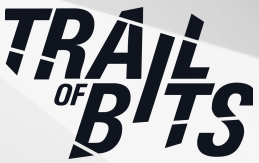

BoLD and DAC Rewards Updates

Security Assessment

August 5, 2024

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

Prepared by: **Gustavo Grieco and Simone Monica**

About Trail of Bits

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

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

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

The following engineering director was associated with this project:

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

The following consultants were associated with this project:

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Simone Monica, Consultant
simone.monica@trailofbits.com

Project Timeline

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

Date	Event
June 10, 2024	Pre-project kickoff call
June 20, 2024	Delivery of report draft
June 20, 2024	Report readout meeting
August 5, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of a series of changes to the BoLD contract and the DAC Rewards contract. These changes include fixes for previous security assessments as well as code quality improvements. The DAC Reward also was modified to support distribution on rewards on the Optimism rollups.

A team of two consultants conducted the review from June 10 to June 20, 2024, for a total of three engineer-weeks of effort. Our testing efforts focused on a selected list of PRs provided by Offchain Labs. With full access to source code and documentation, we performed a manual process to review the code changes as well as some limited mutation testing to assess the quality of the unit tests.

Observations and Impact

We spent the majority of our time on the BOLD changes, checking whether any of them would allow a malicious party to win a challenge by confirmation or prevent an honest party from winning challenges (e.g., by making it impossible for an honest party to continue participating in a challenge). We also checked for possible misconfiguration of the new parameters and features. We reviewed changes to the staking pool contracts to assess whether a malicious user could steal tokens or withdraw their tokens before a challenge is finished.

We did not uncover any serious issues in the implementation. However, we found one issue in the DAC Reward code due to a gas optimization that would make the transaction revert for certain ERC20 tokens (TOB-DACR-1).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Offchain Labs take the following steps:

- Remediate the findings uncovered during this review.
- Improve the quality of the tests and add mutation testing to the software development lifecycle to improve the quality of the testing suite (see appendix B).

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	0
Informational	0
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Undefined Behavior	1

Project Goals

The engagement was scoped to provide a security assessment of the Offchain's BoLD and DAC Rewards contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the fixes for issues identified in previous security audits sufficient?
- Have the code changes introduced any security or correctness issues?

Project Targets

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

BoLD

Repository	https://github.com/OffchainLabs/bold
Version	`6b42a38fc28f9b6a7864001fe797b9f07dad6357`
Type	Solidity
Platform	Ethereum/Arbitrum

DAC-REWARDS

Repository	https://github.com/OffchainLabs/fund-distribution-contracts
Version	`b390734d934e6d8b05b78e5ef38739c4b8e668b6`
Type	Solidity
Platform	Ethereum/Arbitrum/Optimism

Project Coverage

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

- #609 adds an event to track timer cache updates.
- #623 removes `EdgeConfirmedByChildren` and `EdgeConfirmedByClaim` events, since they are unused with bottom-up timers.
- #627 introduces a feature to pin a withdrawal address on creating a new stake (similar to how beacon chain withdrawal credential works), thus reducing the risk of having a hot private key in the validator software.
- #631 fixes a mainly cosmetic error where attempts to confirm an already-confirmed edge by time revert with `RivalEdgeConfirmed`. This PR makes the attempt revert with `EdgeNotPending` instead.
- #617 merges the delay buffer feature that was previously audited.
- #637 and #641 introduce an optimization when a stake is slashed: since the loser stake escrow may not be able to call `withdraw`, the confiscated stake is sent directly to the winner instead of adding to its withdrawable amount.
- #639 changes the documentation of the `firstRival` function.
- #619 introduces a new staking pool similar to the existing assertion staking pool.
- #628 merges changes from Stylus.
- #643 changes `ChallengeManager` contract to inherit the allowlist from the `Rollup` contract, and imposes a limit to ensure that each validator can create only one root edge per sub-challenge.
- #645 removes the old challenge manager.
- #642 introduces several changes:
    - The bold upgrade action contract has been modified to update the sequencer inbox with the delay buffer feature.
    - The action contract also upgrades the inbox to deprecate methods that assume that `tx.origin == msg.sender` implies the use of EOA sender. This upgrade preempts breaking changes caused by future Ethereum hard forks.

