Arbitrum BoLD
Findings & Analysis Report

2024-06-17

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Arbitrum BoLD smart contract system written in Solidity. The audit took place between May 10—May 27 2024.

Wardens

31 Wardens contributed reports to Arbitrum BoLD:

1. xuwinnie
2. Ch_301
3. 0x73696d616f
4. SpicyMeatball
5. Sathish9098
6. Kow
7. Emmanuel
8. ladboy233
9. Rhaydden
10. dontonka
11. josephdara
12. bronze_pickaxe
13. K42
14. fyamf
15. slvDev
16. hihen
17. ZanyBonzy
18. forgebyola
19. Dup1337 (ChaseTheLight, sorrynotsorry, and deliriusz)

20. Takarez
21. twcctop
22. Audinarey
23. guhu95
24. zanderbyte
25. carlitox477
26. LessDupes (3docSec, sin1st3r__, and EV_om)
27. KupiaSec

This audit was judged by Picodes.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 4 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 2 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 23 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Arbitrum BoLD repository, and is composed of 14 interfaces and 27 logic contracts written in the Solidity programming language and includes 3,603 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (2)

[H-01] Adversary can make honest parties unable to retrieve their assertion stakes if the required amount is decreased

Submitted by xuwinnie*, also found by* Ch_301

Impact

When the required stake (to create a new assertions) is updated to a lower amount, adversary can make the honest party unable to retrieve their assertion stakes.

Proof of Concept

```
A -- B -- C -- D(latest confirmed) -- E
```

Suppose the initial stake amount is 1000 ETH, and till now no invalid assertions have been made. (A, B, C, D, E are all valid and made by the same validator). The rollup contract should hold 1000 ETH now.

```
A -- B -- C -- D(latest confirmed) -- E
                                     \
                                      \ F(invalid)
```

Then, the admin update the required stake to 700 ETH, Alice made an invalid assertion F. Since its parent D was created before the update, Alice will still need to stake 1000 ETH, and the 1000 ETH will be sent to loserStakeEscrow.

```
if (!getAssertionStorage(newAssertionHash).isF

    // only 1 of the children can be confirmed
    // so we send the other children's stake t
    IERC20(stakeToken).safeTransfer(loserStakeE
}
```

```
A -- B -- C -- D(latest confirmed) -- E
                                     \

                                      \ F -- G
```

(a) Alice creates F's children, G. Now, only 700 ETH of stake is needed. However, as the comment suggests, no refund will be made since G's ancestor could need more stake.

```
        // requiredStake is user supplied, will be ver
        // the prev's requiredStake is used to make su
        // the staker may have more than enough stake,
        // we cannot do a refund here because the stak
        // excess stake can be removed by calling redu
        require(amountStaked(msg.sender) >= assertion.
```

(b) To bypass the limit in (a), Alice calls her friend Bob to make the assertion G instead , Bob will only need to stake 700 ETH now. The rollup contract currently holds 1700 ETH. Then, Alice can withdraw her stake since she is no longer active. (her last staked assertion have a child)

```
    function requireInactiveStaker(address stakerAddres
        require(isStaked(stakerAddress), "NOT_STAKED");
        // A staker is inactive if
        // a) their last staked assertion is the latest
        // b) their last staked assertion have a child
        bytes32 lastestAssertion = latestStakedAssertio
        bool isLatestConfirmed = lastestAssertion == la
        bool haveChild = getAssertionStorage(lastestAss
        require(isLatestConfirmed || haveChild, "STAKE_
    }
```

Now the rollup contract holds 700 ETH, which means it is insolvent. The honest validator cannot withdraw her original stake. (700 < 1000)

Recommended Mitigation Steps

Ensure the following

1. A staker is considered inactive only if her last staked assertion is confirmed.

2. A staker can only stake on her last staked assertion's descendants. (otherwise Alice can switch to the correct branch and withdraw)

gzeoneth (Arbitrum) confirmed and commented:

> Patched with https://github.com/OffchainLabs/bold/pull/655.

[H-02] Edge from dishonest challenge edge tree can inherit timer from honest tree allowing confirmation of incorrect assertion

Submitted by **Kow**, *also found by* **Emmanuel**, **xuwinnie**, *and* **SpicyMeatball**

Impact

Timers can be inherited across different challenge trees and consequently incorrect assertions can be confirmed.

