Arbitrum Governance

Security Assessment

January 6, 2023

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

Prepared by: **Gustavo Grieco, Troy Sargent, Jaime Iglesias, Nat Chin, and Tarun Bansal**

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.
228 Park Ave S #80688
New York, NY 10003
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

Executive Summary

Engagement Overview

Arbitrum engaged Trail of Bits to review the security of its Governance and Bridge contracts. From October 17 to December 9, 2022, a team of five consultants conducted a security review of the client-provided source code, with 22 person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system. We had access to the source code and documentation. We performed static and manual testing of the target system and its codebase, using both automated and manual processes

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	8
Low	3
Informational	5
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	3
Auditing and Logging	2
Data Validation	13
Undefined Behavior	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **Insufficient access control implementations (TOB-ARBGOV-3, TOB-ARBGOV-6, TOB-ARBGOV-9)**
  The lack of access controls across the system poses a variety of risks for the contracts.

- **Missing data validation and unclear assumptions (TOB-ARBGOV-1, TOB-ARBGOV-5, TOB-ARBGOV-12, TOB-ARBGOV-13)**
  Assumptions made for each section of the system lack a clear specification, which makes it difficult to track where validation is occurring or missing.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

The following engineers were associated with this project:

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Jaime Iglesias, Consultant
jaime.iglesiasbotas@trailofbits.com

Nat Chin, Consultant
natalie.chin@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Project Timeline

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

Date	Event
October 21, 2022	Status update meeting #1
October 28, 2022	Status update meeting #2
November 4, 2022	Status update meeting #3
November 10, 2022	Status update meeting #4
November 18, 2022	Status update meeting #5
December 2, 2022	Status update meeting #6
December 12, 2022	Delivery of report draft
December 12, 2022	Report readout meeting
January 6, 2023	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Arbitrum Governance and Token Bridge contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Regarding the Arbitrum Governance:
    - Can any user other than a member of the security council block or cancel a proposal or an upgrade?
    - Is the usage of retryable tickets safe for proposals or upgrades?
    - Is voting-related information (e.g., votes, quorum) tracked properly?
    - Are the treasury tokens excluded from the quorum computation?
    - Are the vested funds properly protected?
- Regarding the Token Bridge:
    - Is it possible to steal funds?
    - Are there gaps between expected behavior in the protocol and its implementation?
    - Do the bridge contracts adequately protect against a variety of tokens?
    - Does the minting- and burning-related arithmetic hold?
    - Are all the assumptions upon which each contract relies explicitly outlined?
    - Does the bridge adequately track token mappings between L1 and L2?

Project Targets

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

Arbitrum Governance

Repository	https://github.com/OffchainLabs/governance/

Version	`e4562df1fd165042f6fc5400063dd4a23d10e855`
	`6bd1a880df96b36508b55a10c8f29327711f3750`
	`cf6762d45678847cd901544f90989d852dfdd6ea`

Type	Solidity
Platform	Ethereum

Arbitrum Token Bridge

Repository	https://github.com/OffchainLabs/token-bridge-contracts/
Version	`b30dfcda019a5756061638677dae9b28384c7275`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

Governance contracts:

- **L1ArbitrumToken:** This contract is the L1 representation of the Arbitrum tokens ERC20 contracts. We manually verified how this token is registered in the token bridge infrastructure and whether it behaves as expected as a “reverse token.”

- **TokenDistributor:** This contract holds the Arbitrum ERC20 tokens and the registry for users to claim during an airdrop. We manually verified that users can claim only when they have a positive amount available and before a certain deadline. We also checked how the remaining tokens are collected after the deadline passes.

- **FixedDelegateErc20Wallet:** This contract serves as the treasury for Arbitrum tokens, which can be spent using a proposal. We verified that the access controls are in place in order to delegate or transfer funds.

- **L1ArbitrumTimelock and its factory:** Timelock is deployed on L1 for executing proposals and forwarding them to L2. We manually reviewed this contract to ensure that only the expected users are allowed to execute proposals. We also checked that the proposal execution is performed correctly, in particular when retryable tickets are used. Additionally, we verified that the contract that sets up the timelock defines minimal permissions for users as well as other contracts.

- **L2ArbitrumGovernor and its factory:** This standard governor contract has some special functionality to avoid counting votes of some excluded tokens. This contract also allows an owner to set parameters by calling the `relay` function. We also verified that the contract that sets up the governance defines minimal permissions for users as well as other contracts.

- **ArbitrumVestingWallet and its factory:** This wallet vests over time tokens that have been created with a specific factory contract. The full token allowance can be used for delegating and voting immediately. We manually verified how tokens are released over time, as well as their availability for delegation and voting.

- **UpgradeExecutor:** This contract performs arbitrary actions that facilitate upgradeability. It does not contain upgrade logic itself, only the means to call upgrade contracts and execute them.

- Auxiliary code and dependencies, such as the `L1ArbitrumMessenger` and the `Util` contracts, were included in the scope.

Token Bridge contracts:

These contracts allow users to trustlessly move funds (e.g., ERC20 tokens) between Ethereum and Arbitrum. We reviewed the contracts with a focus on the reverse bridge contract, which is the new code introduced by Offchain Labs to allow L1 tokens with a total supply tracked in L2. We looked for flaws in the access controls that could allow attackers to execute unauthorized actions, such as unauthorized calls to functions on the L1/L2 Gateways.

We checked that inbound and outbound messages are properly crafted and processed. We reviewed the process of depositing tokens into an L1 contract and minting the same number of tokens on L2 (as well as the process of burning tokens on L2 and enabling a withdrawal on L1). Additionally, we reviewed the three gateway types and looked for ways that a malicious token could abuse the default gateways. We considered interactions initiated from within the router contract and from outside of it.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Integral review of the token bridge contracts, in particular of interactions with less-common ERC token standards such as ERC777

- Nitro smart contracts (Inbox, Bridge, etc.)

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Arithmetic operations are very limited across the reviewed contracts; however, we have identified a couple of instances in which the assumptions made around the arithmetic operations do not hold (TOB-ARBGOV-13, TOB-ARBGOV-19).	**Moderate**
Auditing	Although most functions emitted events, we found an instance where events were missing (TOB-ARBGOV-2). Additionally, neither an incident response plan nor monitoring was provided during the review.	**Moderate**
Authentication / Access Controls	The extensive composability makes it difficult to track how components interoperate. The lack of documentation on different actors and the number of privileged operations increases the likelihood of issues such as TOB-ARBGOV-3, TOB-ARBGOV-6, and TOB-ARBGOV-9.	**Moderate**
Complexity Management	Because interactions involve on-chain and off-chain components, it is difficult to understand certain operations that involve multiple codebases and component interactions. As a result, the reviewer must have a holistic understanding of the system in order to review its parts in isolation. Moreover, the codebase lacked documentation that highlighted assumptions being made across different components, which required us to spend a substantial amount of time understanding intricate details of how the components worked together.	**Moderate**

	This underscores the need for additional documentation in the codebase.	
Decentralization	With the introduction of governance, the Offchain Labs team began to decentralize important system-related decisions, such as upgrades. However, in practice, Offchain Labs will need to re-evaluate the level of decentralization according to the composition of the 9/12 council, the distribution of the voting token, and the ownership configuration of several critical infrastructure components (e.g., the token bridge gateways).	**Moderate**
Documentation	Although the Arbitrum documentation provides a sufficient high-level overview of the system, it lacks an in-depth detail of the implementation, with several system invariants still undocumented. Additionally, the codebase lacks documentation on the location of data validation. We recommend updating the Arbitrum documentation, especially regarding assumptions. Finally, the Arbitrum team should create a specification for tokens that intend to use Arbitrum's bridge, similar to the specification used for the standardization of gateways. This will help define and document assumptions regarding the bridge operation and what to expect from tokens that want to integrate with the bridge.	**Moderate**
Front-Running Resistance	Although we found no front-running issues on the current contracts, it will not be clear whether cross-chain interactions are vulnerable to front-running until a specific audit focuses on evaluating this aspect.	**Further Investigation Required**
Low-Level Manipulation	The contracts under review contain only minor usage of low-level data manipulation. However, the lack of contract existence (TOB-ARBGOV-15, TOB-ARBGOV-18) impacts the codebase and should be carefully evaluated.	**Moderate**

