Arbitrum Nitro

Security Assessment

October 10, 2022

Prepared for:

Harry Kalodner, Steven Goldfeder, and Ed Felten

Offchain Labs

Prepared by: **Nat Chin, 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 80+ 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

Offchain Labs engaged Trail of Bits to review the security of its Arbitrum Nitro system. From July 5 to August 19, 2022, a team of three consultants conducted a security review of the client-provided source code, with sixteen 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 the compromise of a smart contract, a loss of funds, or unexpected behavior in the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed dynamic testing of the target system, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could result in unexpected behavior. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	2
Low	8
Informational	4
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Configuration	2
Data Validation	5
Patching	2
Testing	1
Undefined Behavior	7

Notable Findings

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

- **Risk of panics resulting in message-processing interruptions (TOB-ArbOS-1, TOB-ArbOS-17)**
  We identified two scenarios that could cause ArbOS to panic when processing inbox messages and messages that include AnyTrust Data Availability Certificates. Panics in these situations would result in a denial of service on the chain.

- **Longer-than-expected block-creation time (TOB-ArbOS-3)**
  The computation involved in the retrieval of values used by the `NUMBER` and `BLOCKHASH` opcodes takes longer than expected.

- **Insufficient testing (TOB-ArbOS-4)**
  An incorrect calculation performed during the unpacking of internal transaction data causes the layer 2 pricing model to generate incorrect results.

- **Risk of unexpected behavior caused by the Classic–Nitro Migration (TOB-ArbOS-13)**
  Users may lose the funds associated with a retryable ticket that expires during the migration from Arbitrum Classic to Nitro.

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

Nat Chin, Consultant
natalie.chin@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 30, 2022	Pre-project kickoff call
July 11, 2022	Status update meeting #1
July 18, 2022	Status update meeting #2
July 25, 2022	Status update meeting #3
August 2, 2022	Status update meeting #4
August 8, 2022	Status update meeting #5
August 15, 2022	Status update meeting #6
August 22, 2022	Delivery of report draft; report readout meeting
October 10, 2022	Delivery of final report

Project Goals

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

- Does the code responsible for the migration from Arbitrum Classic to Arbitrum Nitro behave as expected?
- Will tickets that are redeemable in Arbitrum Classic be redeemable in Arbitrum Nitro?
- Are there appropriate access controls throughout the system?
- Could a participant perform a denial-of-service or spam attack against any of the components?
- Are incoming arguments validated and parsed correctly?
- Are the gas costs of the layer 1 (L1) and layer 2 (L2) opcodes appropriate?
- Could any of the on-chain components be manipulated by the sequencer, a validator, or any other user?

We also sought to answer the following questions regarding ArbOS:

- Does the ArbOS EVM implementation adhere to the behavior described in the Yellow Paper? If it deviates from that behavior, how do the deviations affect the correctness and security of the smart contracts deployed on Arbitrum?
- Are incoming messages properly parsed, validated, and processed?
- Is the ArbOS bookkeeping correct and updated when necessary? Is there any effect from its internal state that is not properly committed or reverted?

Project Targets

The engagement involved a review and testing of the following targets.

`nitro/arbos`

Repository	https://github.com/OffchainLabs/nitro
Version	`861ba3ca52b112eb545d23a4c3332c8df7d192ee`
Type	Go
Platform	L2 operating system

`nitro/contracts`

Repository	https://github.com/OffchainLabs/nitro
Version	`cc7bd52a5ba27087a86161073f272d1f79fefa0b`
Type	Solidity
Platform	Ethereum

`arbitrum`

Repository	https://github.com/OffchainLabs/arbitrum
Version	`9b0b581a97c15daa51fe6b3587c216dedc99d406`
Types	Solidity, C++, and Go
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:

Ethereum Smart Contracts

The Arbitrum Nitro system includes Ethereum smart contracts that manage and secure a rollup chain on L1. The most relevant contracts are listed below.

`Inbox`. The `Inbox` contract allows users to send messages to ArbOS. We reviewed the inbox's receipt of L2 messages, focusing on the impact that user-controlled input can have on the entire system. We manually reviewed the construction, validation, and delivery of messages.

`SequencerInbox` and `Bridge`. The `Bridge` contract executes cross-chain transactions sent from L2, and the `SequencerInbox` controls the inclusion of messages in the ArbOS inbox. We focused on the changes made since the last audit (which concluded in March 2022) and reviewed the contracts' interactions with ArbOS.

`HashProofHelper`. The `HashProofHelper` contract, which is part of the one-step-proof implementation, makes it possible to prove a preimage that is larger than the maximum data size. Our review of this contract focused on identifying flaws in its implementation.

`NitroMigrator`. This contract was implemented to enable the migration from Arbitrum Classic to Arbitrum Nitro by transferring ownership of the Classic contracts and pointing the system to the newly deployed contract versions. We focused on determining whether the state of the Classic or Nitro chain could be disrupted during or after the migration and whether external attackers could delay or block the migration.

ArbOS

ArbOS is the trusted L2 operating system. It isolates untrusted contracts from each other, tracks and limits their resource usage, and manages the mechanism that collects fees from users to fund the operation of a chain's validators.

ArbOS handles trusted and untrusted messages originating from Ethereum. We reviewed the handling of incoming messages and the flow of assets. Our review of the escrow mechanism, which allows certain assets to be saved in order to be collected or burned later, focused on how ether is handled and how gas is tracked and burned.

We also reviewed the translation of the EVM state, the L1 and L2 pricing models, and the changes made to the gas costs of the EVM opcodes used in Arbitrum Nitro. We looked for ways to disrupt or break the processing of blocks or the gas accounting.

Additionally, we looked for unexpected error conditions that could break important ArbOS security or correctness properties. We checked whether ArbOS could be forced to loop or to consume an excessive amount of resources when processing new incoming messages from L1.

We also analyzed the migration-related code, reviewing the process of exporting and then reimporting data and the effects of Arbitrum Classic's state on the correctness and security of Arbitrum Nitro.

Finally, we reviewed the special ArbOS smart contract operations that allow privileged and unprivileged users to perform important tasks in the Arbitrum system. In particular, we analyzed how retryable tickets are redeemed and canceled and how they are removed when they expire.

AnyTrust

AnyTrust is a variant of Arbitrum Nitro in which data is stored by a Data Availability Committee (DAC) and provided on demand rather than posted on L1 Ethereum as calldata. AnyTrust reduces transaction gas costs by making it possible to store data off-chain and assuming that at least two members of the DAC are honest and will make that data available to other parties. We performed a partial review of the differences between AnyTrust and Arbitrum Nitro, focusing on the L2 code's validation of Data Availability Certificates.

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:

- The validator code
- Cryptographic primitives
- The `zeroheavy` encoder / decoder
- The Brotli encoder / decoder
- The geth codebase (with the exception of specific changes made by Offchain Labs)
- The arbitrator and prover code
- The Data Availability Server software run by committee members in AnyTrust mode

Our review of the other smart contracts in the codebase covered only the bridge contracts (specifically the post-migration changes to their interactions with ArbOS).

Additionally, the codebase underwent several changes during the audit, which made the review more complex than expected.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	Used to detect common issues
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	Appendix E
Native Go fuzz	A smart fuzzer for Go code implemented in the standard toolchain as of Go 1.18	Appendix D

Test Results

We used Echidna to test the following cryptographic properties of the `HashProofHelper` component.