Proof of Concept

The function `RollupUserLogic::updateTimerCacheByClaim` allows inheritance of timers between different levels of a challenge. It performs some validation on edge being inherited from in `checkClaimIdLink` (the claiming edge).
https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/6f861c85b281a29f04daacfe17a2099d7dad5f8f/src/challengeV2/libraries/EdgeChallengeManagerLib.sol#L689-L710

```
function checkClaimIdLink(EdgeStore storage store, byte
    private
    view
{
    // the origin id of an edge should be the mutual id
    if (store.edges[edgeId].mutualId() != store.edges[
```

```
            revert OriginIdMutualIdMismatch(store.edges[edg
        }
        // the claiming edge must be exactly one level belo
        if (nextEdgeLevel(store.edges[edgeId].level, numBig
            revert EdgeLevelInvalid(
                edgeId,
                claimingEdgeId,
                nextEdgeLevel(store.edges[edgeId].level, nu
                store.edges[claimingEdgeId].level
            );
        }
    }
```

As per the comments, the claiming edge must be exactly one level below (ie. in the subchallenge directly after the inheriting edge) and its `originId` must match the `mutualId` of the inheriting edge. For clarification, we note that the inheriting edge must be a leaf edge in a challenge/subchallenge tree since the root edges of subchallenges (the layer zero edges) have `originId` derived from the `mutualId` of one of these leaf edges, and this `originId` is inherited by all its children which result from bisection.

Note that rival edges share the same `mutualId` by definition and since there isn't any extra validation, if a specific edge is a valid inheriting edge, all rivals will also be valid inheriting edges. This means rivals belonging to dishonest challenge edge trees will also be able to inherit from the timer of edges in the honest tree. Consequently, if an honest edge accumulates sufficient unrivalled time for confirmation, a malicious actor can frontrun the confirmation of the honest challenge tree to confirm the dishonest challenge, and in turn an incorrect assertion.

It is sufficient for only one dishonest child edge to inherit a sufficient timer via claim since the other will be unrivalled as challenges between two assertions can only follow one unique bisection path in each challenge tree. The only way to deny this would be to create another assertion that can be bisected to rival

the other child to halt the timer accumulation, but this would require loss of the assertion and challenge stake (since only one rival assertion and challenge edge can be confirmed). The timer can then be propogated upwards by children until we reach the root challenge edge to allow confirmation.

Even if confirmation of the dishonest root challenge edge is prevented by admin action, confirmation of the layer zero edges of subchallenges would ensure honest validators lose the stake submitted for creating a rival edge (since only one rival edge can be confirmed) and the dishonest validator(s) regain their stake.

Proof of Concept

Recommended Mitigation Steps

Allow child edges (from bisection) to inherit the `claimId` of their parent and check that the `claimId` of the claiming edge matches the `edgeId` of the inheriting edge (this would require changes to `isLayerZeroEdge`).

godzillaba (Arbitrum) confirmed

gzeoneth (Arbitrum) commented:

> Good catch.

> Fixed in https://github.com/OffchainLabs/bold/pull/659.

Medium Risk Findings (2)

[M-01] Inconsistent sequencer unexpected delay in DelayBuffer may harm users calling `forceInclusion()`

Submitted by 0x73696d616f

https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/main/src/bridge/DelayBuffer.sol#L43
https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/main/src/bridge/DelayBuffer.sol#L90-L98
https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/main/src/bridge/SequencerInbox.sol#L287

Impact

Buffer unexpected delay due to sequencer outage is inconsistent.

Proof of Concept

When the sequencer is down, users may call `SequencerInbox::forceInclusion()` to get their message added to the inbox `sequencerInboxAccs`. However, there is an incosistency when the sequencer has been down and there is more than 1 message, or even if just 1 message.

The `DelayBuffer` is used to dynamically adjust how much delay a user has to wait to call `SequencerInbox::forceInclusion()`. The buffer increase mechanism is not relevant for this issue.

The buffer decrease consists of subtracting the last time the buffer was updated, `self.prevSequencedBlockNumber`, by the previous block number of the last delay buffer update, `self.prevBlockNumber`. This is done this way to ensure that the sequencer delay can not be double counted, as the delayed inbox may have more than 1 delayed message.

However, the approach taken as a way of protecting the sequencer and not depleting the buffer incorrectly as the drawback that it also means that the buffer will not always be decreased.

For example, if the sequencer is working at a block number A, a message is submited at block number A + 10 and another one at block number A + 110. If the sequencer is down, the user has to wait, for example, delay blocks of 100 (or the delay buffer, depending on the **smallest**). If 200 blocks have passed since A + 10 (the first message), both delayed messages may be force included, at block number A + 210.

The discrepancy is that, depending on how the delayed messages are included, the buffer delay will be reduced differently. If both messages are removed at once, by calling `SequencerInbox::forceInclusion()` with the id of the newest message, the buffer delay will not be decreased, as `self.prevBlockNumber` is A, the same as `self.prevSequencedBlockNumber`. This would harm users that want to force included messages as the buffer would be bigger and they would have to wait more time for their message to be force included.

If the oldest message is first force included, the buffer delay will not decreased, as above, but `self.prevSequencedBlockNumber` will be updated to A + 210 and `self.prevBlockNumber` to A + 10. Then, if the second message is force included, `self.prevSequencedBlockNumber - self.prevBlockNumber == A + 210 - (A + 10) == 200`, which means that the buffer would correctly decrease as the sequencer was offline (ignoring the fact that there is a threshold, but the issue is the same as long as the delay is bigger than the threshold).