    - This PR also changes the `BOLDUpgradeAction` to persist the `Bridge`, `Outbox`, and `RollupEventInbox` upgrades.

- #6 in the bold private repository adds the beta parameters described in the section 4.8.1 of the bold paper.
- #8 in the bold private repository applies the same fix as #631 but is confirmed by the OSP.
- #652 enforces that the `delayBlock` parameter is consistent with `delaySeconds` to ensure that the force inclusion window does not inadvertently decrease.
- #655 addresses the issue #8 found in the C4 bug contest. It is a short-term workaround that prevents the admin from reducing the assertion base stake amount.
- #654 addresses the issue #5 found in the C4 bug contest.
- #651 adds more descriptive events to the sequencer inbox and rollup contracts.
- #650 uses OpenZeppelin's enumerable set for the validator allowlist. This makes it easier to retrieve the list of validators off-chain.
- #653 introduces the following changes to the staking and withdrawal flow to make it easier and safer for validators to handle funds:
    - Adds a new `returnOldDepositFor` function, allowing the withdrawal address to trigger withdrawals.
    - Adds a new `newStake` function, allowing validators to become staked with any amount (even 0).
- #659 addresses the issue #2 in the C4 bug contest.
- #664 forbids a pool to be created with an empty edge id or assertion hash.
- #665 allows configuration of `validatorAfkBlocks`, with the option to set it to 0 to disable this feature entirely.
- #29 in the fund distribution repository allows reward distribution in Optimism chains.

We reviewed this code looking for usual flaws in Solidity code as well as any issues that would allow a malicious party to win a challenge by confirmation or prevent an honest party from winning challenges (e.g., by making it impossible for an honest party to continue participating in a challenge). We also checked for possible misconfiguration of the new parameters and features. We reviewed the staking pool contracts changes to check

whether it is possible to steal tokens and whether a malicious user can withdraw their tokens before the challenge is finished.

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:

- We have not reviewed in detail how the BoLD codebase evolved. Instead, we used the diff/PRs provided by Offchain Labs to bound the scope of this assessment.

Summary of Findings

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

ID	Title	Type	Severity
1	routeToken function may fail for certain ERC20 tokens	Undefined Behavior	**Medium**

Detailed Findings

1. routeToken function may fail for certain ERC20 tokens

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-DACR-1
Target: `src/FeeRouter/ArbChildToParentRewardRouter.sol`, `src/FeeRouter/OpChildToParentRewardRouter.sol`, `src/FeeRouter/ParentToChildRewardRouter.sol`	

Description

The `routeToken` function can revert for ERC20 tokens that require the allowance to be set to 0 when calling the `approve` function, such as USDT.

The function has an optimization where it approves `amount + 1` to the gateway, which will transfer the amount of tokens in a following call. The idea is to never change the allowance storage slot from a non-zero to a zero value and saving gas. However, tokens such as USDT require the current allowance to be 0 when a user calls the `approve` function; in this case, this makes the `routeToken` function revert.

```
function routeToken(address parentChainTokenAddr, uint256 maxSubmissionCost, uint256
gasLimit, uint256 maxFeePerGas)
     public
     payable
 {
     ...
     // approve amount on gateway, adding 1 so storage slot doesn't get set to 0,
saving gas.
     IERC20(parentChainTokenAddr).approve(gateway, amount + 1);

     ...
 }
```

Figure 1.1: Snippet of the `routeToken` function (src/FeeRouter/ParentToChildRewardRouter.sol#L162-L163)

Exploit Scenario

The `ParentToChildRewardRouter` contract is deployed with `parentChainTokenAddr` set to the USDT address. A user calls the `routeToken` function, but it unexpectedly reverts.

Recommendations

Short term, remove this optimization or use the `forceApprove` function in the `SafeERC20` library after verifying that this function would still save gas.