Property	Tool	Result
The `proveWithFullPreimage` and `proveWithSplitPreimage` functions never revert.	Echidna	**Passed**
When passed the same input, the `proveWithFullPreimage` and `proveWithSplitPreimage` functions compute the same `fullHash` value.	Echidna	**Passed**

When passed the same input, the `proveWithFullPreimage` and `proveWithSplitPreimage` functions compute the same part of the preimage, with a length of less than 32 bytes.	Echidna	**Passed**

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	Having conducted a manual review and an extensive fuzzing campaign, we believe that the arithmetic operations throughout the codebase are performed correctly; the sole exception is an operation that could cause an integer overflow in the validation of an AnyTrust Data Availability Certificate (TOB-ArbOS-18).	Satisfactory
Auditing	Certain critical administrative functions do not emit events for important state changes (TOB-ArbOS-12). This makes off-chain monitoring difficult to conduct. Additionally, Offchain Labs did not provide an incident response plan.	Moderate
Authentication / Access Controls	We did not identify any authentication or access control issues. The privileges granted to various actors and the limits on those privileges are generally well documented.	Satisfactory
Complexity Management	Most of the functions and contracts are organized and scoped appropriately and contain inline documentation that explains their workings. However, the state export code implemented to facilitate the migration process is complex with little inline documentation.	Satisfactory
Decentralization	The Arbitrum Nitro system is as highly centralized as Arbitrum Classic, as it relies on a small number of whitelisted validators and a sole chain owner. The Offchain Labs team should implement tests related to the ownership roles that cover both happy and unhappy paths. Additional documentation regarding deployment	Weak

	and ownership risks would also be beneficial.	
Documentation	The Nitro documentation provided by Offchain Labs is generally sufficient, though it lacks detail on recently implemented components such as the `HashProofHelper` and the state export / import code.	**Moderate**
Front-Running Resistance	Nitro relies on the `go-ethereum` codebase to process and execute EVM transactions and therefore suffers from the same lack of protections regarding transaction front-running.	**Moderate**
Low-Level Manipulation	All code reviewed during the engagement was high-level.	**Not Applicable**
Testing and Verification	The Offchain Labs team has implemented robust testing of most of the components, including fuzz testing based on advanced techniques such as differential fuzzing. However, the testing of recently implemented components such as the `HashProofHelper` and the export / import code should be expanded to cover expected and unexpected code paths.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Incorrect parsing of message data header	Data Validation	Medium
2	Incorrect gasLimit parsing	Data Validation	Informational
3	Extra computation associated with NUMBER and BLOCKHASH opcodes extends block-creation time	Configuration	High
4	Incorrect updates to the L2 pricing model	Undefined Behavior	High
5	Vulnerable package dependencies	Patching	Undetermined
6	L1 pricing model's susceptibility to manipulation	Undefined Behavior	Informational
7	Use of costly hash function with batched messages	Undefined Behavior	Low
8	Fragile batched message parsing	Undefined Behavior	Low
9	Insufficient testing of HashProofHelper and NitroMigrator	Testing	Low
10	Manual deployment process	Configuration	Low
11	Outdated package dependencies	Patching	Low
12	Lack of events for critical SequencerInbox operations	Auditing and Logging	Low

13	Incorrect migration of retryables	Undefined Behavior	**Medium**
14	Migration code does not scale to accommodate a large number of validators or outboxes	Data Validation	**Informational**
15	Serialization of large JSON integers could result in interoperability issues	Undefined Behavior	**Low**
16	Validators are not compensated for executing the migration	Undefined Behavior	**Low**
17	Risk of a node crash during parsing of DAS sequencer messages	Data Validation	**High**
18	Possible bypass of DACert expiration	Data Validation	**Informational**

Detailed Findings

1. Incorrect parsing of message data header	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ArbOS-1
Target: `arbstate/inbox.go`	

Description
ArbOS parses and validates L2 messages, including compressed messages. The first segment of a compressed message represents the kind of the message, while the rest contain the message. However, ArbOS parses the first segment incorrectly, which could cause a panic that halts the inbox's processing.

```
func (r *inboxMultiplexer) getNextMsg() (*MessageWithMetadata, error) {
	...
	kind := segment[0]
	segment = segment[1:]
	var msg *MessageWithMetadata
	if kind == BatchSegmentKindL2Message || kind ==
BatchSegmentKindL2MessageBrotli {

		if kind == BatchSegmentKindL2MessageBrotli {
			decompressed, err := arbcompress.Decompress(segment[1:],
arbos.MaxL2MessageSize)
			...
		}
```

Figure 1.1: Part of the `getNextMsg` function in `arbstate/inbox.go`

Specifically, ArbOS discards the first segment twice. If there are too few remaining segments, the code will panic.

Exploit Scenario
Eve sends a malformed compressed message, which causes a panic in ArbOS.

Recommendations
Short term, properly validate the length of segments and the content of inbox messages.

Long term, use fuzzing to detect input validation issues early in the development process.

2. Incorrect gasLimit parsing	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ArbOS-2
Target: `arbos/incomingmessage.go`	

Description

When ArbOS parses incoming L1 messages of the `UnsignedTx` type, it validates the `gasLimit` value incorrectly.

The `parseUnsignedTx` function, which parses incoming values, casts the `gasLimit` to a `uint64` value; however, if the `gasLimit` value cannot be represented by a `uint64`, the parsing will result in undefined behavior.

```
func parseUnsignedTx(rd io.Reader, poster common.Address, requestId *common.Hash, chainId
*big.Int, txKind byte) (*types.Transaction, error) {
	gasLimit, err := util.HashFromReader(rd)
	if err != nil {
		return nil, err
	}
	...
	switch txKind {
	case L2MessageKind_UnsignedUserTx:
		inner = &types.ArbitrumUnsignedTx{
			...
			Gas:       gasLimit.Big().Uint64(),
			...
		}
	case L2MessageKind_ContractTx:
		if requestId == nil {
			return nil, errors.New("cannot issue contract tx without L1
request id")
		}
		inner = &types.ArbitrumContractTx{
			...
			Gas:       gasLimit.Big().Uint64(),
			...
		}
	...
```

Figure 2.1: Part of the `parseUnsignedTx` function in `arbos/incomingmessage.go`

Exploit Scenario

Alice, a user, wants to run an L2 transaction with all of the available gas. Thus, she sets the `gasLimit` value to `type(uint256).max`, which results in undefined behavior when the transaction is parsed.

Recommendations

Short term, have the `parseUnsignedTx` function validate that the `gasLimit` value can fit into a `uint64` and return an error if it cannot.

Long term, implement validation of every value cast from one type to another to ensure that it is within the range of acceptable values for the destination type.

3. Extra computation associated with NUMBER and BLOCKHASH opcodes extends block-creation time	
Severity: **High**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-ArbOS-3
Target: `arbos/tx_processor.go`	

Description

Arbitrum is instrumented with a small number of EVM opcodes, including `NUMBER` and `BLOCKHASH`. The `L1BlockHash` and `L1BlockNumber` functions retrieve the `NextBlockNumber` and `BlockHash` values used by the opcodes from the ArbOS state. This process takes longer in the Arbitrum system than it does in the original geth implementation. As a result, block creation can take up to several seconds.

```
func (p *TxProcessor) L1BlockNumber(blockCtx vm.BlockContext) (uint64, error) {
	tracingInfo := util.NewTracingInfo(p.evm, p.msg.From(), arbosAddress,
util.TracingDuringEVM)
	state, err := arbosState.OpenSystemArbosState(p.evm.StateDB, tracingInfo,
false)
	if err != nil {
		return 0, err
	}
	return state.Blockhashes().NextBlockNumber()
}

func (p *TxProcessor) L1BlockHash(blockCtx vm.BlockContext, l1BlockNumber uint64)
(common.Hash, error) {
	tracingInfo := util.NewTracingInfo(p.evm, p.msg.From(), arbosAddress,
util.TracingDuringEVM)
	state, err := arbosState.OpenSystemArbosState(p.evm.StateDB, tracingInfo,
false)
	if err != nil {
		return common.Hash{}, err
	}
	return state.Blockhashes().BlockHash(l1BlockNumber)
}
```

Figure 3.1: The `L1BlockHash` and `L1BlockNumber` functions in `arbos/incomingmessage.go`

However, the gas costs of these opcodes have not been changed, which means that they are underpriced.

Exploit Scenario

Eve runs a transaction that executes the `BLOCKHASH` and `NUMBER` opcodes numerous times, with the goal of causing a denial of service. Her transaction adds several seconds to the block-production process and thus degrades the system's performance, which affects other users' experience.