As a proof of concept, add the following test to `SequencerInbox.t.sol`. It shows that given 2 messages, the delay buffer is only decreased if the first message is forcely included first and only after is the newest message included. If the given `delayedMessagesRead` is the if of the second message, without forcely including the first one first, the buffer will not decrease.

```
    function test_POC_InconsistentBuffer_Decrease() public {
        BufferConfig memory configBufferable = BufferConfig({
```

```
        threshold: 600, //60 * 60 * 2 / 12
        max: 14400, //24 * 60 * 60 / 12 * 2
        replenishRateInBasis: 714
    });

    (SequencerInbox seqInbox, Bridge bridge) = deployRollu
    address delayedInboxSender = address(140);
    uint8 delayedInboxKind = 3;
    bytes32 messageDataHash = RAND.Bytes32();

    (uint64 bufferBlocks, ,,,,) = seqInbox.buffer();
    assertEq(bufferBlocks, 14400);

    vm.startPrank(dummyInbox);
    bridge.enqueueDelayedMessage(delayedInboxKind, delayed
    vm.roll(block.number + 1100);
    bridge.enqueueDelayedMessage(delayedInboxKind, delayed
    vm.stopPrank();

    vm.roll(block.number + 500);

    uint256 delayedMessagesRead = bridge.delayedMessageCou

    // buffer is not decreased if the first and second mes
    seqInbox.forceInclusion(
            delayedMessagesRead,
            delayedInboxKind,
            [uint64(block.number - 500), uint64(block.time
            0,
            delayedInboxSender,
            messageDataHash
        );
    (bufferBlocks, ,,,,) = seqInbox.buffer();
    assertEq(bufferBlocks, 14400);

    // buffer is only decreased if the first message is in
    /*seqInbox.forceInclusion(
            delayedMessagesRead - 6,
            delayedInboxKind,
            [uint64(block.number - 1600), uint64(block.tim
```

```
                0,
                delayedInboxSender,
                messageDataHash
        );
        (bufferBlocks, ,,,,) = seqInbox.buffer();
        assertEq(bufferBlocks, 14400);

        seqInbox.forceInclusion(
                delayedMessagesRead,
                delayedInboxKind,
                [uint64(block.number - 500), uint64(block.times
                0,
                delayedInboxSender,
                messageDataHash
            );
        (bufferBlocks, ,,,,) = seqInbox.buffer();
        assertEq(bufferBlocks, 13478);*/
    }
```

Tools Used

- Vscode
- Foundry

Recommended Mitigation Steps

A mitigation must be carefully taken as not to introduce double accounting of the buffer delay. One solution that would fix this issue is tracking the total unexpected delay separately and making it equal to the `block.number` minus the maximum between the previous sequenced block number and the oldest delayed message that was not yet included. This way, by doing the maximum with the last sequenced, we guarantee that no double accounting of delays takes place. By doing the maximum with the oldest delayed message, we guarantee that the delay is real and not that no message was submitted.

Note: the following discussion has been condensed for this report. To view the full discussion, please see the **original submission**.

gzeoneth (Arbitrum) disputed and commented:

> The delay buffer is an intermediary goal and not a final goal. The purpose of the delay buffer is to provide force inclusion assurances. The delay buffer updates are staggered and the buffer updates proactively in the force inclusion method (https://github.com/OffchainLabs/bold/blob/32eaf85e8ed45d069eb77e299b71fd6f3924bf40/contracts/src/bridge/SequencerInbox.sol#L309). The behavior described is not unexpected and does not have impact on force inclusion.

Picodes (judge) decreased severity to Low/Non-Critical and commented:

> This report shows how the delay buffer update could be nondeterministic and depend on how messages are included, but as shown by the sponsor fails to demonstrate why this would be important and how this fulfills the definition of Medium severity (function of the protocol or its availability could be impacted).

Picodes (judge) increased severity to Medium and commented:

> After further discussion, this issue is to me somewhere between “function incorrect as to spec, state handling”, and “availability issue”.

> Considering:

- that the impact of the end-user not being able to force-include a message as soon as possible is that its funds may be locked for some time.
- that if the sequencer is down, there may be multiple messages to force-include, and that the depletion is advertised in various places but is non-

deterministic and could be “manipulated” to delay the following message, I’ll upgrade to Medium.

godzillaba (Arbitrum) commented:

> that the impact of the end-user not being able to force-include a message as soon as possible is that its funds may be locked for some time

> @Picodes - I’m not sure this is true. In the example @0x73696d616f gave with 3 messages (m1,m2,m3 at T1,T2,T3), m1 not being included right away does not cause funds to be locked since m1 is included when m2 is included at T2, and m1 *could have* been included at any time between T1 and T2.

