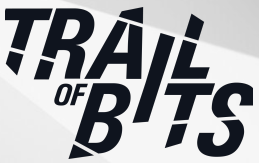

Offchain Labs Sequencer Liveness

Security Assessment (Summary Report)

March 31, 2025

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

Prepared by: **Gustavo Grieco and Tarun Bansal**

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:

Jim Miller, Engineering Director, Blockchain
jim.miller@trailofbits.com

The following consultants were associated with this project:

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Project Timeline

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

Date	Event
March 10, 2025	Delivery of report draft
March 11, 2025	Report readout meeting
March 31, 2025	Delivery of final summary report

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of the core code of the Arbitrum Sequencer, specifically the revision at commit `fcb4018`.

The Sequencer is a pivotal component of the Arbitrum network and is responsible for efficiently ordering and processing transactions. It plays a crucial role in providing users with fast transaction confirmations while maintaining the security and integrity of the blockchain. In this review, we reviewed the core components, including the loop that validates and feeds transactions into ArbOS. The focus of this audit are issues that affect the correctness or liveness of the Sequencer.

A team of two consultants conducted the review from February 24 to March 7, 2025, for a total of three engineer-weeks of effort. With full access to source code and documentation, we performed a manual review of the code in scope.

Observations and Impact

This engagement revealed an issue related to a potential denial-of-service vector using StateDB/ArboOS state when processing transactions. While the severity of the issue was rated as low, Offchain Labs should reevaluate the system to consider how resource consumption degrades the performance of the Sequencer and which alternatives are more likely to mitigate current and future issues.

Additionally, we provide a prototype of a fuzz test for the Sequencer in appendix B.

Recommendations

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

- Fix issue TOB-ARB-SEQ-1.
- Consider adding a fuzz test similar to the one described in appendix B.

1. Uncached access to statedb/arbos when using hooks options

Severity: **Low**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-ARB-SEQ-1
Target: `tx_options.go`, `tx_pre_checker.go`	

Description

The repeated access of go-ethereum/arbos state can be a potential vector for a denial-of-service attack in specific Sequencer configurations.

The Sequencer can be configured with a number of different options, which run in specific hooks:

```
type ConditionalOptions struct {
	KnownAccounts  map[common.Address]RootHashOrSlots `json:"knownAccounts"`
	BlockNumberMin *math.HexOrDecimal64
`json:"blockNumberMin,omitempty"`
	BlockNumberMax *math.HexOrDecimal64
`json:"blockNumberMax,omitempty"`
	TimestampMin   *math.HexOrDecimal64
`json:"timestampMin,omitempty"`
	TimestampMax   *math.HexOrDecimal64
`json:"timestampMax,omitempty"`
}
```

Figure 1.1: Options for hooks

These options are used during the execution of the filter that runs before a transaction is executed:

```
func (s *Sequencer) preTxFilter(_ *params.ChainConfig, header *types.Header, statedb
*state.StateDB, _ *arbosState.ArbosState, tx *types.Transaction, options
*arbitrum_types.ConditionalOptions, sender common.Address, l1Info *arbos.L1Info)
error {
	if s.nonceCache.Caching() {
		stateNonce := s.nonceCache.Get(header, statedb, sender)
		err := MakeNonceError(sender, tx.Nonce(), stateNonce)
		if err != nil {
```

```
                nonceCacheRejectedCounter.Inc(1)
                return err
        }
    }
    if options != nil {
        err := options.Check(l1Info.L1BlockNumber(), header.Time, statedb)
           ...
```

Figure 1.2: Header of the `preCheckTx` function

The options will be used in the `Check` function. In particular, this function can query the `GetStorageRoot` if the user specifies a range of known accounts with nonzero root hashes:

```
func (o *ConditionalOptions) Check(l1BlockNumber uint64, l2Timestamp uint64, statedb
*state.StateDB) error {
    if o.BlockNumberMin != nil && l1BlockNumber < uint64(*o.BlockNumberMin) {
        return NewRejectedError("BlockNumberMin condition not met")
    }
    if o.BlockNumberMax != nil && l1BlockNumber > uint64(*o.BlockNumberMax) {
        return NewRejectedError("BlockNumberMax condition not met")
    }
    if o.TimestampMin != nil && l2Timestamp < uint64(*o.TimestampMin) {
        return NewRejectedError("TimestampMin condition not met")
    }
    if o.TimestampMax != nil && l2Timestamp > uint64(*o.TimestampMax) {
        return NewRejectedError("TimestampMax condition not met")
    }
    for address, rootHashOrSlots := range o.KnownAccounts {
        if rootHashOrSlots.RootHash != nil {
            storageRoot := statedb.GetStorageRoot(address)
                ...
```

Figure 1.3: Header of the `Check` function

However, the query of `GetStorageRoot` is not protected by any cache, and it could be expensive if called multiple times.