Recommendations

Short term, save a local copy of the `NextBlockNumber` and `BlockHash` values to avoid needing to retrieve the variables directly from the ArbOS state.

Long term, implement fuzz tests throughout the codebase to ensure that block creation takes a reasonable amount of time.

4. Incorrect updates to the L2 pricing model	
Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ArbOS-4
Target: `arbos/internal_tx.go`	

Description

One of the calculations performed during the unpacking of internal transaction data is incorrect. The result of this calculation directly affects the L2 pricing model, which means that the pricing model is also incorrect.

The `InternalTxStartBlock` function uses the `util.PackInternalTxDataStartBlock` function, passing in the L1 base fee and block number, the L2 block number, and the amount of time that has passed since the last block (`timePassed`).

```
func InternalTxStartBlock(
	chainId,
	l1BaseFee *big.Int,
	l1BlockNum uint64,
	header,
	lastHeader *types.Header,
) *types.ArbitrumInternalTx {

	l2BlockNum := header.Number.Uint64()
	timePassed := header.Time - lastHeader.Time

	if l1BaseFee == nil {
		l1BaseFee = big.NewInt(0)
	}
	data, err := util.PackInternalTxDataStartBlock(l1BaseFee, l1BlockNum, 
l2BlockNum, timePassed)
	if err != nil {
		panic(fmt.Sprintf("Failed to pack internal tx %v", err))
	}
	return &types.ArbitrumInternalTx{
		ChainId: chainId,
		Data:    data,
	}
}
```

Figure 4.1: The `InternalTxStartBlock` function in `arbos/internal_tx.go`

The data is then unpacked in the `ApplyInternalTxUpdate` function:

```
func ApplyInternalTxUpdate(tx *types.ArbitrumInternalTx, state
*arbosState.ArbosState, evm *vm.EVM) {
	switch *(*[4]byte)(tx.Data[:4]) {
	case InternalTxStartBlockMethodID:
		inputs, err := util.UnpackInternalTxDataStartBlock(tx.Data)
		if err != nil {
			panic(err)
		}
		l1BlockNumber, _ := inputs[1].(uint64) // current block's
		timePassed, _ := inputs[2].(uint64)    // since last block

		…
```

Figure 4.2: The header of the `ApplyInternalTxUpdate` function in `arbos/internal_tx.go`

However, the `ApplyInternalTxUpdate` function assumes that `timePassed` is the third argument passed to `util.PackInternalTxDataStartBlock`, when it is actually the fourth.

Exploit Scenario

A new block is created, and the `ApplyInternalTxUpdate` function is executed. The incorrect `timePassed` value calculated by the function is then used in the creation of new L2 blocks, directly affecting the L2 pricing state (and thus the pricing model).

Recommendations

Short term, adjust the `ApplyInternalTxUpdate` function to use the correct index, `input[3]`, for the `timePassed` variable.

Long term, implement unit and fuzz tests throughout the codebase to ensure that its functions use the expected arguments.

5. Vulnerable package dependencies	
Severity: **Undetermined**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-ArbOS-5
Target: Throughout the codebase	

Description

Although dependency scans did not yield a direct threat to the project under review, `go list -json -m all | nancy sleuth` identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the repositories under review. The output below details these issues.

CVE ID	Description	Dependency
sonatype-2021-3619	Integer Overflow or Wraparound	`github.com/hashicorp/vault`
sonatype-2019-0772	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	`github.com/influxdata/influxdb`
CVE-2022-21698	Uncontrolled Resource Consumption ('Resource Exhaustion')	`github.com/prometheus/client_golang`

Figure 5.1: Advisories affecting Arbitrum dependencies

Additionally, `yarn audit` identified vulnerabilities affecting dependencies of the smart contracts:

CVE ID	Description	Dependency
CVE-2021-23358	Arbitrary Code Execution in underscore	`ethereum-waffle`
CVE-2022-0235	node-fetch is vulnerable to Exposure of Sensitive Information to an Unauthorized Actor	`node-fetch`
CVE-2022-31172	OpenZeppelin Contracts's [*sic*] SignatureChecker may revert on invalid EIP-1271 signers	`@openzeppelin/contracts`, `@openzeppelin/contracts-upgradeable`
CVE-2021-43138	Prototype pollution in async	`async`

Figure 5.2: Advisories affecting Arbitrum contract dependencies

Exploit Scenario

Alice installs the dependencies of an in-scope repository on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice subsequently uses the dependency, disclosing sensitive information to an unknown actor.

Recommendations

Short term, ensure dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, consider integrating automated dependency auditing into the development workflow. If a dependency cannot be updated when a vulnerability is disclosed, ensure the code does not use and is not affected by the vulnerable functionality of the dependency.

6. L1 pricing model's susceptibility to manipulation

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ArbOS-6
Target: `arbos/l1pricing/l1pricing.go`	

Description

The L1 pricing model relies on an account balance that is susceptible to manipulation.

When computing L1 prices, the `UpdateForBatchPosterSpending` function reads the balance of `L1PricerFundsPoolAddress` and computes the amount of funds to be distributed to batch posters:

```
func (ps *L1PricingState) UpdateForBatchPosterSpending(
	statedb vm.StateDB,
	evm *vm.EVM,
	arbosVersion uint64,
	updateTime, currentTime uint64,
	batchPoster common.Address,
	weiSpent *big.Int,
	scenario util.TracingScenario,
) error {
	...
	oldSurplus := am.BigSub(statedb.GetBalance(L1PricerFundsPoolAddress),
am.BigAdd(totalFundsDue, fundsDueForRewards))

	...

	// allocate funds to this update
	collectedSinceUpdate := statedb.GetBalance(L1PricerFundsPoolAddress)
	availableFunds := am.BigDivByUint(am.BigMulByUint(collectedSinceUpdate,
allocationNumerator), allocationDenominator)

	...
	err = util.TransferBalance(
		&L1PricerFundsPoolAddress, &payRewardsTo, paymentForRewards, evm,
scenario, "batchPosterReward",
	)
	...
	for _, posterAddr := range allPosterAddrs {
		...
			err = util.TransferBalance(
				&L1PricerFundsPoolAddress, &addrToPay, balanceToTransfer,
evm, scenario, "batchPosterRefund",
```

```
                              )
                    ...
}
```

Figure 6.1: Part of the `UpdateForBatchPosterSpending` function in `arbos/l1pricing/l1pricing.go`

The base fee amount depends on the balance of the `L1PricerFundsPoolAddress` account. Thus, by donating ether to that account, a user could manipulate the base fee amount. Moreover, there are two transfers of rewards from the `L1PricerFundsPoolAddress` account before the check of its balance and the computation of the values that determine L1 prices. In theory, the destination of these reward transfers could be the `L1PricerFundsPoolAddress` itself. However, self-transfers would not change the balance of `L1PricerFundsPoolAddress`, and the result of the price computation would be incorrect

Exploit Scenario

Eve repeatedly transfers ether to the address of the `L1PricerFundsPoolAddress` account to secretly manipulate the L1 fee amount. She then stops making these transfers, triggering an increase in the overall fee amount. In doing so, Eve manipulates other users of the platform, as the fee increase discourages other users from interacting with it.

Recommendations

Short term, ensure that users are aware of the pricing model's susceptibility to manipulation. Additionally, to prevent incorrect price computations, disallow self-transfers to `L1PricerFundsPoolAddress`.

Long term, implement unit and fuzz tests throughout the codebase to ensure that the pricing model behaves as expected.

7. Use of costly hash function with batched messages	
Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ArbOS-7
Target: `inbox.sol`, `arbos/incomingmessage.go`	

Description

The use of batched messages requires the use of `keccak256`, which is costly and can thus cause a denial of service.

