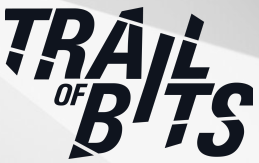

Offchain Labs Arbitrum Nitro External DA

Security Assessment (Summary Report)

January 12, 2026

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

Prepared by: **Simone Monica and Jaime Iglesias**

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:

Benjamin Samuels, Engineering Director, Blockchain
benjamin.samuels@trailofbits.com

The following consultants were associated with this project:

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Simone Monica, Consultant
simone.monica@trailofbits.com

Project Timeline

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

Date	Event
December 9, 2025	Project kick-off call
December 19, 2025	Delivery of report draft
December 30, 2025	Delivery of final summary report
January 12, 2026	Completion of fix review

Project Targets

The engagement involved reviewing and testing the following targets.

nitro

Repository	https://github.com/OffchainLabs/nitro/
Version	`3f226a0ecf91ceed3688a2fc41969fae1f32d03d`
Type	Go/Rust
Platform	EVM

nitro-contracts

Repository	https://github.com/OffchainLabs/nitro-contracts/
Version	`940373b68d0e9cffa006eb9e6d0b4376138d531c`
Type	Solidity
Platform	EVM

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of its Nitro product, with an emphasis on the changes made to implement its external Data Availability (DA) extension, which allows external DA providers to integrate with Nitro without requiring any custom changes to the node or the contracts.

A team of two consultants conducted the review from December 9 to December 19, 2025, for a total of four engineer-weeks. With full access to source code and documentation, we performed static and dynamic testing of the target, using automated and manual processes.

Observations and Impact

The main goal of the review was to assess the correctness of the external DA implementation, looking for potential implementation mistakes (e.g., missing checks), unintended divergences between Nitro and the challenge contracts, and potential opportunities for improvements.

Overall, we found the implementation to be clear and thoughtful. The provided support documentation for the changes is comprehensive and provides explanations behind the reasoning for all the changes, as well as the intended final properties of the system, which was very helpful during the review.

The issues disclosed in this report are mainly related to edge cases, missing checks, and divergences in behavior between Nitro and the contracts.

Recommendations

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

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes or broader refactoring efforts.

Summary of Findings

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

ID	Title	Type	Severity
1	Missing validations in executeValidatePreimage	Data Validation	**Undetermined**
2	Missing sequencer message length checks	Data Validation	**Low**
3	Unclear why the blob decode error is not being propagated	Error Reporting	**Informational**
4	Unclear behavior during sequencer message parsing	Undefined Behavior	**Low**

Detailed Findings

1. Missing validations in executeValidatePreimage

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-EXTDA-1
Target: `arbitrator/prover/src/machine.rs`, `src/osp/OneStepProverHostIo.sol`	

Description

The `executeValidatePregimage` function of the `oneStepProverHostio` contract is missing validation of the `PreImageType` and `ptr` inputs; validation of these inputs is present in the equivalent implementation in Nitro's `machine.rs`, so there is a divergence between Nitro and the challenge protocol.

We can see in the figure below that, in Nitro, both the `preImageType` and the `ptr` inputs are checked. Note that an invalid `preimage_type` will cause 0 to be pushed to the stack, while an invalid `ptr` will cause the machine status to be set to `errored` through the `error!` macro.

```
Opcode::ValidateCertificate => {
    let preimage_type = value_stack.pop().unwrap().assume_u32();
    let hash_ptr = value_stack.pop().unwrap().assume_u32();

    // Try to convert preimage_type to PreimageType
    let Ok(preimage_ty) = PreimageType::try_from(u8::try_from(preimage_type)?)
    else {
        // For invalid preimage types, return 0 (invalid)
        value_stack.push(Value::from(0u32));
        continue;
    };

    // Load the hash from memory
    let Some(hash) = module.memory.load_32_byte_aligned(hash_ptr.into()) else {
        error!();
    };


    [...]
```