Testing and Verification	The codebase contains a number of unit and integration tests. However, the tests are insufficient to catch most important issues, which indicates that there are a number of blind spots that should be covered. We noted that bridge contracts can benefit from additional edge case tests for gateway and router contracts. We recognize that doing effective cross-chain testing is difficult and that the available tooling is not sufficiently mature.	**Moderate**

Summary of Recommendations

The Arbitrum Governance and TokenBridge contracts are works in progress with multiple planned iterations. Trail of Bits recommends that Arbitrum address the findings detailed in this report and take the following additional steps prior to deployment:

- **Design a flowchart outlining function calls throughout the system and who is expected to call them.** This flowchart should identify the functions that are protected by access controls and the internals that each function executes.
- **Create a standardized specification for tokens that intend to use Arbitrum's bridge.** This helps define assumptions that the bridge makes about its tokens, and helps users judge tokens accordingly.
- **Integrate reliable unit tests in the system and the continuous integration pipeline.**
    - The unit tests should test all "happy paths" and expected revert flows in the contracts before they can be deployed.
    - Ensure that all tests have reasonable bounds in the system to ensure that they cover corner cases.
- **Identify and analyze all system properties that are expected to hold.** This effort should include analyzing the properties of critical arithmetic operations, identifying potential edge cases, and checking that rounding occurs in the correct direction. The number of issues related to rounding that we found during this audit suggest that more rounding issues may be present in the codebase.
- **Document assumptions that are made between L1 and L2.** Explicit data validation across platforms and languages is especially important for a codebase of this level of complexity.

Summary of Findings

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

ID	Title	Type	Severity
1	Missing zero address checks	Data Validation	Informational
2	Missing event for state variable update	Auditing and Logging	Informational
3	Anyone can reduce the vote power of vested tokens	Access Control	Medium
4	Users can reduce quorum delegating their votes to the DAO treasury	Undefined Behavior	Low
5	Assumptions about OpenZeppelin contracts integration should be reviewed	Data Validation	Medium
6	Anyone can initialize an ArbitrumTimelock implementation contract	Access Control	Informational
7	TokenDistributor does not allow to change recipients once configured	Data Validation	Medium
8	Governance proposal that rely on retryable tickets could be disabled	Auditing and Logging	Low
9	Admin role of L2GovernanceFactory on UpgradeExecutor is not revoked	Access Control	Informational
10	Missing contract size check	Data Validation	Undetermined
11	L2ArbitrumGateway trusts L2Token to return correct l1Address	Data Validation	High

12	Retryable tickets allow out of order execution of token bridge registration functions	Data Validation	**Medium**
13	Assumption of all tokens being burned in outboundEscrowTransfer	Data Validation	**Medium**
14	Dead code in outboundTransferCustomRefund	Data Validation	**Informational**
15	Lack of contract existence checks in the gateway may not detect failed execution	Data Validation	**Undetermined**
16	Cross-chain message out-of-order execution impacts sequential proposal execution	Data Validation	**Medium**
17	Retryable tickets used in governance proposals can be silently discarded	Data Validation	**Medium**
18	Lack of contract existence checks is error-prone when scheduling transactions through the timelock	Data Validation	**Medium**
19	Potential overflow in TokenDistributor causes imprecise claims	Data Validation	**Low**

Detailed Findings

1. Missing zero address checks

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBGOV-1
Target: `governance/src/TokenDistributor.sol`	

Description

Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

Below we outline two instances we have identified in which incoming arguments are not validated:

- The setter function, `setSweepReceiver`, does not validate that the argument, `_sweepReceiver`, is not the zero address prior to updating the state variable, `sweepReceiver`.

```
function setSweepReciever(address payable _sweepReciever) external onlyOwner {
    sweepReceiver = _sweepReciever;
}
```

Figure 1.1: The setter `setSweepReceiver`

```
function sweep() external {
    [...]
    require(token.transfer(sweepReceiver, leftovers), "TokenDistributor: fail token
transfer");
    [...]
}
```

Figure 1.2: The `sweep` function

However, using a null address will cause the sweep to revert as the token's _transfer function requires that the argument, to, be non-zero.

```
require(to != address(0), "ERC20: transfer to the zero address");
```

Figure 1.3: The `_transfer` function validation of receiver address

- The L2GovernanceFactory contract constructor does not validate the incoming arguments, which can result in the deployment of an unusable contract.

```
constructor(
    address _coreTimelockLogic,
    address _coreGovernorLogic,
    address _treasuryTimelockLogic,
    address _treasuryLogic,
    address _treasuryGovernorLogic,
    address _l2TokenLogic,
    address _upgradeExecutorLogic
) {
    coreTimelockLogic = _coreTimelockLogic;
    coreGovernorLogic = _coreGovernorLogic;
    treasuryTimelockLogic = _treasuryTimelockLogic;
    treasuryLogic = _treasuryLogic;
    treasuryGovernorLogic = _treasuryGovernorLogic;
    l2TokenLogic = _l2TokenLogic;
    upgradeExecutorLogic = _upgradeExecutorLogic;
    proxyAdminLogic = address(new ProxyAdmin());
    step = Step.One;
}
```

Figure 1.4: The constructor in the L2GovernaceFactory contract

Exploit Scenario

Alice creates a new token distributor contract. She sets the zero address as the sweeper, assuming that it will force the destruction of the remaining tokens, but it only blocks the call to the sweep function, disallowing the self-destruction of the contract.

Recommendations

Short term, add the proper validation to incoming arguments.

Long term, ensure input validations across functions are consistent and correct.

2. Missing event for state variable update

Severity: **Informational**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-ARBGOV-2
Target: `governance/src/TokenDistributor.sol`	

Description

The constructor does not emit the event `SweepReceiverSet` when setting the state variable `sweepReceiver`, but it does in the `setSweepReceiver` function.

```
constructor(
    IERC20VotesUpgradeable _token,
    address payable _sweepReceiver,
    address _owner,
    uint256 _claimPeriodStart,
    uint256 _claimPeriodEnd
) Ownable() {
    [...]
    sweepReceiver = _sweepReceiver;
    [...]
}
```

Figure 2.1: `TokenDistributor`'s constructor

This inconsistency would make it more difficult for event indexers to track the value of `sweepReceiver` upon deployment.

Recommendations

Short term, emit the `SweepReceiverSet` event in the constructor.

Long term, review important actions and updates and ensure they are logged appropriately for off-chain monitoring systems.

3. Anyone can reduce the vote power of vested tokens

Severity: **Medium**	Difficulty: **Low**
Type: Access Control	Finding ID: TOB-ARBGOV-3
Target: `governance/src/ArbitrumVestingWallet.sol`	

Description

The contract that vests tokens can be used to vote, but the release function is unprotected. As a result, any user can force a transfer that will reduce the vote power.

Arbitrum governance tokens can be vested in special contracts that will lock them for a certain amount of time. These contracts are based on OZ's VestedWallet but still allow the owner to vote using vested tokens:

```
    function delegate(address token, address delegatee) public onlyBeneficiary {
        IERC20VotesUpgradeable(token).delegate(delegatee);
    }

    /// @notice Claim tokens from a distributor contract
    function claim(address distributor) public onlyBeneficiary {
        TokenDistributor(distributor).claim();
    }

    /// @notice Cast vote in a governance proposal
    function castVote(address governor, uint256 proposalId, uint8 support) public onlyBeneficiary {
        IGovernorUpgradeable(governor).castVote(proposalId, support);
    }
```

Figure 3.1: Voting-related functions in `ArbitrumVestingWallet`

However, since the rest of the code still follow `VestedWallet` code, the release functions are unmodified:

```
function release() public virtual {
    uint256 releasable = vestedAmount(uint64(block.timestamp)) - released();
    _released += releasable;
    emit EtherReleased(releasable);
    Address.sendValue(payable(beneficiary()), releasable);
}

function release(address token) public virtual {
    uint256 releasable = vestedAmount(token, uint64(block.timestamp)) -
released(token);
    _erc20Released[token] += releasable;
    emit ERC20Released(token, releasable);
    SafeERC20.safeTransfer(IERC20(token), beneficiary(), releasable);
}
```

Figure 3.2: Release functions from OZ's `VestingWallet`

Since these functions are not protected, they can be triggered by any user.

