to deploy the code on the L2 side, a generic constructor code is used to deploy the given runtime code.

```
// create L2 router logic and upgrade
address routerLogic = Create2.deploy(
    0, OrbitSalts.UNSALTED, CreationCodeHelper.getCreationCodeFor(runtimeCode)
);
```

Figure 1.2: The L1 provided runtime code is wrapped with a generic constructor code. (`L2AtomicTokenBridgeFactory.sol`)

The effect is that the original constructor is stripped away, which can be dangerous and lead to errors. For example, this could result in the removal of disabling initializers for proxy implementations, or it could lead to referencing invalid addresses on L2 due to immutable addresses included in the runtime code that were valid on L1 (e.g., corrupting the `DelegateCallAware`’s `onlyDelegated` modifier).

Exploit Scenario

In another upgrade, an implementation contract that is deployed on L2 is left uninitialized. A malicious user initializes the implementation and is able to self destruct it.

Recommendations

Short term, consider passing in the contract’s creation code instead of the runtime code.

Long term, ensure that all proxy and logic contracts are initialized correctly by adding further tests to any kind of contract deployed via a ticket.

2. L2 token bridge contract deployment can be griefed	
Severity: **High**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-ARB-TBC-002
Target: `contracts/tokenbridge/arbitrum/L2AtomicTokenBridgeFactory.sol`	

Description

The retryable ticket deploying the L2 token bridge contracts can fail if the call is front-run resulting in a blocked state.

When creating the token bridge, `L1AtomicTokenBridgeCreator` sends out two retryable tickets, one for creating the `L2AtomicTokenBridgeFactory` contract and one for calling `L2AtomicTokenBridgeFactory.deployL2Contracts`. These tickets are expected to be atomic in the sense that they are all executed together and guaranteed to succeed.

However, once both retryable tickets are created and pending, a malicious user has the chance to manually redeem the first ticket, creating the `L1AtomicTokenBridgeCreator`, and then insert a transaction before the second ticket is redeemed. If the user includes a call to `L2AtomicTokenBridgeFactory.deployL2Contracts` from any account other than the expected `L1TokenBridgeRetryableSender`, then the L2 contracts will be deployed at non-canonical addresses and will not match the addresses stored in the contract.

The L2 contract addresses are dependent on the sender. This can be seen in the `_getProxyAddress` function, which computes the address of a `TransparentUpgradeableProxy` that is deployed by the `L2AtomicTokenBridgeFactory` using `create2` given the salt calculated from the prefix, the chain ID, and the sender. This sender is expected to be the `L1TokenBridgeRetryableSender`.

```
function _getProxyAddress(bytes memory prefix, uint256 chainId)
    internal
    view
    returns (address)
{
    return Create2.computeAddress(
        _getL2Salt(prefix, chainId),
        keccak256(
            abi.encodePacked(
                type(TransparentUpgradeableProxy).creationCode,
                abi.encode(
```

```
                    canonicalL2FactoryAddress, _predictL2ProxyAdminAddress(chainId),
bytes("")
                )
            )
        ),
        canonicalL2FactoryAddress
    );
}

//...

function _getL2Salt(bytes memory prefix, uint256 chainId) internal view returns
(bytes32) {
    return keccak256(
        abi.encodePacked(
            prefix, chainId,
AddressAliasHelper.applyL1ToL2Alias(address(retryableSender))
        )
    );
}
```

Figure 2.1: The L2 proxy address is computed. (`L1AtomicTokenBridgeCreator.sol`)

The `deployL2Contracts` function can therefore be kept permissionless—since only the deployments coming from the `L1TokenBridgeRetryableSender` are considered the canonical ones for the rollup. However, some of the contracts are deployed independently of the caller, at fixed addresses using an unsalted `create2` call, such as the `UpgradeExecutor` logic contract.

```
// Create UpgradeExecutor logic and upgrade to it.
address upExecutorLogic = Create2.deploy(
    0, OrbitSalts.UNSALTED, CreationCodeHelper.getCreationCodeFor(runtimeCode)
);
```

Figure 2.2: The upgrade executor logic is deployed at a fixed address on L2. (`L2AtomicTokenBridgeFactory.sol`)

As the address is fixed, the contract cannot be redeployed to the same address once it has already been created. This essentially blocks any further calls to `deployL2Contracts`. In particular, this means that it is possible to block the second retryable ticket (coming from the `L1TokenBridgeRetryableSender`) that deploys the contracts at the precomputed addresses. The result is a mismatch in the stored deployment addresses, requiring a manual recovery.