Figure 1.1: Part of the `step_n` function in `machine.rs#L2470–L2485`

For proving purposes, it is of the utmost importance that the behavior in `machine.rs` is accurately replicated by the challenge protocol; however, when we check the `oneStepProverHostio` contract's `executeValidatePregimage` function, we can see that there is a divergence in behavior.

```
function executeValidatePreimage(
    ExecutionContext calldata,
    Machine memory mach,
    Module memory mod,
    Instruction calldata,
    bytes calldata proof
) internal view {
    uint256 preimageType = mach.valueStack.pop().assumeI32();
    uint256 ptr = mach.valueStack.pop().assumeI32();

    [...]

    if (preimageType == 3) {
        require(address(customDAValidator) != address(0),
"CUSTOM_DA_VALIDATOR_NOT_SUPPORTED");
        if (validateAndCheckCertificate(proof, proofOffset, leafContents)) {
            mach.valueStack.push(ValueLib.newI32(1));
        } else {
            mach.valueStack.push(ValueLib.newI32(0));
        }
    } else {
        // Non-CustomDA always valid
        mach.valueStack.push(ValueLib.newI32(1));
    }

    // Update merkle root
    mod.moduleMemory.merkleRoot = merkleProof.computeRootFromMemory(leafIdx,
leafContents);
}
```

Figure 1.2: Part of the `executeValidatePreimage` function in OneStepProverHostio.sol#L275–307

`preImageType` and `ptr` are simply popped from the stack (just like in `machine.rs`); however, there are no equivalent checks to `u8::try_from(inst.argument_data)?` and `module.memory.load_32_byte_aligned(hash_ptr.into())`;

The reason this finding is of undetermined severity is because, as of this moment, it is unclear whether exploiting this divergence is possible in practice.

Recommendations

Short term, consider including additional checks in `executeValidatePreImage`, as follows.

For `preImageType`, the `executeValidatePreimage` function should return 0 when an

invalid `preImageType` is input (e.g., `>= 4 && <= 255`) and should revert otherwise (e.g., `> 255`).

For `ptr`, the machine status should be set to `errored` if `mod.moduleMemory.isValidLeaf(ptr)` returns false.

Long term, ensure that any divergences between Nitro and the challenge protocol contracts are intended, and thoroughly document them.

2. Missing sequencer message length checks

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-EXTDA-2
Target: `daprovider/server/provider_server.go`, `daprovider/reader.go`	

Description

The `RecoverPayload` and `CollectPreimages` functions in `provider_server.go` do not perform any length checks on the `sequencerMsg`.

First, the public functions in the server will be called, where no length checks are performed.

```
func (s *ReaderServer) RecoverPayload(
	ctx context.Context,
	batchNum hexutil.Uint64,
	batchBlockHash common.Hash,
	sequencerMsg hexutil.Bytes,
) (*daprovider.PayloadResult, error) {
	promise := s.reader.RecoverPayload(uint64(batchNum), batchBlockHash,
sequencerMsg)
	[...]
}

func (s *ReaderServer) CollectPreimages(
	ctx context.Context,
	batchNum hexutil.Uint64,
	batchBlockHash common.Hash,
	sequencerMsg hexutil.Bytes,
) (*daprovider.PreimagesResult, error) {
	promise := s.reader.CollectPreimages(uint64(batchNum), batchBlockHash,
sequencerMsg)
	[...]
```

Figure 2.1: The `RecoverPayload` and `CollectPreimages` functions in `provider_server.go#L188–L214`

Later, the functions in the DA provider reader will be called, which will also not perform any checks.

```
// RecoverPayload fetches the underlying payload from the DA provider
func (b *readerForBlobReader) RecoverPayload(
	batchNum uint64,
```

```
        batchBlockHash common.Hash,
        sequencerMsg []byte,
) containers.PromiseInterface[PayloadResult] {
        return containers.DoPromise(context.Background(), func(ctx context.Context)
(PayloadResult, error) {
                payload, _, err := b.recoverInternal(ctx, batchBlockHash, sequencerMsg,
true, false)
                return PayloadResult{Payload: payload}, err
        })
}

// CollectPreimages collects preimages from the DA provider
func (b *readerForBlobReader) CollectPreimages(
        batchNum uint64,
        batchBlockHash common.Hash,
        sequencerMsg []byte,
) containers.PromiseInterface[PreimagesResult] {
        return containers.DoPromise(context.Background(), func(ctx context.Context)
(PreimagesResult, error) {
                _, preimages, err := b.recoverInternal(ctx, batchBlockHash,
sequencerMsg, false, true)
                return PreimagesResult{Preimages: preimages}, err
        })
}
```