Any user can submit L2 messages through the `inbox.sol` smart contract. To do so, the user pays only the cost of submitting the data on-chain and executing `keccak256` to hash the data.

```
function _deliverMessage(
    uint8 _kind,
    address _sender,
    bytes memory _messageData
) internal returns (uint256) {
    if (_messageData.length > MAX_DATA_SIZE)
        revert DataTooLarge(_messageData.length, MAX_DATA_SIZE);
    uint256 msgNum = deliverToBridge(_kind, _sender, keccak256(_messageData));
    emit InboxMessageDelivered(msgNum, _messageData);
    return msgNum;
}
```

Figure 7.1: The `_deliverMessage` function in `inbox.sol`

Once an L2 message is on the chain, ArbOS parses and processes it. Figure 7.2 shows the parsing of batched L2 messages, which can require a significant amount of computation:

```
func parseL2Message(rd io.Reader, poster common.Address, requestId *common.Hash,
chainId *big.Int, depth int) (types.Transactions, error) {
        ...
        case L2MessageKind_Batch:
                if depth >= 16 {
                        return nil, errors.New("L2 message batches have a max depth
of 16")
                }
                segments := make(types.Transactions, 0)
                index := big.NewInt(0)
                for {
                        nextMsg, err := util.BytestringFromReader(rd,
```

```
MaxL2MessageSize)
                        if err != nil {
                                // an error here means there are no further messages
in the batch
                                // nolint:nilerr
                                return segments, nil
                        }

                        var nextRequestId *common.Hash
                        if requestId != nil {
                                subRequestId := crypto.Keccak256Hash(requestId[:],
math.U256Bytes(index))
                                nextRequestId = &subRequestId
                        }
                        nestedSegments, err :=
parseL2Message(bytes.NewReader(nextMsg), poster, nextRequestId, chainId, depth+1)
                        if err != nil {
                                return nil, err
                        }
                        segments = append(segments, nestedSegments...)
                        index.Add(index, big.NewInt(1))
                }
```

Figure 7.2: Part of the `parseL2Message` function in `arbos/incomingmessage.go`

Specifically, the parsing of each message in a batch of messages involves a call to `keccak256` from the geth code. The gas cost of the on-chain use of `keccak256` (which is paid by the user) is computed only once, so the user incurs a total cost of 30 units of gas, plus 6 units of gas for each word of input data (rounded up). However, when parsing batched messages, ArbOS incurs a cost of ~36 units of gas for each `keccak256` call. Thus, for the cost of only a single `keccak256` transaction, an attacker could cause ArbOS to incur such a high cost that it experienced a denial of service.

Exploit Scenario

Eve crafts a series of large L2 messages that includes several batched messages. Each time a message in the batch is parsed, there is a call to `keccak256`. The cost of these calls is paid by validators, degrading their performance.

Recommendations

Short term, ensure that the cost of submitting batched messages is commensurate with the amount of work involved in parsing them.

Long term, review the use of costly operations such as `keccak256` calls to identify any denial-of-service attack vectors.

8. Fragile batched message parsing

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ArbOS-8
Target: `go-ethereum/core/state_transition.go`	

Description

Each transaction in a batch of messages has a corresponding nonce. The transactions' nonces must be in consecutive order, and if one of the nonces is flagged as invalid, all of the transactions will fail.

Figure 8.1 shows the parsing of batched L2 messages:

```
func parseL2Message(rd io.Reader, poster common.Address, requestId *common.Hash,
chainId *big.Int, depth int) (types.Transactions, error) {
	...
	case L2MessageKind_Batch:
		if depth >= 16 {
			return nil, errors.New("L2 message batches have a max depth
of 16")
		}
		segments := make(types.Transactions, 0)
		index := big.NewInt(0)
		for {
			nextMsg, err := util.BytestringFromReader(rd,
MaxL2MessageSize)
			if err != nil {
				// an error here means there are no further messages
in the batch
				// nolint:nilerr
				return segments, nil
			}

			var nextRequestId *common.Hash
			if requestId != nil {
				subRequestId := crypto.Keccak256Hash(requestId[:],
math.U256Bytes(index))
				nextRequestId = &subRequestId
			}
			nestedSegments, err :=
parseL2Message(bytes.NewReader(nextMsg), poster, nextRequestId, chainId, depth+1)
			if err != nil {
				return nil, err
			}
```

```
                        segments = append(segments, nestedSegments...)
                        index.Add(index, big.NewInt(1))
                }
```

Figure 8.1: Part of the parseL2Message function in arbos/incomingmessage.go

Because the transactions are executed by the geth code, they are validated through all of the standard Ethereum transaction checks. If one of the nonces is incorrect, the corresponding transaction will immediately be rejected.

```
func (st *StateTransition) preCheck() error {
        // Only check transactions that are not fake
        if !st.msg.IsFake() {
                // Make sure this transaction's nonce is correct.
                stNonce := st.state.GetNonce(st.msg.From())
                if msgNonce := st.msg.Nonce(); stNonce < msgNonce {
                        return fmt.Errorf("%w: address %v, tx: %d state: %d",
ErrNonceTooHigh,
                                st.msg.From().Hex(), msgNonce, stNonce)
                } else if stNonce > msgNonce {
                        return fmt.Errorf("%w: address %v, tx: %d state: %d",
ErrNonceTooLow,
                                st.msg.From().Hex(), msgNonce, stNonce)
                } else if stNonce+1 < stNonce {
                        return fmt.Errorf("%w: address %v, nonce: %d", ErrNonceMax,
                                st.msg.From().Hex(), stNonce)
                }
```

Figure 8.2: The header of the preCheck function in go-ethereum/core/state_transition.go

After the execution of `preCheck`, the code checks other important values of each message, including the transaction gas limit, as well as the balance of the sender's account.

However, in Arbitrum, there is no notion of a mempool. Thus, if the transactions in a batch are not in the correct order, or if any transaction is rejected by the `StateTransition` function, the nonces of the remaining transactions will be incorrect, and they will all be rejected.

Exploit Scenario

Alice submits a number of transactions in a batch of messages. The `StateTransition` function finds one of the first transactions to be invalid, so the remaining transactions are dropped.

Recommendations

Short term, document the behavior surrounding batched messages to ensure that users are aware of it.

Long term, review the impacts of the geth design decisions on the Arbitrum system and ensure that they do not negatively affect the user experience.

9. Insufficient testing of HashProofHelper and NitroMigrator	
Severity: **Low**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-ArbOS-9
Target: `contracts/src/osp/HashProofHelper.sol`, `arb-bridge-eth/contracts/bridge/NitroMigrator.sol`	

Description

The `HashProofHelper` and `NitroMigrator` contracts lack sufficient testing. Robust unit and integration tests are critical to the detection of bugs and logic errors early in the development process.

The `HashProofHelper` contract's tests currently check only the production of valid proofs from full preimages and split preimages. As a result, they lack thorough coverage of cases in which input is malformed (i.e., "unhappy" paths). Thorough test coverage would increase users' and developers' confidence in the functionality of the code.

Additionally, randomized testing of proof input is repeated only 16 times, which may be insufficient to detect corner cases. The `HashProofHelper` contract also contains numerous cryptographic primitives that require more in-depth analysis.

Similarly, the `NitroMigrator` tests do not adequately cover the contract's functionality. Its tests check the postconditions of only transactions that do not revert and lack coverage of state changes.

Exploit Scenario

The `HashProofHelper` contract is called to split up a proof as part of the one-step-proof flow. Eve, an attacker, identifies an execution path that has not been tested and exploits it to cause undefined behavior in the system.

Recommendations

Short term, expand the codebase's unit and integration test coverage to include all happy and unhappy paths.

Long term, integrate unit and integration tests into the CI / CD pipeline, and integrate automated testing techniques such as fuzzing and symbolic execution into the codebase.

10. Manual deployment process	
Severity: **Low**	Difficulty: **Medium**
Type: Configuration	Finding ID: TOB-ArbOS-10
Target: Migration process	

Description

The migration-related contracts are deployed from an externally owned account (EOA). The use of a manual upgrade process increases the risk of human error and typos.

The `ProxyAdmin`, `TransparentUpgradeableProxy`, and `NitroMigrator` contracts are deployed through an EOA, after which ownership of the `ProxyAdmin` and `TransparentUpgradeableProxy` is transferred to an Offchain Labs–controlled Gnosis Safe multisig.