Exploit Scenario

Alice vests a number of tokens linearly over a six-month period and uses these tokens to vote in some proposals. Two months into the vesting period, Eve notices that a portion of Alice's tokens can be released and calls the release function, triggering a transfer to the beneficiary. Alice continues voting without realizing that her voting power is lower than expected.

Recommendations

Short term, protect release functions to only allow the beneficiary to call them.

Long term, review all the third-party code that is inherited and overridden to ensure it correctly implements the expected system properties.

4. Users can reduce quorum delegating their votes to the DAO treasury

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARBGOV-4
Target: `governance/src/L2ArbitrumGovernor.sol`	

Description

The governance contract avoids counting the voting power of the DAO treasury in the quorum, but this also allows users to reduce the quorum without risk for them.

Arbitrum governance tokens can be used to vote on different proposals. When the voting contract computes the quorum, it excludes the voting power of the DAO treasury, which will hold tokens for future distribution:

```
/// @notice Get "circulating" votes supply; i.e., total minus excluded vote exclude
address.
function getPastCirculatingSupply(uint256 blockNumber) public view virtual returns
(uint256) {
    return token.getPastTotalSupply(blockNumber)
        - token.getPastVotes(EXCLUDE_ADDRESS, blockNumber);
}

/// @notice Calculates the quorum size, excludes token delegated to the exclude
address
function quorum(uint256 blockNumber)
    public
    view
    override (IGovernorUpgradeable, GovernorVotesQuorumFractionUpgradeable)
    returns (uint256)
{
    return (getPastCirculatingSupply(blockNumber) * quorumNumerator(blockNumber))
        / quorumDenominator();
}
```

Figure 4.1: Quorum computation in `L2ArbitrumGovernor`

The governance contract uses quorum to establish a minimum number of votes for a proposal to pass; the number of votes in favor must exceed the votes against it.

```
function _quorumReached(uint256 proposalId) internal view virtual override returns
(bool) {
    ProposalDetails storage details = _proposalDetails[proposalId];
    return quorum(proposalSnapshot(proposalId)) <= details.forVotes;
}
```

Figure 4.2: Quorum usage in `GovernorCompatibilityBravoUpgradeable`

However, users can still delegate their tokens to the DAO treasure address, which increases its voting power and therefore lowers the quorum:

```
function _delegate(address delegator, address delegatee) internal virtual {
    address currentDelegate = delegates(delegator);
    uint256 delegatorBalance = balanceOf(delegator);
    _delegates[delegator] = delegatee;

    emit DelegateChanged(delegator, currentDelegate, delegatee);

    _moveVotingPower(currentDelegate, delegatee, delegatorBalance);
}
```

Figure 4.3: Delegate implementation from OZ's `ERC20VotesUpgradeable`

Exploit Scenario

Eve poses a large amount of votes. She silently delegates to the DAO treasury excluded address to reduce the quorum needed. Alice wants to create a proposal, so she checks the current quorum amount on-chain and negotiates with a number of delegates to ensure her proposal will have quorum.

When the vote period is about to start, Eve delegates her votes to another address, indirectly increasing the quorum. Alice did not expect that change, and she is unable to reach the quorum with the votes she negotiated.

Recommendations

Short term, document this behavior to ensure users are aware of this issue and actively monitor the blockchain to identify suspicious delegations.

Long term, carefully review the high-level impact of the code changes in any third-party code that is re-used.

5. Assumptions about OpenZeppelin contracts integration should be reviewed

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ARBGOV-5

Target:
- `governance/src/ArbitrumTimelock.sol`,
- `governance/src/L1ArbitrumTimelock.sol`
- `governance/src/L2ArbitrumGovernor.sol`

Description

Arbitrum uses OpenZeppelin's governance modules to implement its governance system. These modules make a series of assumptions about how they should be integrated in larger systems.

Arbitrum relies on OpenZeppelin's contract modules to implement their governance system:

```
contract L2ArbitrumGovernor is
    Initializable,
    GovernorSettingsUpgradeable,
    GovernorCompatibilityBravoUpgradeable,
    GovernorVotesUpgradeable,
    GovernorTimelockControlUpgradeable,
    GovernorVotesQuorumFractionUpgradeable,
    GovernorPreventLateQuorumUpgradeable,
    OwnableUpgradeable
{ … }
```

Figure 5.1: Use of OZ's Governor code to implement `L2ArbitrumGovernor` (old version)

```
// @title L1 timelock for executing proposals on L1 or forwarding them back to L2
/// @dev   Only accepts proposals from a counterparty L2 timelock
contract L1ArbitrumTimelock is TimelockControllerUpgradeable, L1ArbitrumMessenger
{ … }
```

Figure 5.2: Use of OZ's Timelock code to implement `L2ArbitrumGovernor`

The use of these contract modules includes a series of assumptions that are sometimes unclear or require familiarity with Compound-style governance, e.g.:

- The use of `GovernanceBravo` silently limits the weight of the votes to 2**96 - 1. While the OpenZeppelin documentation mentions this limitation for having

COMP-compatible code when creating a token, it fails to mention that limitation in the `GovernanceBravo` documentation.

```
/**
 * @dev See {Governor-_countVote}. In this module, the support follows Governor
Bravo.
 */
function _countVote(
    uint256 proposalId,
    address account,
    uint8 support,
    uint256 weight,
    bytes memory // params
) internal virtual override {
    ProposalDetails storage details = _proposalDetails[proposalId];
    Receipt storage receipt = details.receipts[account];

    require(!receipt.hasVoted, "GovernorCompatibilityBravo: vote already cast");
    receipt.hasVoted = true;
    receipt.support = support;
    receipt.votes = SafeCastUpgradeable.toUint96(weight);
    …
```

Figure 5.3: Implementation of `_countVote` from `GovernorCompatibilityBravoUpgradeable`

When using a simple vote-counting approach, users should be aware that this will change the way in which quorum is reached with respect to `GovernanceBravo`, which will include both positive and abstained votes:

```
contract L2ArbitrumGovernor is
    Initializable,
    GovernorSettingsUpgradeable,
    GovernorCountingSimpleUpgradeable,
    GovernorVotesUpgradeable,
    GovernorTimelockControlUpgradeable,
    GovernorVotesQuorumFractionUpgradeable,
    GovernorPreventLateQuorumUpgradeable,
    OwnableUpgradeable
{ … }
```

Figure 5.4: Use of OZ's Governor code to implement `L2ArbitrumGovernor` (new version)

```
    function _quorumReached(uint256 proposalId) internal view virtual override
returns (bool) {
        ProposalVote storage proposalvote = _proposalVotes[proposalId];
        return quorum(proposalSnapshot(proposalId)) <= proposalvote.forVotes +
proposalvote.abstainVotes;
    }
```

Figure 5.5: `_quorumReached` function from `GovernorCountingSimpleUpgradeable`

Finally, in other instances the documentation does cover the assumptions and/or limitations; however, these should be thoroughly reviewed and documented:

- The `chainId` and the governor address are not part of the proposal ID computation. Consequently, the same proposal (with same operation and same description) will have the same ID if submitted on multiple governors across multiple networks. This also means that in order to execute the same operation twice (on the same governor), the proposer will have to change the description in order to avoid proposal ID conflicts.
- It is not recommended to change the timelock while there are other queued governance proposals, because this can bring unexpected consequences to the other proposals.
- Setting up the `TimelockController` to have additional proposers besides the governor is very risky.
- The use of `TimeLockController` should be carefully considered, as there are instances in which it is being used with a delay of 0. This essentially makes `TimeLockController` redundant while adding additional attacker surface, operational overhead, and gas cost.