Exploit Scenario

Bob, a malicious user, waits for the rollup owner to call `createTokenBridge`, starting the token bridge deployment process. On the rollup chain, Bob then manually redeems the first ticket, which creates the L2 token bridge factory, and then calls `deployL2Contracts` from his own address. The rollup owner is not aware that their call fails. After many failed bridging attempts, the issue is discovered. The bridged funds are stuck, and the rollup owner is unable to recreate the L2 token bridge contracts via `createTokenBridge`.

Recommendations

Short term, ensure that the L2 bridge contract creation does not end up being blocked. Either deploy the currently unsalted contracts using `create1` or make the `create2` salt dependent on the sender by using `_getL2Salt(OrbitSalts.UNSALTED)`.

Long term, critically examine whether assumptions—such as sending multiple retryable tickets being atomic—are always valid and whether these can be exploited.

3. Incorrect L2 Multicall address predicted	
Severity: **Low**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-ARB-TBC-003
Target: `contracts/tokenbridge/ethereum/L1AtomicTokenBridgeCreator.sol`	

Description

The calculation of the deployment address for the L2 Multicall is incorrect.

The code for `createTokenBridge` predicts that the address of `Multicall` will be deployed on the L2 using the `.codehash` value of the `Multicall` template.

```
function _predictL2Multicall(uint256 chainId) internal view returns (address) {
    return Create2.computeAddress(
        _getL2Salt(OrbitSalts.L2_MULTICALL, chainId),
        l2MulticallTemplate.codehash,
        canonicalL2FactoryAddress
    );
}
```

Figure 3.1: The L2 Multicall address is predicted (L1AtomicTokenBridgeCreator.sol)

A contract's code hash is the `keccak256` hash of the contract's runtime code. The `create2` opcode, however, computes the address using the contract's creation code.

The `Multicall` contract is created using a retryable ticket containing the runtime code and wraps it with a generic creation code.

```
// deploy multicall
Create2.deploy(
    0,
    _getL2Salt(OrbitSalts.L2_MULTICALL),
    CreationCodeHelper.getCreationCodeFor(l2Code.multicall)
);
```

Figure 3.2: The Multicall contract is deployed on L2 using `create2` (`L2AtomicTokenBridgeFactory.sol`)

This disparity causes the `createTokenBridge` contract to predict an incorrect address for the L2 `Multicall` contract.

It is worth noting that this issue was not discovered during testing because the tests do not check the initialization of the `L2Multicall` contract properly; etherj's `getCode` returns

`"0x"` when the account has no code instead of returning empty bytes (" "). Therefore, the `expect` statement in figure 3.3 passes for a non-deployed contract (i.e., because `"0x".length > 0`).

```
async function checkL2MulticallInitialization(l2Multicall: ArbMulticall2) {
  // check l2Multicall is deployed
  const l2MulticallCode = await l2Provider.getCode(l2Multicall.address)
  expect(l2MulticallCode.length).to.be.gt(0)
}
```

Figure 3.3: The `Multicall` contract initialization unit test (`tokenBridgeDeploymentTest.ts`)

Exploit Scenario

A user makes RPC `Multicalls` on the rollup using the address stored in the L1 contract. However, the RPC calls fail because the `Multicall` address is invalid.

Recommendations

Short term, fix the precomputed address calculation or consider using the contract creation code directly.

```
function _predictL2Multicall(uint256 chainId) internal view returns (address) {
    return Create2.computeAddress(
        _getL2Salt(OrbitSalts.L2_MULTICALL, chainId),
        keccak256(CreationCodeHelper.getCreationCodeFor(l2MulticallTemplate.code)),
        canonicalL2FactoryAddress
    );
}
```

Figure 3.3: The L2 Multicall address prediction is fixed

Long term, include end-to-end tests checking that the computed address actually matches the deployed address. Additionally, consider checking that the actual deployed code matches the expected template instead of checking whether or not the address has code.

Finally, whenever an external API (e.g., etherjs) is used, it is of utmost importance to check the documentation to ensure that the values returned by the functions are the expected ones.

4. Rollup owner is assumed to be an EOA	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARB-TBC-004
Target: `contracts/tokenbridge/ethereum/{L1AtomicTokenBridgeCreator,L2AtomicTokenBridgeFactory}.sol`	

Description

The rollup owner is currently assumed to be an EOA (externally owned account); however, this is neither explicitly checked nor verified.