Long term, when developing a contract that interacts with ERC20 tokens, consider all possible different implementations and which implementations your contract should support.

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. Mutation Testing

During our review, we ran two mutation testing tools, Necessist and `slither-mutate`, to identify potential shortcomings in the test suite and implementation.

To use Necessist, run the following command:

```
necessist --framework foundry
test/challengeV2/EdgeChallengeManager.t.sol
```

To use `slither-mutate`, run the following command:

```
slither-mutate src/challengeV2/EdgeChallengeManager.sol
--test-cmd='forge test'
```

The same commands can be used for other contracts.

Note that due to a Solidity issue regarding unicode characters #14733, `slither-mutate` will use an incorrect source mapping for the mutation; as a result, any unicode characters should be removed. The `EdgeChallengeManager.sol` includes the unicode characters β and ∗ in the comments on lines 86, 94, and 103.

We tested the following contracts:

- Necessist
  - ChallengeEdgeLib.t.sol
  - EdgeChallengeManager.t.sol
  - EdgeChallengeManagerLib.t.sol
  - Rollup.t.sol
- Slither-mutate
  - AssertionStakingPool.sol
  - EdgeChallengeManager.sol
  - EdgeChallengeManagerLib.sol
  - RollupCore.sol
  - RollupUserLogic.sol
  - RollupAdminLogic.sol

After reviewing the results, we identified missing test coverage for the cases described in the following sections.

1. Timer cache is never tested for a cache hit
The `updateTimerCache` function is never tested for a cache hit in the highlighted line.

```
function updateTimerCache(EdgeStore storage store, bytes32 edgeId, uint256 newValue,
```

```
uint256 maximumCachedTime)
    internal
    returns (bool, uint256)
{
    uint256 currentAccuTimer = validateCurrentTimer(store, edgeId,
maximumCachedTime);
    newValue = newValue > type(uint64).max ? type(uint64).max : newValue;
    // only update when increased
    if (newValue > currentAccuTimer) {
        store.edges[edgeId].totalTimeUnrivaledCache = uint64(newValue);
        return (true, newValue);
    }
    return (false, currentAccuTimer);
}
```

Figure B.1: The `updateTimerCache` function (src/challengeV2/libraries/EdgeChallengeManagerLib.sol#L516-L528)

Additionally, the branch where `newValue > type(uint64).max` is never tested.

2. Failure to check returned value of current timer

Related to the previous finding, the `updateTimerCache` function is not properly tested, as the current timer can be reduced to zero without producing any failure in the tests:

```
function validateCurrentTimer(EdgeStore storage store, bytes32 edgeId, uint256
maximumCachedTime)
    internal
    view
    returns (uint256)
{
    uint256 currentAccuTimer = store.edges[edgeId].totalTimeUnrivaledCache;
    if (currentAccuTimer >= maximumCachedTime) {
        revert CachedTimeSufficient(currentAccuTimer, maximumCachedTime);
    }
    return currentAccuTimer;
}
```

Figure B.2: The `validateCurrentTimer` function (src/challengeV2/libraries/EdgeChallengeManagerLib.sol#L500-L510)

This function is called in other contexts, but the returned value is never checked in those cases.

3. Total unrivaled time is always zero in the tests

In the context of the tests, the `timeUnrivaled` function always returns zero:

```
function updateTimerCacheByClaim(
    EdgeStore storage store,
    bytes32 edgeId,
    bytes32 claimingEdgeId,
```

```
    uint8 numBigStepLevel,
    uint256 maximumCachedTime
) internal returns (bool, uint256) {
    // calculate the time unrivaled without inheritance
    uint256 totalTimeUnrivaled = timeUnrivaled(store, edgeId);
    checkClaimIdLink(store, edgeId, claimingEdgeId, numBigStepLevel);
    totalTimeUnrivaled += store.edges[claimingEdgeId].totalTimeUnrivaledCache;
    return updateTimerCache(store, edgeId, totalTimeUnrivaled, maximumCachedTime);
}
```