Exploit Scenario

The `GovernanceBravo` compatibility module is used with a non-compound voting token that implicitly limits the supply of the token to 2 ** 96 - 1. However, the Arbitrum team is unaware of this sets the supply to a higher value, effectively limiting users' voting capabilities.

Recommendations

Short term, carefully review and document the assumptions and limitations regarding third-party code integrations and consider whether the limitations are acceptable and whether the assumptions hold.

Long term, monitor the third-party code libraries for any relevant changes and security advisories.

6. Anyone can initialize an ArbitrumTimelock implementation contract

Severity: **Informational**	Difficulty: **Low**
Type: Access Control	Finding ID: TOB-ARBGOV-6
Target: `governance/src/ArbitrumTimelock.sol`	

Description

Once the `ArbitrumTimelock` contract implementation is deployed, any user can initialize it. Since the Offchain Labs team initiated this implementation, a malicious user could leverage this issue to conduct phishing attacks.

The `ArbitrumTimelock` contract is an upgradable smart contract with an initializer. It is considered a security best practice to either disable initializers or to initialize the implementation contract at the time of deployment. We have observed that other upgradable contracts include a constructor that disables the initialization.

```
constructor() {
    _disableInitializers();
}
```

Figure 6.1: Constructor in `L2ArbitrumGovernor`

However, this is not the case with the `ArbitrumTimeLock` contract; it has an empty constructor, which allows an attacker to initialize and take control of an uninitialized instance.

Exploit Scenario

Eve notices that the implementation instance of `ArbitrumTimelock` is not initialized and calls the `initialize` function on it with desired arguments. Eve can then market it as an official `ArbitrumTimelock` contract to convince others to send funds to it, especially since it can be proved that the contract is deployed by the Offchain Labs team. All funds sent to this contract will be under Eve's full control and can be stolen at any time.

Recommendations

Short term, disable initializers for the `ArbitrumTimelock` contract during construction.

Long term, carefully review the codebase to make sure that same pattern is followed for upgradable smart contracts.

7. TokenDistributor does not allow to change recipients once configured

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ARBGOV-7
Target: `governance/src/TokenDistributor.sol`	

Description

If the owner makes an error while setting the recipients of the tokens, there is no way to correct it. The error will result in loss of funds if the owner sets the value of claimable tokens for a recipient to a greater value than what was intended.

The `TokenDistributor` contract allows the owner to use the `setRecipients` function to set a list of recipients to receive tokens.

```
// sanity check that the address being set is consistent
    // if for some reason the owner made an error they can still set the address to
zero in order to correct it
    require(claimableTokens[_recipients[i]] == 0, "TokenDistributor: recipient
already set");
    claimableTokens[_recipients[i]] = _claimableAmount[i];
    emit CanClaim(_recipients[i], _claimableAmount[i]);
    unchecked {
        sum += _claimableAmount[i];
    }
```

Figure 7.1: Part of `setRecipient` in TokenDistributor

According to the comment, it is assumed that the owner can correct an error by setting the value of `claimableTokens` for a recipient to 0, but it is not possible to set the value of `claimableTokens[recipient]` to 0 once it has been set to a non-zero value. The `require` statement will revert if the owner tries to set the value of `claimableTokens` for a recipient that has already been set to a non-zero value.

Note that the value of `sum` is always incremented by the `_claimableAmount[i]` while making changes to this function. Therefore, if the code is modified to change the value of `claimableTokens` for a recipient, the value of `sum` should be updated accordingly. Otherwise, the contract will be left with a wrong value of `totalClaimable`.

Exploit Scenario

Alice, the owner, sets the recipients but makes a mistake; Alice tries to correct it, but the function reverts. As a result, a recipient can claim more tokens than they should.

Recommendations
Short term, allow the owner to change the configuration to correct mistakes.

Long term, thoroughly review the assumptions made during development; in this case, a comment indicates an assumption that does not hold.

8. Governance proposal that rely on retryable tickets could be disabled	
Severity: **Low**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-ARBGOV-8
Target: `governance/src/L1ArbitrumTimelock.sol`	

Description

Certain governance proposals can be temporarily blocked by the owner of the bridge, disabling the use of retryable tickets.

Part of the governance system lives in the Ethereum mainnet, and can involve the use of retryable tickets:

```
/// @dev If the target is reserved "magic" retryable ticket address
address(bytes20(bytes("retryable ticket magic")))
 /// we create a retryable ticket at provided inbox; otherwise, we execute directly
 function _execute(address target, uint256 value, bytes calldata data)
     internal
     virtual
     override
 {
     if (target == RETRYABLE_TICKET_MAGIC) {
         // if the target is reserved retryable ticket address,
         // we retrieve the inbox from the data object and
         // then we create a retryable ticket,
         (
     …
```

Figure 8.1: Code to execute proposal that require retryable tickets in `L1ArbitrumTimelock`

However, the owner of the bridge mainnet can still stop the execution of a proposal that disallows the creation of new retryable tickets:

```
function createRetryableTicketNoRefundAliasRewrite(
    address destAddr,
    uint256 l2CallValue,
    uint256 maxSubmissionCost,
    address excessFeeRefundAddress,
    address callValueRefundAddress,
    uint256 maxGas,
    uint256 gasPriceBid,
    bytes calldata data
) public payable virtual onlyWhitelisted returns (uint256) {
    require(!isCreateRetryablePaused, "CREATE_RETRYABLES_PAUSED");
    …
```
Figure 8.2: Header of `createRetryableTicketNoRefundAliasRewrite` in the Arbitrum Inbox

In the example above, the timelock will not be able to execute the proposal despite the governance decision.

Exploit Scenario
The governance decides to execute a proposal that involves creating a retryable ticket. The owner of the bridge does not want to execute it, so they temporarily disable ticket creation.

Recommendations
Short term, properly document this limitation to ensure delegates are aware of this issue.

Long term, review how privileged actors can interfere with each other during governance proposals.

9. Admin role of L2GovernanceFactory on UpgradeExecutor is not revoked

Severity: **Informational**	Difficulty: **High**
Type: Access Control	Finding ID: TOB-ARBGOV-9
Target: `governance/src/L2GovernanceFactory.sol`	

Description

The `L2GovernanceFactory` contract is left with an admin role on `UpgradeExecutor`. This role can be used to introduce backdoors into the governance system.

The Arbitrum governance is created and configured by a factory contract across several steps. In the first step, the factory contract is granted an admin role on `UpgradeExecutor` when it is initialized:

```
// we make this contract the admin of the upgrade executor for now, then
// switch that over in step 3
dc.executor.initialize(address(this), new address[](0));
```

Figure 9.1: Part of `deployStep1` function in `L2GovernanceFactory`

`Although` this role is necessary in the following step, it should be revoked in step 3 (as suggested in the above code comment):

```
function deployStep3(address[] memory _l2UpgradeExecutors) public onlyOwner {
    require(step == Step.Three, "L2GovernanceFactory: not step three");
    // now that we all the address we can grant roles to them on the upgrade
executor
    UpgradeExecutor exec = UpgradeExecutor(upExecutor);
    for (uint256 i = 0; i < _l2UpgradeExecutors.length; ++i) {
        exec.grantRole(exec.EXECUTOR_ROLE(), _l2UpgradeExecutors[i]);
    }
    exec.grantRole(exec.ADMIN_ROLE(), upExecutor);

    step = Step.Complete;
}
```

Figure 9.2: `deployStep3` function in `L2GovernanceFactory`

However, as shown in figure 9.2 above, there is no statement to revoke the admin role of the factory contract on the `UpgradeExecutor` contract.

Recommendations

Short term, add the following statement to the `deployStep3` function of the `L2GovernanceFactory` contract:

```
exec.revokeRole(exec.ADMIN_ROLE(), address(this));
```

Figure 9.3 Statement to be added to `deployStep3` in `L2GovernanceFactory`

Long term, carefully document the roles that each contract should have and review the codebase to make sure that these roles are being granted and/or revoked per the expectations.