Exploit Scenario

Alice deploys the `ProxyAdmin`, `TransparentUpgradeableProxy`, and `NitroMigrator` contracts. When transferring ownership of the `ProxyAdmin` and `TransparentUpgradeableProxy` contracts to the Gnosis Safe multisig, she mistypes the address of the multisig wallet. As a result, the contracts can be controlled only by the owner of that incorrect address and must be redeployed.

Recommendations

Short term, use the Gnosis Safe multicall function to deploy these contracts. That way, the multisig will have ownership of the contracts from their deployment.

Long term, use automated mechanisms such as deployment scripts, smart contract factory patterns, and multicalls to reduce the risk of errors in the deployment process.

11. Outdated package dependencies

Severity: **Low**	Difficulty: **High**
Type: Patching	Finding ID: TOB-ArbOS-11
Target: `arbitrum/packages/arb-bridge-eth`	

Description

The system's use of outdated dependencies may cause unexpected behavior.

For example, `npm`, which can check a repository for outdated package versions, found that the Arbitrum Classic repository uses an outdated OpenZeppelin package. Because of a vulnerability in this package, it may be possible for the `initialize()` function to be invoked twice.

```
Affected versions of this package are vulnerable to Deserialization of Untrusted
Data. It is possible for initializer() protected functions to be executed twice, if
this happens in the same transaction. For this to happen, either one call has to be
a subcall to the other, or both calls have to be subcalls of a common initializer()
protected function. This can be particularly dangerous if the initialization is not
part of the proxy construction, and reentrancy is possible by executing an external
call to an untrusted address.
```

Figure 11.1: An explanation of the deserialization of untrusted data vulnerability in the OpenZeppelin package

The `go list -u -m -json all | go-mod-outdated -style markdown` command can be used to identify outdated dependencies in the Arbitrum Nitro version of ArbOS.

Exploit Scenario

Alice, an Arbitrum developer, refactors the contracts such that they can be initialized in two separate transactions through a deployment script rather than through a proxy. Because of Arbitrum's use of a vulnerable and outdated OpenZeppelin package, the `initialize()` function can be invoked twice, causing unexpected behavior.

Recommendations

Short term, upgrade to newer versions of the system's outdated dependencies. If versioning constraints prevent updates to any vulnerable package, document the vulnerability and ensure that it will not become exploitable as new code is introduced.

Long term, integrate Dependabot or dependency checks into the CI pipeline and use the latest versions of packages whenever possible.

12. Lack of events for critical SequencerInbox operations

Severity: **Low**	Difficulty: **Medium**
Type: Auditing and Logging	Finding ID: TOB-ArbOS-12
Target: `arb-bridge-eth/contracts/bridge/SequencerInbox.sol`	

Description

The two Arbitrum Nitro–specific `SequencerInbox` functions do not emit events for critical operations. A lack of events makes it difficult to review the correct behavior and state of a contract once it has been deployed.

If the sequencer called both functions in a single transaction, the calls would not be reflected in the event logs:

```
/// @dev this function is intended to force include the delayed inbox a final time
in the nitro migration
function shutdownForNitro(uint256 _totalDelayedMessagesRead, bytes32 delayedAcc)
    external
    whenNotShutdownForNitro
{
    // no delay on force inclusion, triggered only by rollup's owner
    require(Rollup(payable(rollup)).owner() == msg.sender, "ONLY_ROLLUP_OWNER");

    // if _totalDelayedMessagesRead == totalDelayedMessagesRead, we don't need to
force include
    // if _totalDelayedMessagesRead < totalDelayedMessagesRead we are trying to read
backwards and will revert in forceInclusionImpl
    // if _totalDelayedMessagesRead > totalDelayedMessagesRead we will force include
the new delayed messages into the seqInbox
    if (_totalDelayedMessagesRead != totalDelayedMessagesRead) {
        forceInclusionImpl(_totalDelayedMessagesRead, delayedAcc);
    }

    isShutdownForNitro = true;
}

function undoShutdownForNitro() external {
    require(Rollup(payable(rollup)).owner() == msg.sender, "ONLY_ROLLUP_OWNER");
    require(isShutdownForNitro, "NOT_SHUTDOWN");
    isShutdownForNitro = false;
}
```

Figure 12.1: The `shutdownForNitro` and `undoShutdownForNitro` functions in `Portal.sol#L460–469`

Without events, users and blockchain-monitoring systems cannot easily detect suspicious behavior.

Exploit Scenario

Eve, an attacker, is able to take ownership of the `SequencerInbox` contract. She calls the `shutdownForNitro` function to shut the inbox down for the Nitro upgrade. While the system is in “Nitro mode,” she abuses sequencers by forcing messages into the inbox without any delay. Eve then calls `undoShutdownForNitro` to revert the contract to its pre-shutdown state.

Recommendations

Short term, add events for all critical operations that result in state changes. Events aid in contract monitoring and the detection of suspicious behavior.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

13. Incorrect migration of retryables	
Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ArbOS-13
Target: `arbos/arbosState/initialize.go`	

Description

The migration from Arbitrum Classic to Nitro is meant to preserve the state of the blockchain. However, the migration could cause users to lose the funds sent through a retryable.

The `initializeRetryables` function loops through the retryables exported from Arbitrum Classic. If the `Timeout` value of a retryable is less than the `currentTimestamp` (i.e., the retryable is no longer redeemable and must be deleted), the function will skip over it without sending the `Callvalue` to the `Beneficiary` address specified in the retryable. As a result, the ether sent by the user will be lost.

```
func initializeRetryables(statedb *state.StateDB, rs *retryables.RetryableState,
initData statetransfer.RetryableDataReader, currentTimestamp uint64) error {
	var retryablesList []*statetransfer.InitializationDataForRetryable
	for initData.More() {
		r, err := initData.GetNext()
		if err != nil {
			return err
		}
		if r.Timeout <= currentTimestamp {
			continue
		}
		retryablesList = append(retryablesList, r)
	}
	sort.Slice(retryablesList, func(i, j int) bool {
		a := retryablesList[i]
		b := retryablesList[j]
		if a.Timeout == b.Timeout {
			return arbmath.BigLessThan(a.Id.Big(), b.Id.Big())
		}
		return a.Timeout < b.Timeout
	})
	for _, r := range retryablesList {
		var to *common.Address
		if r.To != (common.Address{}) {
			to = &r.To
		}
		statedb.AddBalance(retryables.RetryableEscrowAddress(r.Id),
```

```
r.Callvalue)
		_, err := rs.CreateRetryable(r.Id, r.Timeout, r.From, to, r.Callvalue,
r.Beneficiary, r.Calldata)
		if err != nil {
			return err
		}
	}
	return initData.Close()
}
```

Figure 13.1: The `initializeRetryables` function in `initialize.go#L167–199`

Exploit Scenario

Bob, a user, creates a retryable with a `Callvalue` of 1 ETH. He intends to execute the retryable in the future, as he knows that if it expires, the `Callvalue` will be sent back to the specified `Beneficiary` address. However, after the migration, Bob notices that his retryable has expired and that the `Callvalue` has not been refunded, leaving him with a loss of 1 ETH.

Recommendations

Short term, have ArbOS send the `Callvalue` of any retryable that expires during the migration to the specified `Beneficiary` address.

Long term, expand the unit and fuzz tests to cover any migration-related edge cases that could lead to an undefined state.

14. Migration code does not scale to accommodate a large number of validators or outboxes	
Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ArbOS-14
Target: `NitroMigrator.sol`, `RollupAdmin.sol`	

Description

If the number of validators or outboxes in the system is too large, the smart contract code that handles the migration process will fail to execute, blocking the migration.