Additionally, in the `PreCheckTx` function, there is an another uncached access to nonces from the StateDB object, as well as several uses of the ArbOS state:

```
func PreCheckTx(bc *core.BlockChain, chainConfig *params.ChainConfig, header
*types.Header, statedb *state.StateDB, arbos *arbosState.ArbosState, tx
*types.Transaction, options *arbitrum_types.ConditionalOptions, config
```

```
*TxPreCheckerConfig) error {
	..
	baseFee := header.BaseFee
	if config.Strictness < TxPreCheckerStrictnessLikelyCompatible {
		baseFee, err = arbos.L2PricingState().MinBaseFeeWei()
		if err != nil {
			return err
		}
	}
	if arbmath.BigLessThan(tx.GasFeeCap(), baseFee) {
		return fmt.Errorf("%w: address %v, maxFeePerGas: %s baseFee: %s",
core.ErrFeeCapTooLow, sender, tx.GasFeeCap(), header.BaseFee)
	}
	stateNonce := statedb.GetNonce(sender)
	if tx.Nonce() < stateNonce {
		return MakeNonceError(sender, tx.Nonce(), stateNonce)
	}
	…
```

Figure 1.4: Header of the `preCheckTx` function

Exploit Scenario

The owner of a Sequencer configures its hooks to check for certain known accounts. This causes the performance to be degraded for all users.

Recommendations

Short term, consider using a cached access for the call to `GetStorageRoot` as well as the ArbOS state itself.

Long term, review the bottlenecks across the components, particularly the ones that require direct access to external components.

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. Prototype of a Sequencer Fuzzer

For this engagement, we created a prototype of a fuzz test for the Sequencer that uses Go native fuzzing. This test covers the processing of simple L2 transactions with arbitrary nonces (which are essentially untrusted) as well as occasional reorgs:

```
func FuzzNonceQueue(f *testing.F) {
        f.Fuzz(func(t *testing.T, nonces []byte) {
        t.Parallel()
        ctx, cancel := context.WithCancel(context.Background())
        defer cancel()

        builder := NewNodeBuilder(ctx).DefaultConfig(t, false)
        builder.takeOwnership = false
        builder.execConfig.Sequencer.NonceFailureCacheSize = 8
        builder.execConfig.Sequencer.NonceFailureCacheExpiry = 100 *
time.Millisecond

        cleanup := builder.Build(t)
        defer cleanup()
        count := min(16, len(nonces));
        var completed atomic.Uint64
        for i := 0; i < count; i++ {
                println(i)
                value := uint64(nonces[i] % 16)
                println(value)
                builder.L2Info.GetInfoWithPrivKey("Owner").Nonce.Store(value)
                tx := builder.L2Info.PrepareTx("Owner", "Owner",
builder.L2Info.TransferGas, common.Big0, nil)
                go func() {
                        err := builder.L2.Client.SendTransaction(ctx, tx)
                        if err != nil {
                                println(err.Error())
                        }
                        completed.Add(1)
                }()
        }

        startMsgCount, err := builder.L2.ConsensusNode.TxStreamer.GetMessageCount()


        for {
                got := int(completed.Load())
                println(got)
                if got >= count {
                        break
                }
                if got == count / 2 {
                        err =
builder.L2.ConsensusNode.TxStreamer.ReorgTo(startMsgCount)
                        if err != nil {
```

```
                                println(err.Error())
                        }
                }
                time.Sleep(time.Millisecond * 10)
        }
        })
}
```

Figure A.1: The FuzzNonceQueue fuzz test

Moving forward, we recommend that Offchain Labs implement a “fuzzer-friendly mode” that avoids performing very CPU-intensive operations that can be skipped during a fuzzing campaign.

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.