10. Missing contract size check	
Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBGOV-10
Target: `governance/src/UpgradeExecutor.sol`	

Description

The `UpgradeExecutor` contract uses `delegatecall` to conduct upgrades. If the `upgrade` parameter is set to an address that does not contain code to execute, a `delegatecall` to the parameter will still return success. Since this contract is designed to allow the executor to execute arbitrary logic, the function should not allow calling an address that does not have any effect.

```
function execute(address upgrade, bytes memory upgradeCallData)
    public
    payable
    onlyRole(EXECUTOR_ROLE)
    nonReentrant
{
    (bool success,) = address(upgrade).delegatecall(upgradeCallData);
    require(success, "UpgradeExecutor: inner delegate call failed");
}
```

Figure 10.1: The execute function in `UpgradeExecutor`

The Solidity documentation includes the following warning:

```
The low-level functions call, delegatecall and staticcall return true as their first
return value if the account called is non-existent, as part of the design of the
EVM. Account existence must be checked prior to calling if needed.
```

Figure 10.2: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`

Recommendations

Short term, implement a contract existence check before a `delegatecall`.

Long term, carefully review the Solidity documentation, especially the “Warnings” section, and document failure cases for the upgrade functionality of the `UpgradeExecutor` contract.

11. L2ArbitrumGateway trusts L2Token to return correct l1Address

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBGOV-11
Target: `tokenbridge/arbitrum/gateway/L2ArbitrumGateway.sol`	

Description

If a malicious user controls the return value of the L1 token address provided by the L2 contract code, they can drain the tokens using the Arbitrum token bridge.

Bridge contracts do not store mappings of L2 tokens to L1 tokens, and `L2ArbitrumGateway` depends on the L2 token contract to return the correct value of the L1 token to be bridged. Additionally, t custom gateway contracts store a mapping of the L1 token to the L2 token in `mapping(address => address)`, which allows two different L1 tokens to be mapped to the same L2 token. These two issues combined can be used to steal funds from the bridge.

The `outboundTransfer` function in `L2ArbitrumGateway` accepts an L1 token address as an argument, then loads the corresponding L2 token address from the `l1ToL2Token` mapping. The function proceeds to call the `l1Address` function on the L2 token contract to get the address of the associated L1 token, comparing the returned value with the provided L1 token address to check if they match.

However, if an attacker can control the value of `l1Address` returned by L2 token, they can use this **limited control over L2 token** with a malicious L1 token contract to steal funds from the bridge.

```
address l2Token = calculateL2TokenAddress(_l1Token);
require(l2Token.isContract(), "TOKEN_NOT_DEPLOYED");
require(IArbToken(l2Token).l1Address() == _l1Token, "NOT_EXPECTED_L1_TOKEN");
```

Figure 11.1: Part of the `OutboundTransfer` function in `L2ArbitrumGateway`

The client is aware of a similar issue where the attack can be executed only by the owner of the gateway contract; in this case, the attack can be executed by anyone with limited control over the L2 token contract.

Exploit Scenario

A legitimate token that allows its L2 counterpart to update its associated L1 token address is deployed and registered in the Arbitrum bridge.

Eve, a malicious user, changes the L2 contract's associated L1 token address. She proceeds to deploy another L1 token contract that she controls and mint new tokens. She then registers the token contract in the Arbitrum bridge and associates it with the legitimate L2 token.

Eve changes the L2 contract's associated L1 address to return the newly deployed token contract triggers a deposit. The deposit is executed, and Eve proceeds to change the address associated with the L2 token back to the original, allowing her to convert the fake tokens into the legitimate ones.

Recommendations

Short term, consider:

- Document assumptions that are made when interacting with untrusted contracts and whether the bridge should include additional safeguards. If it should not, consider documenting these trust assumptions so that users interacting with the bridge are aware of them.
- Implement additional checks in `registerL2Token` to avoid registering two L1 tokens with one L2 token.
- Consider including a L2 token to L1 token mapping.

Long term, carefully review interactions with third-party code and their trust assumptions. Minimize trust in third-party contracts by implementing stricter validation.

12. Retryable tickets allow out-of-order execution of token bridge registration functions	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBGOV-12
Target: `governance/src/L2GovernanceFactory.sol`	

Description

Arbitrum's retryable tickets can be exploited to break assumptions about how tokens are registered and configured in the bridge.

The Arbitrum token bridge uses retryable tickets to propagate the information regarding a new token from Ethereum to Arbitrum in a multi-step procedure.

The process starts when the `registerTokenToL2` function is called using `_l2Address` from the token to register. This value can be either `0x0` or some other address. Once a non-zero address is used, only the same input can be used to recreate the retryable ticket:

```
function registerTokenToL2(
    address _l2Address,
    uint256 _maxGas,
    uint256 _gasPriceBid,
    uint256 _maxSubmissionCost,
    address _creditBackAddress
) public payable returns (uint256) {
    require(
        ArbitrumEnabledToken(msg.sender).isArbitrumEnabled() == uint8(0xa4b1),
        "NOT_ARB_ENABLED"
    );

    address currL2Addr = l1ToL2Token[msg.sender];
    if (currL2Addr != address(0)) {
        // if token is already set, don't allow it to set a different L2 address
        require(currL2Addr == _l2Address, "NO_UPDATE_TO_DIFFERENT_ADDR");
    }

    l1ToL2Token[msg.sender] = _l2Address;

    address[] memory l1Addresses = new address[](1);
    address[] memory l2Addresses = new address[](1);
    l1Addresses[0] = msg.sender;
    l2Addresses[0] = _l2Address;
    emit TokenSet(l1Addresses[0], l2Addresses[0]);

    bytes memory _data = abi.encodeWithSelector(
        L2CustomGateway.registerTokenFromL1.selector,
        l1Addresses,
        l2Addresses
    );

    return
        sendTxToL2(
            inbox,
            counterpartGateway,
            _creditBackAddress,
            msg.value,
            0,
            _maxSubmissionCost,
            _maxGas,
            _gasPriceBid,
            _data
        );
}
```

Figure 12.2: `registerTokenToL2` function in `L1CustomGateway`

On the Arbitrum side, the code just maps L1 and L2 addresses if the sender is correct, regardless of the validity of the data itself.

```
function registerTokenFromL1(address[] calldata l1Address, address[] calldata
l2Address) external onlyCounterpartGateway {
  // we assume both arrays are the same length, safe since its encoded by the L1
  for (uint256 i = 0; i < l1Address.length; i++) {
    // here we don't check if l2Address is a contract and instead deal with that
behaviour
    // in `handleNoContract` this way we keep the l1 and l2 address oracles in sync
    l1ToL2Token[l1Address[i]] = l2Address[i];
    emit TokenSet(l1Address[i], l2Address[i]);
  }
}
```

Figure 12.2: `registerTokenFromL1` function in `L2CustomGateway`

However, it is possible to break the assumptions in the L1 code using the fact that retryable tickets can be executed out of order. A token administration could force the token to call `registerTokenToL2` function using `0x0` as `_l2Address` but with a low gas/value amount, forcing the retryable ticket to go directly into the retryable queue. Then, the same user can force the token to again call the `registerTokenToL2` function, but using a non-zero address as `_l2Address`. This time, the retryable ticket can be immediately redeemed. However, the old ticket will still be available for execution.

Note that the same issue affects the selection of gateway using `L1GatewayRouter.setGateway`.

Exploit scenario

1. Eve deploys a new token that has been audited and has no security or correctness issues.
2. Alice buys 20 of Eve's tokens.
3. Eve registers her token into Arbitrum L2, secretly keeping an extra retryable ticket that overrides its address to zero.
4. Alice starts moving her tokens into Arbitrum using the bridge.
5. Eve triggers the extra retryable override in her token address, trapping Alice's 20 tokens.