As part of the migration from Arbitrum Classic to Arbitrum Nitro, smart contract code calls specific functions in each Arbitrum component. The third step in the migration process involves iterating through a list of all outboxes and reconfiguring each one to use the new bridge.

```
function nitroStep3(
    uint64 nitroGenesisBlockNumber,
    bytes32 nitroGenesisHash,
    bool skipCheck
) external onlyOwner {
    …
    uint256 numOutboxes = bridge.allowedOutboxListLength();
    for (uint256 i = 0; i < numOutboxes; i++) {
        // when we disable the list, it always shrinks by 1, so first index should
always be a new one
        address currOutbox = bridge.allowedOutboxList(0);
        IOutbox(currOutbox).setBridge(IBridge(address(nitroBridge)));
        bridge.setOutbox(currOutbox, false);
        nitroBridge.setOutbox(currOutbox, true);
    }
```

Figure 14.1: Part of the `nitroStep3` function in `NitroMigrator.sol`

Additionally, the `shutdownForNitro` function is called to shut down components such as the `Rollup` contract.

```
function shutdownForNitro(
    uint256 finalNodeNum,
    bool destroyAlternatives,
    bool destroyChallenges
) external whenNotPaused {
    …
```

```
    // we separate the loop that gets staker addresses to be different from the loop
that withdraw stakers
    // since withdrawing stakers has side-effects on the array that is queried in
`getStakerAddress`.
    for (uint64 i = 0; i < stakerCount; ++i) {
        stakerAddresses[i] = getStakerAddress(i);
    }

    for (uint64 i = 0; i < stakerCount; ++i) {
        address stakerAddr = stakerAddresses[i];
        address chall = currentChallenge(stakerAddr);

        if (chall != address(0)) {
            require(destroyChallenges, "CHALLENGE_NOT_EXPECTED");
            address asserter = IChallenge(chall).asserter();
            address challenger = IChallenge(chall).challenger();

            clearChallenge(asserter);
            clearChallenge(challenger);

            IChallenge(chall).clearChallenge();
            emit ChallengeDestroyedInMigration(chall);
        }

        if (getNode(latestStakedNode(stakerAddr)) == INode(0)) {
            // this node got destroyed, so we force refund the staker
            withdrawStaker(stakerAddr);
            emit StakerWithdrawnInMigration(stakerAddr);
        }
        // else the staker can unstake and withdraw regularly using
`returnOldDeposit`
    }

    shutdownForNitroBlock = block.number;
    _pause();
    emit OwnerFunctionCalled(25);
}
```

Figure 14.2: Part of the `shutdownForNitro` function in `RollupAdmin.sol`

Both the `nitroStep3` function and the `shutdownForNitro` function must iterate over a number of elements. If the number of elements (i.e., the number of outboxes or validators) is very large, the functions may experience an out-of-gas exception and revert.

Exploit Scenario

Numerous validators are added to the Arbitrum system before the migration. This causes the `shutdownForNitro` transaction to exceed the per-block gas limit, preventing the execution of the migration code. As a result, the migration code must be changed or the number of validators, reduced.

Recommendations

Short term, document the expected number of validators and outboxes that can exist in the system without causing an out-of-gas exception during the migration process.

Long term, identify and evaluate all implicit or explicit loops in the smart contract code to ensure that their execution will not trigger an out-of-gas exception.

15. Serialization of large JSON integers could result in interoperability issues

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ArbOS-15
Target: `datadump.cpp`	

Description

Arbitrum Nitro's method of serializing JSON values exported by validators can differ from that of mainstream implementations such as `NodeJS` and `jq`.

The JSON standard warns about certain "interoperability problems" in numeric types outside the range `[-(2`^{53}`)+1, (2`^{53}`)-1]`. These issues are caused by widely used JSON implementations that use `IEEE 754` (double-precision) numbers to implement integers.

For instance, if a validator saved the nonce value `"1152921504606846976"` (2^{60}), it would be serialized as the expected value `1152921504606846976`. However, web browsers, `NodeJS`, and `jq 1.5` would parse it as `1152921504606847000`.

```
[
  {
    ...
    "nonce": 1152921504606846976,
    ...
  }
]
```

Figure 15.1: An example of part of a JSON file

This parsing affects `uint64` fields such as the `nonce` and `Timeout` fields, the values of which are saved and parsed directly from JSON numbers, without the use of strings:

```
nlohmann::json serializeRetryable(ValueLoader loader, Value, Value retryable) {
    nlohmann::json json;
    auto tup = resolveTuple(loader, retryable);
    json["Id"] = hashString(indexInt(tup, 0));
    json["From"] = addressString(indexInt(tup, 1));
    json["To"] = addressString(indexInt(tup, 2));
    json["Callvalue"] = intx::to_string(indexInt(tup, 3));
    json["Beneficiary"] = addressString(indexInt(tup, 5));
    json["Calldata"] = serializeBytes(loader, indexTup(loader, tup, 6));
    auto rem = indexTup(loader, tup, 7);
    json["Timeout"] = uint64_t(indexInt(rem, 0));
    return json;
```

```
}
```

Figure 15.2: The `serializeRetryable` function

Exploit Scenario

When migrating from Arbitrum Classic to Arbitrum Nitro, Alice exports data generated by her node. This data contains a value of the `uint64` type, which is not supported by the JSON standard. As a result, the Nitro chain is initialized incorrectly, causing some accounts and retryable tickets to use incorrect data.

Recommendations

Short term, use strings instead of JSON numeric values to implement the `uint64` fields. This will prevent any ambiguity when parsing the numeric fields of JSON files.

Long term, review the standards regarding the data exported and imported by validators to identify any sources of ambiguity.

References

- RFC 8259, Section 6 “Numbers”

16. Validators are not compensated for executing the migration	
Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ArbOS-16
Target: Migration process	

Description

When migrating from Arbitrum Classic to Arbitrum Nitro, validators must export the state of Arbitrum Classic and import it into Arbitrum Nitro. However, they are not compensated for this work.

There are currently around 1 million accounts in Arbitrum Classic. These accounts must be migrated to Arbitrum Nitro one by one, through an iterative process that could cause each validator to incur a significant computational cost.

Exploit Scenario

Eve, an attacker, deploys a contract that sends 1 wei to a pseudorandom set of addresses, creating new state data that will need to be migrated. In this way, Eve forces a validator to migrate a large amount of additional data without spending much herself.

Recommendations

Short term, document the fact that validators are not compensated for executing the migration.

Long term, review the costs and incentives associated with the recurrent and exceptional processes that are executed in Arbitrum.

17. Risk of a node crash during parsing of DAS sequencer messages

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ArbOS-17
Target: `arbstate/inbox.go`	

Description

A specially crafted AnyTrust sequencer message could cause a validator crash.

As part of Arbitrum Nitro's AnyTrust mode, an external Data Availability Committee (DAC) stores data on Data Availability Servers (DAS) and provides a Data Availability Certificate (DACert). The L2 code then parses and checks the validity of the DACert in the inbox, which is posted on L1 Ethereum in place of standard calldata.

```
func RecoverPayloadFromDasBatch(
	ctx context.Context,
	batchNum uint64,
	sequencerMsg []byte,
	dasReader DataAvailabilityReader,
	preimages map[common.Hash][]byte,
	keysetValidationMode KeysetValidationMode,
) ([]byte, error) {
	cert, err := DeserializeDASCertFrom(bytes.NewReader(sequencerMsg[40:]))
	if err != nil {
		log.Error("Failed to deserialize DAS message", "err", err)
		return nil, nil
	}
	version := cert.Version
	[...]
	getByHash := func(ctx context.Context, hash common.Hash) ([]byte, error) {
		newHash := hash
		if version == 0 {
			newHash = dastree.FlatHashToTreeHash(hash)
		}

		preimage, err := dasReader.GetByHash(ctx, newHash)
		if err != nil && hash != newHash {
			log.Debug("error fetching new style hash, trying old", "new",
newHash, "old", hash, "err", err)
			preimage, err = dasReader.GetByHash(ctx, hash)
		}
		if err != nil {
			return nil, err
		}
```

```
            switch {
            case version == 0 && crypto.Keccak256Hash(preimage) != hash:
                fallthrough
            case version == 1 && dastree.Hash(preimage) != hash:
                log.Error(
                    "preimage mismatch for hash",
                    "hash", hash, "err", ErrHashMismatch, "version", version,
                )
                return nil, ErrHashMismatch
            case version >= 2:
                log.Error(
                    "Committee signed unsuported certificate format",
                    "version", version, "hash", hash, "payload", preimage,
                )
                panic("node software out of date")
            }
            return preimage, nil
        }
    [...]
```

Figure 17.1: Part of the `RecoverPayloadFromDasBatch` function in `inbox.go`

However, if the version specified in the DACert is greater than or equal to 2, the validator that processes it will panic.