Figure B.3: The `updateTimerCacheByClaim` function (src/challengeV2/libraries/EdgeChallengeManagerLib.sol#L537-L549)

4. **Overflow assertions are never tested**

There is a gap in the testing of overflow assertions, which affects important sections of the `createNewAssertion` function:

```
function createNewAssertion(
    AssertionInputs calldata assertion,
    bytes32 prevAssertionHash,
    bytes32 expectedAssertionHash
) internal returns (bytes32 newAssertionHash, bool overflowAssertion) {
        ...
        if (assertion.afterState.machineStatus != MachineStatus.ERRORED &&
afterStateCmpMaxInbox < 0) {
            // If we didn't reach the target next inbox position, this is an
overflow assertion.
            overflowAssertion = true;
            // This shouldn't be necessary, but might as well constrain the
assertion to be non-empty
            require(afterGS.comparePositions(beforeGS) > 0, "OVERFLOW_STANDSTILL");
        }
        ...
```

Figure B.4: Snippet of the `createNewAssertion` function (src/rollup/RollupCore.sol#L378-L531)

4. **Edge staking fuzz test missing precondition**

The `testProperInitialization` fuzz test is missing a precondition that sometimes causes it to fail. The new code requires that the assertion is not zero.

```
function testProperInitialization(bytes32 edgeId) public {
    IEdgeStakingPool stakingPool =
stakingPoolCreator.createPool(address(challengeManager), edgeId);

    assertEq(address(stakingPoolCreator.getPool(address(challengeManager), edgeId)),
address(stakingPool));

    assertEq(address(stakingPool.challengeManager()), address(challengeManager));
    assertEq(stakingPool.edgeId(), edgeId);
```

```
    assertEq(address(stakingPool.stakeToken()), address(token));
}
```

Figure B.5: The `testProperInitialization` function (`test/stakingPool/EdgeStakingPool.t.sol#L44-L52`)

If the `edgeId` is zero, then the code will revert in the pool creation:

```
constructor(
    address _challengeManager,
    bytes32 _edgeId
) AbsBoldStakingPool(address(EdgeChallengeManager(_challengeManager).stakeToken()))
{
    if (_edgeId == bytes32(0)) {
        revert EmptyEdgeId();
    }
    challengeManager = _challengeManager;
    edgeId = _edgeId;
}
```

Figure B.6: The `EdgeStakingPool` constructor (`src/assertionStakingPool/EdgeStakingPool.sol#L35-L44`)

5. **The `makeStakeWithdrawableAndWithdrawBackIntoPool` function is never tested**
The `makeStakeWithdrawableAndWithdrawBackIntoPool` function of the `AssertionStakingPool` contract is never tested.

6. confirmEdgeByTime corner case is never tested
The case where the total time unrivaled is less than the confirmation threshold blocks is never reached and needs an additional unit test.

```
function confirmEdgeByTime(
    EdgeStore storage store,
    bytes32 edgeId,
    uint64 claimedAssertionUnrivaledBlocks,
    uint64 confirmationThresholdBlock
) internal returns (uint256) {
    if (!store.edges[edgeId].exists()) {
        revert EdgeNotExists(edgeId);
    }

    uint256 totalTimeUnrivaled = timeUnrivaledTotal(store, edgeId);

    // since sibling assertions have the same predecessor, they can be viewed as
    // rival edges. Adding the assertion unrivaled time allows us to start the
confirmation
    // timer from the moment the first assertion is made, rather than having to wait
until the
    // second assertion is made.
    totalTimeUnrivaled += claimedAssertionUnrivaledBlocks;
```

```
    if (totalTimeUnrivaled < confirmationThresholdBlock) {
        revert InsufficientConfirmationBlocks(totalTimeUnrivaled,
confirmationThresholdBlock);
    }
    ...
}
```

Figure B.7: The `confirmEdgeByTime` function (src/challengeV2/libraries/EdgeChallengeManagerLib.sol#L744-L773)