Some proxy logic contracts must first be initialized to protect against an unexpected initialization that can cause the execution of critical operations (e.g., `selfdestruct`).

```
// sweep the balance to send the retryable and refund the difference
// it is known that any eth previously in this contract can be extracted
// tho it is not expected that this contract will have any eth
retryableSender.sendRetryable{value: isUsingFeeToken ? 0 : address(this).balance}(
    RetryableParams(
        inbox,
        canonicalL2FactoryAddress,
        msg.sender,
        msg.sender,
        maxGasForContracts,
        gasPriceBid
    ),
    L2TemplateAddresses(
        l2RouterTemplate,
        l2StandardGatewayTemplate,
        l2CustomGatewayTemplate,
        isUsingFeeToken ? address(0) : l2WethGatewayTemplate,
        isUsingFeeToken ? address(0) : l2WethTemplate,
        address(l1Templates.upgradeExecutor),
        l2MulticallTemplate
    ),
    l1Deployment,
    l2Deployment.standardGateway,
    rollupOwner,
    msg.sender,
    AddressAliasHelper.applyL1ToL2Alias(upgradeExecutor),
    isUsingFeeToken
);
```

Figure 4.1: The rollup owner address is passed as a parameter to the retryable ticket (`L1AtomicTokenBridgeCreator.sol`)

The address is then included as an executor role in the upgrade executor contract.

```
// init upgrade executor
address[] memory executors = new address[](2);
executors[0] = rollupOwner;
executors[1] = aliasedL1UpgradeExecutor;
IUpgradeExecutor(canonicalUpgradeExecutor).initialize(canonicalUpgradeExecutor,
executors);
```

Figure 4.2: The rollup owner is given the executor role in the upgrade executor (`L2AtomicTokenBridgeFactory.sol`)

This implicitly assumes that the rollup owner will always be an EOA, as otherwise, if it was a contract, the address would be aliased to an L2 address. If this were the case, it would not be able to make the calls to the upgrade executor, because it has stored the unaliased L1 address.

This assumption is not clearly stated and not verified on-chain. In order to prevent centralization issues, the rollup's owner should be expected to be a timelock-controlled multisig contract.

Exploit Scenario

A multisig rollup owner creates a token bridge. The rollup owner is added as an executor to the upgrade executor. However, the owner is now unable to make any upgrade calls because the unaliased address has been stored.

Recommendations

Short term, document the assumption that the rollup owner is expected to be an EOA and consider explicitly checking whether or not the rollup owner is a contract. Additionally, both the aliased and unaliased address could be given executor control in the upgrade executor.

Long term, revisit assumptions that may not be explicitly stated; include explicit checks for these, or ensure that no unexpected scenario is created.

5. Depositing before the token bridge is fully deployed can result in loss of funds

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARB-TBC-005
Target: `contracts/tokenbridge/ethereum/L1AtomicTokenBridgeCreator.sol`	

Description

If a user triggers a deposit in the L1 side of the token bridge before it is fully deployed, their deposit will not be executed as expected, and their funds will not be minted on the L2.

Token bridge creation relies on the usage of two retryable tickets that will correctly set up the L2 side of the bridge.

```
    /**
     * @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.
     */
```

Figure 5.1: Documentation on the deployment of the token bridge (`L1AtomicTokenBridgeCreator.sol`)

However, if the both tickets are not immediately redeemed, a deposit from the L1 will produce an incomplete deposit in L2.

Exploit Scenario

Alice starts the process of the token bridge deployment. A spike in the L2 activity causes the retryable tickets not to execute immediately. Bob is eager to use the L2, so he quickly deposits into the token bridge, even though the bridge is not fully deployed yet. If Bob's retryable ticket with the deposit is executed before the L2 of the bridge is ready, the retryable ticket will call an empty account and it will not revert, leaving Bob without the L2 counterpart of his tokens.

Recommendations

Short term, consider adding a front-end check to verify that the L2 counterpart deployment has succeeded. Provide a clear error to the client that the token bridge is not yet fully deployed yet and that any deposit will result in loss of funds.

Long term, review the assumptions of the token bridge during its usage to ensure that the new deployment will not break any of them.

6. Dangerous aliasing assumption	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ARB-TBC-006
Target: `nitro-contracts/src/bridge/AbsInbox.sol`	

Description

Applying the L1-to-L2 alias for user-provided addresses depends on L1 contracts, which can cause aliasing addresses to point to invalid addresses.