Picodes (judge) commented:

> @godzillaba - I meant locked for some time. Funds can’t be withdrawn as soon as they should because the buffer isn’t depleting as fast as it should.

godzillaba (Arbitrum) commented:

> If the same user (eg someone playing in an L3’s bold game) submits m2 and m3, and they force include only their own messages promptly (as in the example where there is a claimed bug) they are not affected. That user, the force includer, still can only experience some max amount of sequencer censorship over a given time period.

gzeoneth (Arbitrum) commented:

> Arbitrum’s protocol has always been optimistic, we build most of our logic under the assumption of a honest participant exists. e.g. https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/main/README.md

> It is assumed ... honest validators can react to malicious action broadcasted on-chain immediately in the same block unless the malicious party spend

> their censorship budget.
> This is unless the function is designed to be non-admin permissioned, then we enforce straighter guarantee. e.g.

> There is a correct fix that is applied in the `delayProofImpl()` but not in `forceInclusion()`.

> Batch poster's `delayProofImpl` have the "correct fix" where permissionless `forceInclusion` do not. This is by design to limit the complexity of the contracts.

> If anyone is concerned about the total delay of delayed message, it is assumed they SHOULD force include as soon as possible. If they do not call force include, there are no protocol guarantee their message will ever be included. Any delay in calling forceInclusion on parent chain can be modeled as part of the censorship budget of an attacker.

Picodes (judge) commented:

> @0x73696d616f - Can you please edit your PoC to show a force-inclusion reverting in one scenario because of the delay whereas in the other it works.

0x73696d616f (warden) commented:

> Here is a POC showing how the buffer is correctly depleted only if messages are included sequentially. The buffer gets depleted due to actively including messages sequentially. If the second message in the loop is included directly instead, the buffer is not correctly depleted, and instead of ending up being the minimum, is a much larger value (600 vs 7320). Thus, users have to wait 2000 blocks instead of 600. It can be confirmed that initially they have to wait 2000 blocks, but in the end only 600, if we include sequentially and fix this bug. If we don't include sequentially, they still have to wait 2000 blocks.

> We can play around with the numbers and observe different outcomes, but this is a real, proven risk. Here users have to wait 2000/600 = 3.3 times more to get their transaction force included. This issue will happen, how it happens depends on the conditions. The buffer being depleted is an expected scenario and the code is built to handle this situation, so arguments based on the fact that this will never happen do not stand. The impact depends on `delayBlocks`, `threshold`, `max`, `replenishRate` and user behaviour. We can find a lot of combinations showing very strong impact, as well as some showing less impact, but we know for sure this is a real risk. A list of parameters or user behavior that increase/decrease the impact can be made, but it does not erase the fact that this risk exists. It fits exactly in the definition of a medium severity issue:

> with a hypothetical attack path with stated assumptions, but external requirements.

```
function test_POC_InconsistentBuffer_Decrease() public {
    bool fix = false;
    maxTimeVariation.delayBlocks = 2000;
    BufferConfig memory configBufferable = BufferConfig({
        threshold: 600, //60 * 60 * 2 / 12
        max: 14400, //24 * 60 * 60 / 12 * 2
        replenishRateInBasis: 714
    });

    (SequencerInbox seqInbox, Bridge bridge) = deployRollu
    address delayedInboxSender = address(140);
    uint8 delayedInboxKind = 3;
    bytes32 messageDataHash = RAND.Bytes32();

    for (uint i = 0; i < 7; i++) {
        vm.startPrank(dummyInbox);
        bridge.enqueueDelayedMessage(delayedInboxKind, dela
        vm.roll(block.number + 1100);
        bridge.enqueueDelayedMessage(delayedInboxKind, dela
        vm.stopPrank();
```

```
            vm.roll(block.number + 2001);
            uint256 delayedMessagesRead = bridge.delayedMessage
            if (fix) {
                seqInbox.forceInclusion(
                        delayedMessagesRead - 1,
                        delayedInboxKind,
                        [uint64(block.number - 3101), uint64(b
                        0,
                        delayedInboxSender,
                        messageDataHash
                );
            }
            seqInbox.forceInclusion(
                    delayedMessagesRead,
                    delayedInboxKind,
                    [uint64(block.number - 2001), uint64(block.
                    0,
                    delayedInboxSender,
                    messageDataHash
            );
        }
        (uint256 bufferBlocks, ,,,,) = seqInbox.buffer();
        assertEq(bufferBlocks, fix ? 600 : 7320);

        vm.startPrank(dummyInbox);
        bridge.enqueueDelayedMessage(delayedInboxKind, delayedI
        vm.stopPrank();
        vm.roll(block.number + 601);
        uint256 delayedMessagesRead = bridge.delayedMessageCoun

        if (!fix) vm.expectRevert(ForceIncludeBlockTooSoon.sele
        seqInbox.forceInclusion(
                delayedMessagesRead,
                delayedInboxKind,
                [uint64(block.number - 601), uint64(block.timest
                0,
                delayedInboxSender,
                messageDataHash
        );
    }
```

Picodes (judge) commented:

> With the above PoC in a longer sequence the effect on the `sequenced - start` gets indeed neutralized and we're back with a DoS issue. This is Medium severity.