Recommendations

Short term, consider adding extra validation on the Arbitrum side to verify that retryable tickets are not executed out of order. Keep in mind that the owner of the gateway should still be able override token addresses.

Long term, review the impact of the out-of-order execution resulting from retryable tickets across the codebase.

13. Assumption of all tokens being burned in outboundEscrowTransfer

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBGOV-13

Target:
- `tokenbridge/arbitrum/gateway/L1ReverseCustomGateway.sol`
- `tokenbridge/arbitrum/gateway/L2CustomGateway.sol`

Description
The sender balance checks performed by the `L1ReverseCustomGateway` and `L2CustomGateway` contracts' `outboundEscrowTransfer` assume that any change in balance is caused by tokens being burned.

Whenever tokens are bridged between Arbitrum and Ethereum, a mint-and-burn mechanism is used. This allows tokens to properly track their supply in the adequate chain (i.e., Ethereum in the case of "L1-native tokens" and Arbitrum in the case of "L2-native tokens" or "reverse tokens").

To support this burning mechanism, tokens that implement bridging to Arbitrum using the `L1ReverseCustomGateway` or to Ethereum using `L2CustomGateway` contract are expected to comply with a particular specification.

In the case of the L1-native tokens, their L2 counterpart has to comply with the `IArbToken` interface, and in the case of L2-native tokens, their L1 counterpart would need to comply with the `L1ReverseToken` interface.

```
interface IArbToken {
    function bridgeMint(address account, uint256 amount) external;

    function bridgeBurn(address account, uint256 amount) external;

    function l1Address() external view returns (address);
}
```

Figure 13.1: `IArbToken` interface

```
interface L1ReverseToken is L1MintableToken {
    function bridgeBurn(address account, uint256 amount) external;
}
```

Figure 13.2: L1ReverseToken interface

This bridgeBurn is used whenever a withdrawal is triggered (i.e., when L1-native tokens are bridged back to Ethereum and when L2-native tokens—reverse tokens—are bridged back to Arbitrum). However, the way the bridge determines the amount of tokens that have been burned—and thus the amount of tokens that are being withdrawn—does not offer enough security guarantees when it comes to custom token implementations.

```
function outboundEscrowTransfer(
    address _l2Token,
    address _from,
    uint256 _amount
) internal virtual override returns (uint256 amountBurnt) {
    uint256 prevBalance = IERC20(_l2Token).balanceOf(_from);

    super.outboundEscrowTransfer(_l2Token, _from, _amount);

    uint256 postBalance = IERC20(_l2Token).balanceOf(_from);
    return SafeMath.sub(prevBalance, postBalance);
}
```

Figure 13.3: outboundEscrowTransfer function in L2CustomGateway.sol

As shown above, the balance of _from is checked before and after executing the super outboundEscrowTransfer function, which will call bridgeBurn on the token contract (Figure 13.4) . These balance checks are used to calculate the amount of tokens that were burned by the user.

```
function outboundEscrowTransfer(
    address _l2Token,
    address _from,
    uint256 _amount
) internal virtual returns (uint256 amountBurnt) {
    IArbToken(_l2Token).bridgeBurn(_from, _amount);
    return _amount;
}
```

Figure 13.4: outboundEscrowTransfer function in L2CustomGateway.sol

However, if the L2 token implementation is non-standard and has on-transfer hooks, then the caller (_from) could transfer funds to a different address during the bridgeBurn. The transfer would decrease the balance of the _from address but not necessarily burn, or remove the tokens from, the supply. Thus, a user could bridge tokens, transfer them to another during the token hook, and inflate the aggregate (L1 and L2) total supply of the token.

```
function outboundTransfer(
    address _l1Token,
    address _to,
    uint256 _amount,
    uint256, /* _maxGas */
    uint256, /* _gasPriceBid */
    bytes calldata _data
) public payable override returns (bytes memory res) {
    [...]
        _amount = outboundEscrowTransfer(l2Token, _from, _amount);
    [...]
}
```

Figure 13.5: `outboundTransfer` function in `L2ArbitrumGateway.sol`

```
function outboundTransfer(
    address _l1Token,
    address _to,
    uint256 _amount,
    uint256 _maxGas,
    uint256 _gasPriceBid,
    bytes calldata _data
) public payable override returns (bytes memory res) {
    return
        outboundTransferCustomRefund(_l1Token, _to, _to, _amount, _maxGas,
_gasPriceBid, _data);
}

function outboundTransferCustomRefund(
    address _l1Token,
    address _refundTo,
    address _to,
    uint256 _amount,
    uint256 _maxGas,
    uint256 _gasPriceBid,
    bytes calldata _data
) public payable virtual override returns (bytes memory res) {
    require(isRouter(msg.sender), "NOT_FROM_ROUTER");
    [...]
    _amount = outboundEscrowTransfer(_l1Token, _from, _amount);
    [...]
}
```

Figure 13.6: `outboundTransfer` function in `L1ArbitrumGateway.sol`

Exploit scenario

A user registers a reverse token in the Arbitrum bridge that implements on-transfer hooks (e.g., ERC777 or custom ERC20) that notify sender and recipient of the transfer. Whenever the tokens are being bridged from Ethereum to Arbitrum (i.e., they are withdrawn), the `bridgeBurn` function of the token contract is called and the on-transfer hook is triggered. This results in the transfer of the execution's control flow to the sender, who is notified of the transfer.

Eve, a malicious user, notices that the `outboundEscrowTransfer` function of the `L1ReverseGateway` queries the balance of the sender before and after the call to `bridgeBurn`. Capitalizing on this fact, Eve deploys a malicious contract with a fallback function that sends part of its balance to a different address when the on-transfer hook of the `bridgeBurn` function is executed.

Eve uses the malicious contract to withdraw her funds into Arbitrum. The on-transfer hook is triggered, and a part of the contract's balance is transferred to a different address. Because the `outboundEscrowTransfer` relies on the balance of the sender to determine the amount of tokens that were burned, the bridge registers an inflated withdrawal. Although a part of Eve's tokens was burned, a bigger part was transferred out of her address into a different one she controls.

This allows Eve to withdraw more tokens than she should be able to and therefore to steal tokens from the bridge.

For a concrete example:

- Eve has a balance of 100 tokens.
- Eve triggers a withdrawal of 25 tokens through the malicious contract.
- The on-transfer hook transfers 75 tokens from Eve's contract address to another address she controls.
- The `bridgeBurn` function burns 25 tokens.
- 100 tokens are registered as "withdrawn" because Eve's balance decreased by 100.

Recommendations

Short term, consider one of the following:

- Check that `totalSupply` decreases by the same amount that the user burns following the `bridgeBurn` call.
- Transfer tokens from the caller to the gateway contract and track the gateway contract's change in balance during transfer and burning.

Long term, carefully review interactions with third-party code and their trust assumptions. Minimize trust in third-party contracts by implementing stricter validation.

14. Dead code in outboundTransferCustomRefund

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-ARBGOV-14
Target: `tokenbridge/ethereum/gateway/L1ArbitrumGateway.sol`	

Description

Tokens can register their official gateways with the `L1ArbitrumRouter`. The gateway logic no longer allows arbitrary callers, and instead allows only `router` contracts. However, the codebase still contains remnant dead code that supports arbitrary callers:

```
function outboundTransferCustomRefund(
    address _l1Token,
    address _refundTo,
    address _to,
    uint256 _amount,
    uint256 _maxGas,
    uint256 _gasPriceBid,
    bytes calldata _data
) public payable virtual override returns (bytes memory res) {
    require(isRouter(msg.sender), "NOT_FROM_ROUTER");
    // This function is set as public and virtual so that subclasses can override
    // it and add custom validation for callers (ie only whitelisted users)
    address _from;
    uint256 seqNum;
    bytes memory extraData;
    {
        uint256 _maxSubmissionCost;
        if (super.isRouter(msg.sender)) {
            // router encoded
            (_from, extraData) =
GatewayMessageHandler.parseFromRouterToGateway(_data);
        } else {
            _from = msg.sender;
            extraData = _data;
        }
```

Figure 14.1: The `outboundTransferCustomRefund` function in `L1ArbitrumGateway.sol`

Note that this code is also present in the L2 gateway contracts; however, those do not have the `require` statement that restricts the caller to the router.