Both the `excessFeeRefundAddress` and the `callValueRefundAddress`—two addresses that the user provides when `createRetryableTicket` is called—are aliased depending on whether the L1 address contains code.

```
// if a refund address is a contract, we apply the alias to it
// so that it can access its funds on the L2
// since the beneficiary and other refund addresses don't get rewritten by arb-os
if (AddressUpgradeable.isContract(excessFeeRefundAddress)) {
    excessFeeRefundAddress =
AddressAliasHelper.applyL1ToL2Alias(excessFeeRefundAddress);
}
if (AddressUpgradeable.isContract(callValueRefundAddress)) {
    // this is the beneficiary. be careful since this is the address that can cancel
the retryable in the L2
    callValueRefundAddress =
AddressAliasHelper.applyL1ToL2Alias(callValueRefundAddress);
}
```

Figure 6.1: Aliasing of user provided addresses (`AbsInbox.sol`)

Because it is not possible to reliably determine whether the user wants to use an aliased or unaliased version of a given L2 address, this step can lead to mistakes by erroneously applying an alias where it was not expected. This could cause the refund to be sent to a nonexistent address on L2, where it could be recovered only from its L1 counterpart, which could be an immutable contract.

Exploit Scenario

The following events occur:

- Alice (EOA) deploys a random token contract using nonce 0 at address `0xabc` on L1.
- Alice also deploys a multisig contract using nonce 0 at address `0xabc` on L2.

- Alice creates a retryable ticket and specifies the `callValueRefundAddress` as the L2 multisig (at `0xabc`).
- The alias check converts `0xabc` to a nonexistent address on L2 due to the unrelated L1 token contract.
- Only L1 token contract is able to recover funds, but it does not contain logic to do so, as it is an immutable token contract.

Recommendations
Short term, clearly document the behavior and make it clear to a user that an alias will apply in certain cases. Consider not making any assumptions on behalf of the user and include off-chain checks and validations in a web app.

Long term, consider adding further on-chain checks to ensure that the L1 contract is able to send retryable tickets and recover the funds when applying an alias.

7. Unclear decimal units of provided amounts	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ARB-TBC-007
Target: `nitro-contracts/src/bridge/AbsInbox.sol`	

Description

When creating retryable tickets, the caller must provide multiple token values in various units, which could be prone to errors.

The `createRetryableTicket` function requires the user to provide various parameters in different units.

```
function _createRetryableTicket(
    address to,
    uint256 l2CallValue,
    uint256 maxSubmissionCost,
    address excessFeeRefundAddress,
    address callValueRefundAddress,
    uint256 gasLimit,
    uint256 maxFeePerGas,
    uint256 amount,
    bytes calldata data
) internal returns (uint256) {
    // ensure the user's deposit alone will make submission succeed
    uint256 amountToBeMintedOnL2 = _fromNativeTo18Decimals(amount);
    if (amountToBeMintedOnL2 < (maxSubmissionCost + l2CallValue + gasLimit *
maxFeePerGas)) {
        revert InsufficientValue(
            maxSubmissionCost + l2CallValue + gasLimit * maxFeePerGas,
            amountToBeMintedOnL2
        );
    }

    // ...
)
```

Figure 7.1: The `_createRetryableTicket` function (`AbsInbox.sol`)

The parameter `amount` is given in the native token's decimal units. The parameters `l2CallValue`, `maxSubmissionCost`, and `maxFeePerGas` are denominated using 18 decimal units.

Neither the parameter names nor the NatSpec comments suggest that the values are given in differing units, which can cause mistakes in integration.

Exploit Scenario

An optimistic cross-chain bridge and AMM protocol is built on top of Arbitrum. When integrating with an Orbit chain with non-standard decimals, the incorrect decimal units are hard coded into the token handler contract, causing values that are too high to be sent to the Orbit chain when bridging the native token.

Recommendations

Short term, consider making a clear distinction between the units depending on the chain. For example, on L1, all values will always be denominated in the native token's decimal units, and on L2, all values will always be denominated using 18 decimal points.

Long term, be aware of confusions that can arise when assumptions in the API are not clearly documented. Aim to remove the risk of mistakes by focusing on clear usability when interacting with the bridge contracts.

8. Token values in DeployHelper are not adjusted to token decimals

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ARB-TBC-008
Target: `nitro-contracts/src/rollup/DeployHelper.sol`	

Description

The `DeployHelper` contract deploys helper contracts to the L2 chain through retryable tickets containing hard-coded values that could be chain- and token decimal-dependent.