[M-02] `BOLDUpgradeAction.sol` will fail to upgrade contracts due to error in the `perform` function

Submitted by SpicyMeatball*, also found by* dontonka *and* josephdara

Impact

An error in the `BOLDUpgradeAction.sol` contract prevents it from upgrading and deploying new BOLD contracts.

Proof of Concept

The `perform` function serves as the entry point in the `BOLDUpgradeAction.sol` and is responsible for migrating stakers from the old rollup and deploying the challenge manager with a new rollup contract.

One of the first subroutines in this function is the `cleanupOldRollup()`.

```
function perform(address[] memory validators) external
    // tidy up the old rollup - pause it and refund sta
    cleanupOldRollup();
```

This subroutine pauses the old rollup contract and attempts to refund existing stakers.

```
        function cleanupOldRollup() private {
            IOldRollupAdmin(address(OLD_ROLLUP)).pause();

    >>      uint64 stakerCount = ROLLUP_READER.stakerCount();
            // since we for-loop these stakers we set an arbit
            // expect any instances to have close to this numb
            if (stakerCount > 50) {
                stakerCount = 50;
            }
            for (uint64 i = 0; i < stakerCount; i++) {
    >>          address stakerAddr = ROLLUP_READER.getStakerAd
                OldStaker memory staker = ROLLUP_READER.getSta
                if (staker.isStaked && staker.currentChallenge
                    address[] memory stakersToRefund = new add
                    stakersToRefund[0] = stakerAddr;

                    IOldRollupAdmin(address(OLD_ROLLUP)).force
                }
            }

            // upgrade the rollup to one that allows validator
            DoubleLogicUUPSUpgradeable(address(OLD_ROLLUP)).up
        }
```

This function contains a bug that prevents execution of the subsequent procedures. Let's check the `forceRefundStaker` in the old rollup contract.

According to: https://docs.arbitrum.io/build-decentralized-apps/reference/useful-addresses

Proxy:
https://etherscan.io/address/0x5eF0D09d1E6204141B4d37530808eD19f60FBa35
Implementation:
https://etherscan.io/address/0x72f193d0f305f532c87a4b9d0a2f407a3f4f585f#code

RollupAdminLogic.sol

```
    function forceRefundStaker(address[] calldata staker) e
        require(staker.length > 0, "EMPTY_ARRAY");
        for (uint256 i = 0; i < staker.length; i++) {
            require(_stakerMap[staker[i]].currentChallenge
            reduceStakeTo(staker[i], 0);
>>          turnIntoZombie(staker[i]);
        }
        emit OwnerFunctionCalled(22);
    }
```

RollupCore.sol

```
    function turnIntoZombie(address stakerAddress) interna
        Staker storage staker = _stakerMap[stakerAddress];
        _zombies.push(Zombie(stakerAddress, staker.latestSt
>>      deleteStaker(stakerAddress);
    }

    function deleteStaker(address stakerAddress) private {
        Staker storage staker = _stakerMap[stakerAddress];
        require(staker.isStaked, "NOT_STAKED");
        uint64 stakerIndex = staker.index;
        _stakerList[stakerIndex] = _stakerList[_stakerList
        _stakerMap[_stakerList[stakerIndex]].index = staker
>>      _stakerList.pop();
        delete _stakerMap[stakerAddress];
    }
```

From the above code, it is evident that the staker's address is eventually deleted from the `_stakerList`, causing the array to shrink. As a result, the `cleanupOldRollup` function will throw an "array out-of-bounds" error

because it tries to iterate through an array with the original number of elements.

Coded POC

Here we use mainnet fork with only the `cleanupOldRollup` function.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.17;

import {Test} from "forge-std/Test.sol";
import "forge-std/console.sol";

struct OldStaker {
    uint256 amountStaked;
    uint64 index;
    uint64 latestStakedNode;
    // currentChallenge is 0 if staker is not in a challeng
    uint64 currentChallenge; // 1. cannot have current cha
    bool isStaked; // 2. must be staked
}

interface IOldRollup {
    function pause() external;
    function forceRefundStaker(address[] memory stacker) ex
    function getStakerAddress(uint64 stakerNum) external vi
    function stakerCount() external view returns (uint64);
    function getStaker(address staker) external view retur
}

contract C4 is Test {
    IOldRollup oldRollup;
    address admin;
    function setUp() public {
        uint256 forkId = vm.createFork("https://rpc.ankr.co
        vm.selectFork(forkId);
        oldRollup = IOldRollup(0x5eF0D09d1E6204141B4d375308
        admin = 0x3ffFbAdAF827559da092217e474760E2b2c3CeDd
```