Exploit Scenario

Eve, a malicious party with control of the sequencer (or acting in collusion with it), posts a message that includes a DACert set to version 2. As a result, the validator node that processes the message crashes.

Recommendations

Short term, ensure that noncritical error conditions such as an invalid version number or the use of outdated software by a node are handled correctly.

Long term, review the use of panics and ensure that the system panics only when absolutely necessary—that is, when it has entered a critical state from which it cannot recover.

18. Possible bypass of DACert expiration	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ArbOS-18
Target: `arbstate/inbox.go`	

Description

An integer overflow during the validation of a DACert could lead to the acceptance of an expired DACert, meaning that committee members would not be required to provide the data associated with the certificate.

The `RecoverPayloadFromDasBatch` function deserializes DACerts from sequencer messages and then validates them.

```
func RecoverPayloadFromDasBatch(
	ctx context.Context,
	batchNum uint64,
	sequencerMsg []byte,
	dasReader DataAvailabilityReader,
	preimages map[common.Hash][]byte,
	keysetValidationMode KeysetValidationMode,
) ([]byte, error) {
   [...]
	maxTimestamp := binary.BigEndian.Uint64(sequencerMsg[8:16])
	if cert.Timeout < maxTimestamp+MinLifetimeSecondsForDataAvailabilityCert {
		log.Error("Data availability cert expires too soon", "err", "")
		return nil, nil
	}
   [...]
```

Figure 18.1: Part of the `RecoverPayloadFromDasBatch` function in `inbox.go`

However, the validation of a DACert's `Timeout` value can result in an integer overflow, as `MinLifetimeSecondsForDataAvailabilityCert` is a constant set to one week (in seconds) and the `maxTimestamp` value is controlled by the sequencer.

Exploit Scenario

Eve, a malicious party who controls a sequencer, obtains a signed batch of data that has already expired. After collecting enough signatures, she posts a DACert with a `maxTimestamp` set to a very large value. As a result, if there is a challenge, the data necessary for its resolution may not be available.

Recommendations

Short term, use saturation arithmetic in the addition of `maxTimestamp` and `MinLifetimeSecondsForDataAvailabilityCert`.

Long term, improve the suite of unit tests and integrate automated testing techniques such as fuzzing into the codebase.

Summary of Recommendations

Offchain Labs's Arbitrum codebase is a work in progress with multiple planned iterations. Trail of Bits recommends that Offchain Labs address the findings detailed in this report and take the following additional steps prior to deployment:

- **Enhance the suite of unit tests to ensure that the system behaves as expected when handling both happy and unhappy paths.** This will help identify problematic code and increase users' and developers' confidence in the code.
- **Integrate fuzz testing into the development process to detect potential panics.** See appendix D for recommendations on fuzzing ArbOS.
- **Integrate automated dependency auditing into the development workflow.** That way, if a dependency is found to be vulnerable, Offchain Labs will be made aware of the vulnerability and will be able to take the necessary steps to prevent its exploitation.
- **Review the geth architecture and ensure that its effects on the Arbitrum implementation are explicitly documented.** This will help ensure that users and developers are aware of the specifics of the Arbitrum ecosystem.
- **Favor the use of automated processes over manual processes.** The use of deployment scripts, contract factory patterns, and multicalls during the migration process will limit the chance of mistakes.
- **Execute static analysis tools as part of the development workflow to ensure that any issues are caught early in the process.** The use of these tools will also provide the team with additional real-time feedback on pull requests.
- **Clearly document the expectations surrounding the `HashProofHelper` contract and the data import / export code.** That way, the team will be able to cross-reference any unexpected behavior with the documentation. Clear developer documentation will also ensure that the process of cloning and setting up the repository is a smooth one.

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. Risks Associated with Malicious Sequencers

Offchain Labs intends to allow third parties to serve as sequencers. The introduction of third-party sequencers could introduce the following risks:

- **Malicious sequencers could control block numbers, `timePassed`, `batchTimestamp`, `batchPosterAddress`, and `batchDataGas` values, and the L1 base fee.**

```
func ApplyInternalTxUpdate(tx *types.ArbitrumInternalTx, state
*arbosState.ArbosState, evm *vm.EVM) {
	switch *(*[4]byte)(tx.Data[:4]) {
	case InternalTxStartBlockMethodID:
		inputs, err := util.UnpackInternalTxDataStartBlock(tx.Data)
		if err != nil {
			panic(err)
		}
		l1BlockNumber, _ := inputs[1].(uint64) // current block's
		timePassed, _ := inputs[2].(uint64)    // since last block

		nextL1BlockNumber, err := state.Blockhashes().NextBlockNumber()
		state.Restrict(err)

		l2BaseFee, err := state.L2PricingState().BaseFeeWei()
		state.Restrict(err)

		if l1BlockNumber >= nextL1BlockNumber {
			var prevHash common.Hash
			if evm.Context.BlockNumber.Sign() > 0 {
				prevHash =
evm.Context.GetHash(evm.Context.BlockNumber.Uint64() - 1)
			}

state.Restrict(state.Blockhashes().RecordNewL1Block(l1BlockNumber, prevHash))
		}

		currentTime := evm.Context.Time.Uint64()

		// Try to reap 2 retryables
		_ = state.RetryableState().TryToReapOneRetryable(currentTime, evm,
util.TracingDuringEVM)
		_ = state.RetryableState().TryToReapOneRetryable(currentTime, evm,
util.TracingDuringEVM)

		state.L2PricingState().UpdatePricingModel(l2BaseFee, timePassed, false)

		state.UpgradeArbosVersionIfNecessary(currentTime, evm.ChainConfig())
```

Figure C.1: Part of the `ApplyInternalTxUpdate` function in `arbos/internal_tx.go`

- **Sequencers could replay individual messages in a batch.** Replaying a transaction could enable a sequencer to change the structure of a batch.

D. Recommendations for Fuzzing ArbOS

Trail of Bits reviewed the fuzz tests for ArbOS provided by Offchain Labs, which cover state transitions and the precompiled contracts. We recommend that Offchain Labs take the following steps to further strengthen its suite of fuzz tests:

- Keep the tests up to date and run them at least once before every candidate or internal release or (if the code is not frozen) once every three days.
- Identify any roadblocks in the fuzzer's random exploration of data and develop workarounds for them. For instance, when exploring the `ArbRetryable` code, the fuzzer was unable to reach code that is executed upon the successful redemption of a retryable ticket. To overcome this, we added code that creates a valid ticket:

```
state, err := arbosState.OpenSystemArbosState(sdb, nil, false)
if err != nil {
        panic(err)
}
id := common.BytesToHash([]byte{0})
lastTimestamp := 1657530074

from := common.BytesToAddress([]byte{3, 4, 5})
to := common.BytesToAddress([]byte{6, 7, 8, 9})
timeout := lastTimestamp + 10000000000
callvalue := big.NewInt(0)
beneficiary := from
calldata := make([]byte, 42)
for i := range calldata {
        calldata[i] = byte(i + 3)
}
_, err = state.RetryableState().CreateRetryable(id, uint64(timeout + 10000000000),
from, &to, callvalue, beneficiary, calldata)
```