Certain helper contracts can be deployed in the `DeployHelper` contract as part of the rollup creation process. These helper contracts are sent via signed transactions.

```
// Nick's CREATE2 Deterministic Deployment Proxy
// https://github.com/Arachnid/deterministic-deployment-proxy
address public constant NICK_CREATE2_DEPLOYER =
0x3fAB184622Dc19b6109349B94811493BF2a45362;
uint256 public constant NICK_CREATE2_VALUE = 0.01 ether;
bytes public constant NICK_CREATE2_PAYLOAD = hex"04f8a58085174876e80083...";
```

Figure 8.1: Aliasing of user-provided addresses (`AbsInbox.sol`)

Before sending these L2 transactions, a retryable ticket is first created in order to fund the deployer address.

```
uint256 feeAmount = _value + submissionCost + GASLIMIT * maxFeePerGas;

// fund the target L2 address
if (_isUsingFeeToken) {
    IERC20Inbox(inbox).createRetryableTicket({
        to: _l2Address,
        l2CallValue: _value,
        maxSubmissionCost: submissionCost,
        excessFeeRefundAddress: msg.sender,
        callValueRefundAddress: msg.sender,
        gasLimit: GASLIMIT,
        maxFeePerGas: maxFeePerGas,
        tokenTotalFeeAmount: feeAmount,
        data: ""
    });
} else {
```

Figure 8.2: The token total fee amount is being calculated (`DeployHelper.sol`)

The fee amount is computed in order to cover the L2 value, the submission cost (`0` in the case of using a fee token), and the retryable TX gas cost.

When creating a retryable ticket, the `tokenTotalFeeAmount` parameter is expected to be given in the native token decimal units, as it is converted to 18 decimals. This means that if the custom fee token's decimals differ, then the actual token value that is sent will be miscalculated. When the bridge receives a token value that is too little, it will transfer the missing funds from `msg.sender` (`DeployHelper`), causing the transaction to revert.

Exploit Scenario

Alice creates a new rollup with her custom fee token that uses six decimal points. When deploying the helper contracts, due to the miscalculation, the `Inbox` requests a large amount of tokens (0.01e18 * (1e18 - 1e6) = 10M tokens) from the `DeployHelper`. As the `DeployHelper` has not given any token spending approval to the `Inbox`, this would result in a transaction failure. If the tokens are manually sent to the `Inbox`, a large part would be stuck in an unrecoverable address.

Conversely, if the fee token uses decimal points greater than 18, then it is possible that too little value would be sent for the successful contract creation.

Recommendations

Short term, convert the value given in 18 decimal units to the native token decimal units (rounded up if needed).

Long term, implement further testing that includes bounds on expected deployment costs. Additionally, avoid patterns that can cause confusion in function APIs.

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. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From December 14 to December 15, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Offchain Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the eight issues described in this report, OffchainLabs has resolved three issues and left five issues unresolved. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	L2 runtime code does not contain constructor code	Unresolved
2	L2 token bridge contract deployment can be griefed	Resolved
3	Incorrect L2 Multicall address predicted	Resolved
4	Rollup owner is assumed to be an EOA	Resolved
5	Depositing before the token bridge is fully deployed can result in loss of funds	Unresolved
6	Dangerous aliasing assumption	Unresolved
7	Unclear decimal units of provided amounts	Unresolved
8	Token values in DeployHelper are not adjusted to token decimals	Unresolved

Detailed Fix Review Results

TOB-ARB-TBC-001: L2 runtime code does not contain constructor code
Unresolved.

TOB-ARB-TBC-002: L2 token bridge contract deployment can be griefed
Resolved in PR 26 and PR 28. Contracts are no longer being deployed using the unsalted method, which could lead to front-running issues; instead, all contracts include the aliased retryable sender as part of the deployment salt, therefore linking it with the sender. Additionally, the ability to resend the L2 deployment retryable ticket has been added which would help mitigate the scenario in which the original tickets are redeemed out of order blocking the deployment.

TOB-ARB-TBC-003 Incorrect L2 Multicall address predicted
Resolved in PR 27, the contract's runtime code hash is no longer being used to predict the L2 address; instead its creation code is. Additionally, the tests have been updated to check that the deployed contract code matches that of the expected template and the previous code length checks have been updated to match etherjs's behavior.

TOB-ARB-TBC-004: Rollup owner is assumed to be an EOA
Resolved in PR 27, the rollup owner is no longer assumed to be an EOA - instead a check is performed to determine whether it's a contract or not and, in the case that it is, the address is aliased mitigating the issue.

TOB-ARB-TBC-005: Depositing before the token bridge is fully deployed can result in loss of funds
Unresolved.

TOB-ARB-TBC-006: Dangerous aliasing assumption
Unresolved.

TOB-ARB-TBC-007: Unclear decimal units of provided amounts
Unresolved.

TOB-ARB-TBC-008: Token values in DeployHelper are not adjusted to token decimals
Unresolved.