```
        }

        function test_Cleanup() public {
            vm.startPrank(admin);
            oldRollup.pause();
            uint64 stakerCount = oldRollup.stakerCount();
            // since we for-loop these stakers we set an arbit
            // expect any instances to have close to this numb
            if (stakerCount > 50) {
                stakerCount = 50;
            }
            for (uint64 i = 0; i < stakerCount; i++) {
                // FAILS with panic: array out-of-bounds acces
                address stakerAddr = oldRollup.getStakerAddres
                OldStaker memory staker = oldRollup.getStaker(
                if (staker.isStaked && staker.currentChallenge
                    address[] memory stakersToRefund = new add
                    stakersToRefund[0] = stakerAddr;
                    oldRollup.forceRefundStaker(stakersToRefun
                }
            }
        }
    }
```

Tools Used

Foundry

Recommended Mitigation Steps

```
        function cleanupOldRollup() private {
            IOldRollupAdmin(address(OLD_ROLLUP)).pause();

            uint64 stakerCount = ROLLUP_READER.stakerCount();
            // since we for-loop these stakers we set an arbit
            // expect any instances to have close to this numb
            if (stakerCount > 50) {
```

```
                stakerCount = 50;
            }
+           for (uint64 i = 0; i < stakerCount;) {
                address stakerAddr = ROLLUP_READER.getStakerAd
                OldStaker memory staker = ROLLUP_READER.getSta
                if (staker.isStaked && staker.currentChallenge
                    address[] memory stakersToRefund = new add
                    stakersToRefund[0] = stakerAddr;

                    IOldRollupAdmin(address(OLD_ROLLUP)).force
+                   stakerCount -= 1;
+               } else {
+                   i++;
+               }
            }
```

godzillaba (Arbitrum) confirmed

gzeoneth (Arbitrum) commented:

> Fixed with https://github.com/OffchainLabs/bold/pull/654/.

Picodes (judge) commented:

> Keeping this report as best as the mitigation takes into account the `if` condition.

Low Risk and Non-Critical Issues

For this audit, 23 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Sathish9098** received the top score from the judge.

The following wardens also submitted reports: ladboy233, Rhaydden, dontonka, K42, slvDev, Dup1337, xuwinnie, SpicyMeatball, bronze_pickaxe, fyamf, hihen, ZanyBonzy, forgebyola, Takarez, twcctop, Audinarey, josephdara, guhu95, zanderbyte, carlitox477, LessDupes, *and* KupiaSec.

[L-01] Risk of Confirming Assertion Prematurely if `totalTimeUnrivaled` Equals `confirmationThresholdBlock`

The current check if (totalTimeUnrivaled < confirmationThresholdBlock) only reverts when totalTimeUnrivaled is strictly less than confirmationThresholdBlock. This means that if totalTimeUnrivaled is exactly equal to confirmationThresholdBlock, the condition is not met, and the code proceeds without reverting.

However, in the context of confirming assertions, this can be problematic. If the totalTimeUnrivaled is exactly equal to confirmationThresholdBlock, it implies that the confirmationThresholdBlock has not been fully passed yet. To ensure that the assertion is confirmed only after the required confirmation blocks have fully passed, the check should also include the case where totalTimeUnrivaled is equal to confirmationThresholdBlock.

```
FILE:2024-05-arbitrum-foundation/src/challengeV2/libraries/

if (totalTimeUnrivaled < confirmationThresholdBlock) {
            revert InsufficientConfirmationBlocks(totalTime
        }
```

https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/6f861c85b281a29f04daacfe17a2099d7dad5f8f/src/challengeV2/libraries/EdgeChallengeManagerLib.sol#L741-L743

Recommended Mitigation

```
- if (totalTimeUnrivaled < confirmationThresholdBlock) {
+ if (totalTimeUnrivaled <= confirmationThresholdBlock) {
            revert InsufficientConfirmationBlocks(totalTime
        }
```

[L-02] `mandatoryBisectionHeight()` not return expected results

Expected Result

The documentation states: `Returns the highest power of 2 in the differing lower bits of start and end.` This means the function should identify the highest power of 2 in the bits where start and end differ.

Actual Result

The function does not return the highest power of 2 in the differing lower bits; rather, it uses the most significant differing bit to create a mask and apply it to (end - 1).