Recommendations

Short term, if the function should be callable only by the router, consider removing the `else` statement. Otherwise, clarify the intended access controls for the `outboundTransferCustomRefund` function.

Long term, remove unused code from a codebase. This will help keep the contracts cleaner and more readable.

15. Lack of contract existence checks in the gateway may not detect failed execution

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBGOV-15
Target: `tokenbridge/ethereum/gateway/L1ERC20Gateway.sol`	

Description

The `L1ERC20Gateway` contract uses `staticcall` without checking contract existence. This may lead to tokens getting stuck on the L1ERC20Gateway.

For a user to bridge a token through the L1ERC20Gateway, the token's name, symbol, and number of decimals must be encoded on layer 1. This process uses the `callStatic` function to retrieve the results of an external call:

```
/**
 * @notice utility function used to perform external read-only calls.
 * @dev the result is returned even if the call failed or was directed at an EOA,
 * it is cheaper to have the L2 consumer identify and deal with this.
 * @return result bytes, even if the call failed.
 */
function callStatic(address targetContract, bytes4 targetFunction)
    internal
    view
    returns (bytes memory)
{
    (
        ,
        /* bool success */
        bytes memory res
    ) = targetContract.staticcall(abi.encodeWithSelector(targetFunction));
    return res;
}
```

Figure 15.1: The `callStatic` function in `L1ERC20Gateway.sol`

This data is sent verbatim to layer 2, which decodes the token's name, symbol, and number of decimals before the bridge is initialized. The parsing for these return values assumes the name and symbol are strings and that the decimals are uint8. If any of the types or parsing fails, tokens will not be redeemable and be stuck on the L1ERC20Gateway.

```
/**
```

```
 * @notice initialize the token
 * @dev the L2 bridge assumes this does not fail or revert
 * @param _l1Address L1 address of ERC20
 * @param _data encoded symbol/name/decimal data for initial deploy
 */
function bridgeInit(address _l1Address, bytes memory _data) public virtual {
    (bytes memory name_, bytes memory symbol_, bytes memory decimals_) = abi.decode(
        _data,
        (bytes, bytes, bytes)
    );
    // what if decode reverts? shouldn't as this is encoded by L1 contract

    /*
     *  if parsing fails, the type's default value gets assigned
     *  the parsing can fail for different reasons:
     *      1. method not available in L1 (empty input)
     *      2. data type is encoded differently in the L1 (trying to abi decode the
wrong data type)
     *  currently (1) returns a parser fails and (2) reverts as there is no
`abi.tryDecode`
     *  https://github.com/ethereum/solidity/issues/10381
     */

    (bool parseNameSuccess, string memory parsedName) = BytesParser.toString(name_);
    (bool parseSymbolSuccess, string memory parsedSymbol) =
BytesParser.toString(symbol_);
    (bool parseDecimalSuccess, uint8 parsedDecimals) =
BytesParser.toUint8(decimals_);
```

Figure 15.2: The `bridgeInit` function in `StandardArbERC20.sol`

Recommendations

Short term, either:

- Add a contract existence check if the function is expected to be used only for tokens and not for externally owned accounts.
- Document the location of the associated L2 consumer identification check and/or what the L2 check entails. Interconnected calls should be explicitly documented to ensure correct assumptions are being made across systems.

Long term, document assumptions that are made between L1 and L2 and where checks should occur. Explicit data validation across different platforms and languages is especially important for a codebase of this level of complexity.

16. Cross-chain message out-of-order execution impacts sequential proposal execution

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARBGOV-16
Target: `governance/src/L1ArbitrumTimelock.sol`, `governance/src/ArbitrumTimelock.sol`	

Description

Out-of-order execution of outbox transactions on L1 and retryable tickets on L2 can lead to unexpected results when governance proposals rely on specific ordering of execution of actions.

Part of the governance system lives in Ethereum mainnet and can involve the use of retryable tickets:

```
/// @dev If the target is reserved "magic" retryable ticket address
address(bytes20(bytes("retryable ticket magic")))
 /// we create a retryable ticket at provided inbox; otherwise, we execute directly
 function _execute(address target, uint256 value, bytes calldata data)
     internal
     virtual
     override
 {
     if (target == RETRYABLE_TICKET_MAGIC) {
         // if the target is reserved retryable ticket address,
         // we retrieve the inbox from the data object and
         // then we create a retryable ticket,
         (
     …
```

Figure 16.1: Code to execute proposal that require retryable tickets in `L1ArbitrumTimelock`

At the same time, passed proposals can be executed with a certain order either in a batch or using the `predecessor` field:

```
/// @inheritdoc TimelockControllerUpgradeable
/// @dev Adds the restriction that only the counterparty timelock can call this func
function scheduleBatch(
    address[] calldata targets,
    uint256[] calldata values,
    bytes[] calldata payloads,
    bytes32 predecessor,
    bytes32 salt,
    uint256 delay
) public virtual override (TimelockControllerUpgradeable) onlyCounterpartTimelock {
    TimelockControllerUpgradeable.scheduleBatch(
        targets, values, payloads, predecessor, salt, delay
    );
}

/// @inheritdoc TimelockControllerUpgradeable
/// @dev Adds the restriction that only the counterparty timelock can call this func
function schedule(
    address target,
    uint256 value,
    bytes calldata data,
    bytes32 predecessor,
    bytes32 salt,
    uint256 delay
) public virtual override (TimelockControllerUpgradeable) onlyCounterpartTimelock {
    TimelockControllerUpgradeable.schedule(target, value, data, predecessor, salt,
delay);
}
```

Figure 16.2: `scheduleBatch` and `schedule` functions in `L1ArbitrumTimelock.sol`

However, a malicious user can leverage out-of-order execution of retryable tickets to break the assumptions of one proposal's execution following another proposal's execution.

Exploit Scenario

Governance votes the execution of two proposals: A and B, where A is a prerequisite of B. The first proposal produces a retryable ticket, but the execution of this ticket fails for some reason and needs to be manually redeemed. The L1Timelock contract registered the execution of A as successful and allows the execution of B, which can cause failed upgrades or some other unforeseen consequences.

Recommendations

Short term, consider following changes for both L1 and L2 timelock contracts:

1. Override the `schedule` and `scheduleBatch` functions of timelock to not accept the `predecessor` argument.
2. Override the `scheduleBatch` to accept only one action in the array.

Long term, carefully read through the imported library code to understand the design decisions and implement customizations to suit your requirements and limitations.

17. Retryable tickets used in governance proposals can be silently discarded

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBGOV-17
Target: `governance/src/L1ArbitrumTimelock.sol`	

Description

Inconsistency in API and `ArbOS` retryable ticket parser validation in the `Inbox` contract can cause the `ArbOS` to silently discard a retryable ticket.