Figure D.1: Part of the fuzzing code that adds a new retryable ticket

- Enable the use of the DAS reader during the parsing of messages from the inbox.

```
type PreimageDASReader struct {
}

func (dasReader *PreimageDASReader) GetByHash(ctx context.Context, hash common.Hash)
([]byte, error) {
        return hash.Bytes(), nil
}

func (dasReader *PreimageDASReader) HealthCheck(ctx context.Context) error {
        return nil
}
```

```
func (dasReader *PreimageDASReader) ExpirationPolicy(ctx context.Context)
(arbstate.ExpirationPolicy, error) {
	return arbstate.DiscardImmediately, nil
}


func BuildBlock(
	statedb *state.StateDB,
	lastBlockHeader *types.Header,
	chainContext core.ChainContext,
	chainConfig *params.ChainConfig,
	inbox arbstate.InboxBackend,
	seqBatch []byte,
) (*types.Block, error) {
	var delayedMessagesRead uint64
	if lastBlockHeader != nil {
		delayedMessagesRead = lastBlockHeader.Nonce.Uint64()
	}
	var dasReader arbstate.DataAvailabilityReader
	dasReader = &PreimageDASReader{}
	inboxMultiplexer := arbstate.NewInboxMultiplexer(inbox, delayedMessagesRead,
dasReader, arbstate.KeysetDontValidate)
	...
```

Figure D.2: Part of the fuzzing code for the DACert validation process

- Implement a fuzzing mode in which the execution of unexpected code triggers a panic. For instance, we instrumented the `TransferBalance` function with the following code to detect suspicious uses of the function:

```
func TransferBalance(
	from, to *common.Address,
	amount *big.Int,
	evm *vm.EVM,
	scenario TracingScenario,
	purpose string,
) error {
	if arbmath.BigLessThan(amount, big.NewInt(0)) {
		panic(amount)
	}
	if (from == to) {
		panic("self transfer")
	}
	if from != nil {
		balance := evm.StateDB.GetBalance(*from)
		if arbmath.BigLessThan(balance, amount) {
			return fmt.Errorf("%w: addr %v have %v want %v",
vm.ErrInsufficientBalance, *from, balance, amount)
		}
		evm.StateDB.SubBalance(*from, amount)
```

```
        }
        ...
```

Figure D.3: The header of the `TransferBalance` function, which includes two new checks

An incorrect use of `TransferBalance` will trigger the first panic. An occurrence of the second panic may not be indicative of incorrect use but should be investigated regardless. There is no need to include both checks in the final version of ArbOS, so we recommend using build tags when compiling the code.

E. Echidna Invariant Test for HashProofHelper

Trail of Bits wrote a differential fuzz test to compare the `proveWithFullPreimage` function and the `proveWithSplitPreimage` function. It checks that the functions return the same `fullHash` and split preimage when given the same inputs and that they never revert when provided valid inputs. When executed with an unusually high Echidna test limit of 1 million runs, the test passed. Additionally, to achieve better coverage, we implemented a small fix to allow Echidna to generate arrays of up to 150 elements.

```
pragma solidity 0.8.9;

import "./HashProofHelper.sol";

contract Echidna {
    HashProofHelper hashProofHelperFull;
    HashProofHelper hashProofHelperSplit;

    constructor() {
        hashProofHelperFull = new HashProofHelper();
        hashProofHelperSplit = new HashProofHelper();
    }

    function fuzz(bytes calldata data, uint64 offset) public {
        bytes32 fullHash_full;
        bool reverted_full;
        offset = uint64(uint256(offset) % data.length);
        try hashProofHelperFull.proveWithFullPreimage(data, offset) returns(bytes32
fullHash_ret) {
            fullHash_full = fullHash_ret;
        } catch (bytes memory e) {
            reverted_full = true;
        }

        uint256 len = data.length;
        uint256 provenLen;
        bytes32 fullHash_split;
        bool reverted_split;
        while (fullHash_split == bytes32(0) && !reverted_split) {
            uint256 newProvenLen = provenLen + 136;
            if (newProvenLen > len) {
                newProvenLen = len;
            }
            uint256 isFinal = newProvenLen == len ? 1 : 0;
            try
hashProofHelperSplit.proveWithSplitPreimage(data[provenLen:newProvenLen], offset,
isFinal) returns(bytes32 fullHash_ret) {
                provenLen = newProvenLen;
                fullHash_split = fullHash_ret;
            } catch (bytes memory e) {
                reverted_split = true;
```

```
            }
        }
        assert(reverted_full == reverted_split && !reverted_full);
        assert(fullHash_split == fullHash_full);
        bytes memory part_full = hashProofHelperFull.getPreimagePart(fullHash_full,
offset);
        bytes memory part_split =
hashProofHelperSplit.getPreimagePart(fullHash_split, offset);
        assert(part_full.length <= 32 && part_split.length <= 32);
        assert(keccak256(part_full) == keccak256(part_split));
    }
}
```

Figure E.1: The Echidna property test

F. 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.

ArbOS

- **Use lowercase letters for package names.**

```
nodeInterface/virtual-contracts.go:4:1: don't use MixedCaps in package name;
nodeInterface should be nodeinterface (golint)
package nodeInterface
^
nodeInterface/NodeInterfaceDebug.go:4:1: don't use MixedCaps in package name;
nodeInterface should be nodeinterface (golint)
package nodeInterface
^
nodeInterface/NodeInterface.go:4:1: don't use MixedCaps in package name;
nodeInterface should be nodeinterface (golint)
package nodeInterface
^
```

Figure F.1: Package names that use `MixedCaps`

- **Replace all static error messages with dynamic error messages.** This will make it easier to reuse code throughout the repository.

- **Use `crypto/rand` rather than `math/rand` in `util/testhelpers/testhelpers.go`.**

- **Ensure that the codebase handles panics consistently.**

- **Ensure that all casting operations are safe casts.**

```
err = con.LifetimeExtended(c, evm, ticketId, big.NewInt(int64(newTimeout)))
return big.NewInt(int64(newTimeout)), err
```

Figure F.2: An unsafe cast in `nitro/precompiles/ArbRetryableTx.go`

- **Remove unused parameters (e.g., the `timeToAdd` parameter in `arbos/retryable/retryable.go#L218`).**

HashProofHelper

- **Replace the expression `delete keccakStates[msg.sender]` with a call to the `clearSplitProof` function.** The use of helper functions such as this one will reduce the amount of duplicated code.

- **Remove the shift by 0 on line 58 of `HashProofHelper.sol`.**

- **Remove the + `stateIdx / 5` from the state index calculation; this operation is unnecessary, as `stateIdx / 5` will always be equal to 0.**

```
for (uint256 i = 0; i < 32; i++) {
    uint256 stateIdx = i / 8;
    // work around our weird keccakF function state ordering
    stateIdx = 5 * (stateIdx % 5) + stateIdx / 5;
```

Figure F.3: `HashProofHelper.sol#L87–90`

- **Change the pragma from ^0.8.0 to a higher version that supports custom errors.**

```
Error: Expected ';' but got '('
  --> src/osp/HashProofHelper.sol:10:16:
   |
10 | error NotProven(bytes32 fullHash, uint64 offset);
   |
```

Figure F.4: `HashProofHelper.sol#L87–90`

Migration-Related Contracts

- **Remove the unused `Sequencer.postUpgradeInit` function.** The inclusion of dead code can increase the size of a codebase and reduce readability.

- **Remove `latestCompleteStep` from the `NitroMigrator.initialize()` function.** The enum is already set to `Uninitialized` by default.

- **Use consistent function names.** The use of the function names `pause` and `resume` (rather than `pause` and `unpause`) may cause confusion.

G. Nitro Migration Plan Recommendations

- **Differentiate between the operations that are performed manually and those that are handled by off-chain scripts.** Where possible, implement automated deployment checks to supplement manual analysis work.
- **Specify concrete time limits for the execution of various steps in the migration process.** Additional detailed information on how long it will take for a batch to be posted, for example, will be helpful during the migration process.
- **Ensure that any errors in off-chain components are surfaced, and add links to the relevant off-chain scripts where possible.**
- **Clearly differentiate between validator and sequencer events that *should* be logged during a migration and those that should not be.** This will enable the team to triage the state of these components.
- **Ensure that all smart contract calls are explicitly mentioned in the migration plan.** For example, the call to the `configureDeployment` function should be "step 0" of the plan so that the migration begins with the setup of the `NitroMigrator` contract.
- **Consider using automated scripts to handle transfers of contracts' ownership.** Automating this process will reduce the likelihood of mistakes.