```
FILE: 2024-05-arbitrum-foundation/src/challengeV2/libraries
/EdgeChallengeManagerLib.sol

 function mandatoryBisectionHeight(uint256 start, uint256 e
        if (end - start < 2) {
            revert HeightDiffLtTwo(start, end);
        }
        if (end - start == 2) {
            return start + 1;
        }

        uint256 diff = (end - 1) ^ start;
```

```
        uint256 mostSignificantSharedBit = UintUtilsLib.mos
        uint256 mask = type(uint256).max << mostSignificant
        return ((end - 1) & mask);
    }
```

https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/6f861c85b281a29f04daacfe17a2099d7dad5f8f/src/challengeV2/libraries/EdgeChallengeManagerLib.sol#L574-L588

[L-03] Misleading comment in `setOutbox()` function

The current comment indicates that the function is adding a contract authorized to put messages into the rollup's inbox, but the function itself is setting an outbox contract and registering it with the bridge, rather than directly dealing with the inbox

```
FILE: 2024-05-arbitrum-foundation/src/rollup/RollupAdminLo

/**
     * @notice Add a contract authorized to put messages in
     * @param _outbox Outbox contract to add
     */
    function setOutbox(IOutbox _outbox) external override
        outbox = _outbox;
        bridge.setOutbox(address(_outbox), true);
        emit OwnerFunctionCalled(0);
    }
```

https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/6f861c85b281a29f04daacfe17a2099d7dad5f8f/src/rollup/RollupAdminLogic.sol#L113-L121

Recommended Mitigation

Here's a revised version of the comment to more accurately reflect the function's purpose.

```
/**
 * @notice Set the outbox contract for the rollup and autho
 * @param _outbox The outbox contract to be set and author
 */
function setOutbox(IOutbox _outbox) external override {
    outbox = _outbox;
    bridge.setOutbox(address(_outbox), true);
    emit OwnerFunctionCalled(0);
}
```

[L-04] `if (ard.assertionHash != args.claimId) {` Potentially Redundant Check Between assertionHash and claimId in `layerZeroTypeSpecificChecks()` function

There is no possibility that values are different. `args.claimId` value only assigned to `ard.assertionHash` when creating `ard` variable .

```
FILE: Breadcrumbs2024-05-arbitrum-foundation/src/challengeV
/EdgeChallengeManager.sol

 ard = AssertionReferenceData(
                args.claimId,
                claimStateData.prevAssertionHash,
                assertionChain.isPending(args.claimId),
                assertionChain.getSecondChildCreationBlock
                predecessorStateData.assertionState,
                claimStateData.assertionState
            );
```

https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/6f861c85b281a29f04daacfe17a2099d7dad5f8f/src/challengeV2/EdgeChallengeManager.sol#L409

```
FILE: 2024-05-arbitrum-foundation/src/challengeV2/libraries
/EdgeChallengeManagerLib.sol

if (ard.assertionHash != args.claimId) {
                revert AssertionHashMismatch(ard.assertionH
            }
```

https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/6f861c85b281a29f04daacfe17a2099d7dad5f8f/src/challengeV2/libraries/EdgeChallengeManagerLib.sol#L232-L234

[L-05] Incorrect Comment Describing Execution State Check in `layerZeroTypeSpecificChecks()` Function

```
FILE:2024-05-arbitrum-foundation/src/challengeV2/libraries
/EdgeChallengeManagerLib.sol

// check the start and end execution states exist, the bloc
            if (ard.startState.machineStatus == MachineStat
                revert EmptyStartMachineStatus();
            }
            if (ard.endState.machineStatus == MachineStatus
                revert EmptyEndMachineStatus();
            }
```

This comment suggests that the check is verifying the existence of the start and end execution states and implies something about block hash entries, which is not what the code does.

Recommended Mitigation

Add appropriate comments.

```
// Check that the start and end execution states are not ru
// The machine status should not be 'RUNNING' for either th
            if (ard.startState.machineStatus == MachineStat
                revert EmptyStartMachineStatus();
            }
            if (ard.endState.machineStatus == MachineStatus
                revert EmptyEndMachineStatus();
            }
```

[L-06] Consequences of Missing Validation in critical `setMinimumAssertionPeriod` and `setBaseStake` Functions

The function setMinimumAssertionPeriod sets the minimumAssertionPeriod state variable to newPeriod without performing any validation checks. This lack of validation can lead to the assignment of invalid or unreasonable values, which can adversely affect the contract's behavior and security. Same to `setBaseStake()` function.

```
FILE: 2024-05-arbitrum-foundation/src/rollup
/RollupAdminLogic.sol

function setMinimumAssertionPeriod(uint256 newPeriod) exte
        minimumAssertionPeriod = newPeriod;
        emit OwnerFunctionCalled(8);
    }

function setBaseStake(uint256 newBaseStake) external overr
        baseStake = newBaseStake;
        emit OwnerFunctionCalled(12);
```

```
    }
```

https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/6f861c85b281a29f04daacfe17a2099d7dad5f8f/src/rollup/RollupAdminLogic.sol#L204-L207

[L-07] Inefficient Array Resizing in `append()` Function

The gas cost of the append function might exceed the gas limit set for a transaction. This gas limit specifies the maximum amount of gas a user is willing to spend on a transaction. This can be problematic, especially in situations where the exact size of the appended data might not be known beforehand.