Part of the governance system lives in Ethereum mainnet, and can involve the use of retryable tickets:

```
/// @dev If the target is reserved "magic" retryable ticket address
address(bytes20(bytes("retryable ticket magic")))
 /// we create a retryable ticket at provided inbox; otherwise, we execute directly
 function _execute(address target, uint256 value, bytes calldata data)
     internal
     virtual
     override
 {
     if (target == RETRYABLE_TICKET_MAGIC) {
         // if the target is reserved retryable ticket address,
         // we retrieve the inbox from the data object and
         // then we create a retryable ticket,
         (
     [...]
```

Figure 16.1: Code to execute proposal that require retryable tickets in `L1ArbitrumTimelock`

The bridge verifies the execution of the retryable tickets. In particular, the `Inbox` contract checks that the ticket has enough value for the submission to succeed:

```
function createRetryableTicket(
    address to,
    uint256 l2CallValue,
    uint256 maxSubmissionCost,
    address excessFeeRefundAddress,
    address callValueRefundAddress,
    uint256 gasLimit,
    uint256 maxFeePerGas,
    bytes calldata data
) external payable whenNotPaused onlyAllowed returns (uint256) {
    // ensure the user's deposit alone will make submission succeed
    ..
    if (
        msg.value < (maxSubmissionCost + l2CallValue + gasLimit * maxFeePerGas) &&
        msg.sender != UNSAFE_CREATERETRYABLETICKET_CALLER
    ) {
        revert InsufficientValue(
            maxSubmissionCost + l2CallValue + gasLimit * maxFeePerGas,
            msg.value
        );
    }
```

Figure 16.2: `createRetryableTicket` in Arbitrum Inbox contract

However, these checks are not enough, since the `ArbOS` will discard a retryable ticket with a gas limit larger than 2**64 - 1. A user submitting a ticket using `maxFeePerGas` equal to zero can bypass the `msg.value` check. In that case, `ArbOS` will silently discard the retryable message later, during the parsing:

```
func parseSubmitRetryableMessage(rd io.Reader, header *L1IncomingMessageHeader,
chainId *big.Int) (*types.Transaction, error) {
	...
	gasLimitBig := gasLimit.Big()
	if !gasLimitBig.IsUint64() {
		return nil, errors.New("gas limit too large")
	}
	...
```

Figure 16.2: Part of the `parseSubmitRetryableMessage` function in `incomingmessage.go`

Exploit Scenario

Governance votes the execution of proposal A, which creates a retryable ticket. However, the execution of this ticket fails because the parameters of gas limit and max fee per gas are incorrect. Instead of being allowed to re-execute it, the ticket is never created and the proposal should be voted and executed again.

Recommendations

Short term, whenever possible, validation should be performed as early as possible (e.g., on-chain) to prevent instances in which invalid data is deemed valid on-chain but invalid off-chain, which can create misleading expectations to users.

Long term, thoroughly document the assumptions regarding ticket creation and how these are processed by off-chain components to prevent similar issues.

18. Lack of contract existence checks is error-prone when scheduling transactions through the timelock

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBGOV-18
Target: `governance/src/L1ArbitrumTimelock.sol`	

Description

A lack of contract existence check will cause the contract to consider proposals as successfully executed even when their intended side effects have not been triggered.

Arbitrum's governance system uses a timelock contract through which proposals are submitted. Through this contract, proposals are first scheduled and, after a period of time, executed. When a proposal is ready to be executed, any address with the `EXECUTOR_ROLE` can call the `execute` function to trigger the proposal's execution.

```
function execute(
    address target,
    uint256 value,
    bytes calldata payload,
    bytes32 predecessor,
    bytes32 salt
) public payable virtual onlyRoleOrOpenRole(EXECUTOR_ROLE) {
    bytes32 id = hashOperation(target, value, payload, predecessor, salt);

    _beforeCall(id, predecessor);
    _execute(target, value, payload);
    emit CallExecuted(id, 0, target, value, payload);
    _afterCall(id);
}
```

Figure 18.1: `execute` function in OpenZeppelin's `TimelockControllerUpgradeable`

Arbitrum's timelock implementation overrides OpenZeppelin's by including additional functionality to handle retryable tickets in the `_execute` function; however, OpenZeppelin's original code is still relied upon for other use-cases.

```
function _execute(address target, uint256 value, bytes calldata data)
    internal
    virtual
    override
{
    if (target == RETRYABLE_TICKET_MAGIC) {
      [...]
    else {
        // Not a retryable ticket, so we simply execute
        super._execute(target, value, data);
    }
 }
```

Figure 18.2: _execute function in L1ArbitrumTimelock

```
function _execute(
    address target,
    uint256 value,
    bytes calldata data
) internal virtual {
    (bool success, ) = target.call{value: value}(data);
    require(success, "TimelockController: underlying transaction reverted");
}
```

Figure 18.3: _execute function in OpenZeppelin's TimelockControllerUpgradeable.sol

In this instance, OpenZeppelin's implementation uses a low-level call, which will always succeed if `target` is not a contract. Because there is no way to indicate whether `target` is supposed to be a contract or not, any proposal that is supposed to target a contract, but that contract has not been deployed or has been destroyed, will be deemed "successfully executed" even though its intended side effects have not been triggered.

Exploit scenario

Governance votes, approves, and schedules a proposal to deploy a new contract and then execute some functionality implemented by it.

A mistake is made during the proposal execution, and the transaction that would deploy the new contract is performed after the transaction that calls the contract. Because no contract existence check is made, the proposal is deemed successful even though some of the intended side effects (i.e., calling the newly deployed contract to execute the functionality) have not been triggered. A new proposal should be created and voted on, and the expected delay should be enforced.

Recommendations

Short term, consider including a contract existence check when `data` is not empty; however, note that this will prevent sending ETH and non-empty `data` to EOAs, which may be a use-case that the Offchain Labs team would like to support. Use an retryable ticket special address that contains more letters, which will enable the compiler to successfully verify it.

Long term, thoroughly document the assumptions about proposal correctness when scheduling and executing them through the timelock.

19. Potential overflow in TokenDistributor causes imprecise claims

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARBGOV-19
Target: `governance/src/TokenDistributor.sol`	

Description
In the `setRecipients` function, the unchecked block allows `sum` to overflow. If `sum` overflows, the balance check may succeed and result in users having a claim to more tokens than are available in the contract. In practice, the claimable amounts should add up to the token supply, which will not exceed the maximum 256-bit, unsigned integer.

```
{
[...]
    unchecked {
        sum += _claimableAmount[i];
    }
}

// sanity check that the current has been sufficiently allocated
require(token.balanceOf(address(this)) >= sum, "TokenDistributor: not enough
balance");
```

Figure 19.1: The `setRecipients` *function in* `TokenDistributor.sol`

Exploit Scenario
Alice, the owner of the `TokenDistributor` contract, accidentally sets a very high claimable amount (e.g., the maximum 256-bit unsigned integer) for one of the recipients. As a result, the recipient is able to claim all available tokens, leaving the other recipients with nothing.

Recommendations
Short term, remove the unchecked block around the calculation of `sum`.

Long term, ensure that validations will not permit exceptional behavior such as overflows.

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. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

TokenDistributor

- **Consider allowing other addresses to claim tokens on behalf of the token owners.** The `claim` function allows only the sender to claim their own tokens, but if the owner is a smart contract, a call to that contract could be impossible.
- **Document the fact that the sweep function can be called multiple times.** The self-destruction of an Ethereum contract occurs only at the end of a top-level transaction, so the sweep function can be called multiple times, emitting more than one `Swept` event in a single transaction. Off-chain components should not assume a single `Swept` event on a transaction.

L1ArbitrumGateway and L1CustomGateway

- **Consider explicitly overriding the `outboundTransfer`** function of the `L1ArbitrumGateway` into `L1CustomGateway` to apply the `nonReentrant` modifier. The `outboundTransfer` function in `L1ArbitrumGateway` calls `outboundTransferCustomRefund`, which is overridden in `L1CustomGateway` with the `nonReentrant` modifier, making it secure against reentrancy. However, if some code is later added in the `outboundTransfer` function before the call to `outboundTransferCustomRefund`, that new code will not be secure against reentrancy.

L1GatewayRouter

- **ERC20 tokens allowing users to make external calls also allow users to set malicious gateways.** The `L1GatewayRouter` contract allows tokens to call the `setGateway` function, which registers the provided gateway with the `msg.sender`. If a token allows users to make an external call, then users can exploit it to register a malicious gateway while it is not already set.

TestUpgrade

- The `upgradeWithValue` function checks that the balance is exactly equal to the provided `value` argument, which may fail if used in production.

L1ArbitrumTimelock

- **`_execute` ignores the provided `value` argument while creating a retryable ticket**. This issue may lead to unexpected behavior if the proposer does not pay attention while creating the proposal. It should be documented that the `value` parameter is encoded in the `data` argument for L1 timelock instead of the `value` argument for L1 timelock. Additionally, consider correcting the incomplete comment that mentions `value` being ignored.