```
FILE: 2024-05-arbitrum-foundation/src/challengeV2/libraries
/ArrayUtilsLib.sol

function append(bytes32[] memory arr, bytes32 newItem) inte
        bytes32[] memory clone = new bytes32[](arr.length -
        for (uint256 i = 0; i < arr.length; i++) {
            clone[i] = arr[i];
        }
        clone[clone.length - 1] = newItem;
        return clone;
    }
```

https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/6f861c85b281a29f04daacfe17a2099d7dad5f8f/src/challengeV2/libraries/ArrayUtilsLib.sol#L16

Recommended Mitigation

Use a resizable array library like OwnableArray from the OpenZeppelin Contracts library (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol). These libraries offer functions to append elements dynamically without the need to manually allocate and copy the entire array.

[L-08] `block.number >= assertion.createdAtBlock + prevConfig.confirmPeriodBlocks` not implemented as per docs

Comment: “Check that deadline has passed”

This implies the deadline should be strictly in the past, meaning the current block number must be greater than the deadline block number.

```
FILE: 2024-05-arbitrum-foundation/src/rollup
/RollupUserLogic.sol

// Check that deadline has passed
        require(block.number >= assertion.createdAtBlock +
```

Recommended Mitgation

1. Change comment as per implementation.

```
// Check that deadline has passed or equal
        require(block.number >= assertion.createdAtBlock +
```

2. Change code as per docs.

```
// Check that deadline has passed
```

```
        require(block.number > assertion.createdAtBlock + p
```

[L-09] `reduceStakeTo` Function Allows Call Even with Zero Staked Amount

The reduceStakeTo function does not have any check to prevent it from being called when staker.amountStaked is zero. This means if a staker's amountStaked is zero, the function can still be called, and it would set the amountStaked to the target value, which could result in unintended behavior.

```
FILE: 2024-05-arbitrum-foundation/src/rollup
/RollupCore.sol

 function reduceStakeTo(address stakerAddress, uint256 targ
        Staker storage staker = _stakerMap[stakerAddress];
        address withdrawalAddress = staker.withdrawalAddres
        uint256 current = staker.amountStaked;
        require(target <= current, "TOO_LITTLE_STAKE");
        uint256 amountWithdrawn = current - target;
        staker.amountStaked = target;
        increaseWithdrawableFunds(withdrawalAddress, amount
        emit UserStakeUpdated(stakerAddress, withdrawalAddr
        return amountWithdrawn;
    }
```

https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/6f861c85b281a29f04daacfe17a2099d7dad5f8f/src/rollup/RollupCore.sol#L300-L310

Recommended Mitigation

`staker.amountStaked` is 0 then the call should be reverted.

```
require(current > 0,"zero amount staked");
```

[N-01] `newStakeOnNewAssertion` function reverts all calls when contract paused

There is a potential flaw related to the function stakeOnNewAssertion being called within newStakeOnNewAssertion. The stakeOnNewAssertion function has the whenNotPaused modifiers, meaning it should only be executed only when the contract is not paused. However, these conditions are not enforced within the newStakeOnNewAssertion function, leading to potential inconsistencies.

```
FILE: 2024-05-arbitrum-foundation/src/rollup
/RollupUserLogic.sol

function newStakeOnNewAssertion(
        uint256 tokenAmount,
        AssertionInputs calldata assertion,
        bytes32 expectedAssertionHash,
        address withdrawalAddress
    ) public {
        require(withdrawalAddress != address(0), "EMPTY_WI
        _newStake(tokenAmount, withdrawalAddress);
        stakeOnNewAssertion(assertion, expectedAssertionHa
        /// @dev This is an external call, safe because it
        receiveTokens(tokenAmount);
    }

function stakeOnNewAssertion(AssertionInputs calldata asse
        public
        onlyValidator
        whenNotPaused
    {
```

https://github.com/code-423n4/2024-05-arbitrum-foundation/blob/6f861c85b281a29f04daacfe17a2099d7dad5f8f/src/rollup/RollupUserLogic.sol#L163-L167

Recommended Mitigation

```
FILE: 2024-05-arbitrum-foundation/src/rollup
/RollupUserLogic.sol

function newStakeOnNewAssertion(
        uint256 tokenAmount,
        AssertionInputs calldata assertion,
        bytes32 expectedAssertionHash,
        address withdrawalAddress
-    ) public {
+    ) public whenNotPaused {
        require(withdrawalAddress != address(0), "EMPTY_WI
        _newStake(tokenAmount, withdrawalAddress);
        stakeOnNewAssertion(assertion, expectedAssertionHa
        /// @dev This is an external call, safe because it
        receiveTokens(tokenAmount);
    }
```

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top