Figure 2.2: The `RecoverPayload` and `CollectPreimages` functions in `reader.go#L115–L136`

Finally, the `recoverInternal` function will be reached, where the server will crash if it attempts to access an offset of the message and the length is not the expected one.

```
// recoverInternal is the shared implementation for both RecoverPayload and
CollectPreimages
func (b *readerForBlobReader) recoverInternal(
        ctx context.Context,
        batchBlockHash common.Hash,
        sequencerMsg []byte,
        needPayload bool,
        needPreimages bool,
) ([]byte, PreimagesMap, error) {
        blobHashes := sequencerMsg[41:]
        [...]
```

Figure 2.3: The `recoverInternal` function in `reader.go#L70–L77`

Note that we assume this server is intended for local use; therefore, a malicious user is not part of the threat model, which is why this issue is not of higher severity.

Exploit Scenario

A malicious user sends an invalid sequencer message to the provider server, and when it is parsed, it causes the server to crash.

Recommendations

Short term, implement length checks on the `sequencerMsg`.

Long term, thoroughly review similar code paths, ensuring all the needed checks are present. This can also be done effectively by including tests with malformed or invalid messages.

3. Unclear why the blob decode error is not being propagated

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-EXTDA-3
Target: `daprovider/reader.go`	

Description

The `recoverInternal` function will treat blobs that fail to decode as empty batches instead of propagating the error. While we believe this is the correct behavior, we recommend adding code comments to explain why the error is not propagated.

```
	var payload []byte
	if needPayload {
		payload, err = blobs.DecodeBlobs(kzgBlobs)
		if err != nil {
			log.Warn("Failed to decode blobs", "batchBlockHash",
batchBlockHash, "versionedHashes", versionedHashes, "err", err)
			return nil, nil, nil
		}
	}

	return payload, preimages, nil
}
```

Figure 3.1: Snippet of the `recoverInternal` function, showing that it does not propagate the error, in `reader.go#L102–L109`

Recommendations

Short term, include a code comment explaining why the error is not being propagated and why decode failures are treated as empty batches.

Long term, whenever possible, include code comments that explain system behavior under certain situations (e.g., edge cases, errors).

4. Unclear behavior during sequencer message parsing

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-EXTDA-4
Target: `arbstate/inbox.go`	

Description

The following error message suggests that the `ParseSequencerMessage` function treats sequencer batches that fail certificate validation as empty batches; however, for the batch to be treated as an empty batch, the `payload` needs to be set to `nil`.

```
        } else if daprovider.IsDACertificateMessageHeaderByte(payload[0]) &&
daprovider.IsCertificateValidationError(err) {
                log.Warn("Certificate validation of sequencer batch failed, treating it
as an empty batch", "batch", batchNum, "error", err)
        } else {
                return nil, err
        }
} else {
        payload = result.Payload
}
if payload == nil {
        return parsedMsg, nil
}
[...]
```

Figure 4.1: Part of the `ParseSequencerMessage` function in `inbox.go#L124–L135`

Exploit Scenario

An invalid certificate is posted to the inbox; however, when processed, it is not processed as an empty batch, contrary to what was expected.

Recommendations

Short term, have the `ParseSequencerMessage` function set the payload to `nil` when the certificate validation fails.

Long term, thoroughly document this behavior and ensure code branches behave as expected. This can be done by enhancing the testing by including invalid inputs—in this case, invalid certificates.

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 Quality Issues

This appendix contains findings that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- Generic "CustomDA enhancement" language is used in error messages across the codebase. Replace these uses with the relevant name (`validateCertificate` or `readPreimage`).

```
if len(proof) < minProofSize {
	return nil, fmt.Errorf("proof too short for CustomDA enhancement:
expected at least %d bytes, got %d", minProofSize, len(proof))
}

// Verify marker
if proof[markerPos] != MarkerCustomDAReadPreimage {
	return nil, fmt.Errorf("invalid marker for CustomDA enhancer: 0x%02x",
proof[markerPos])
}
```

Figure B.1: Example error messages in readpreimage_proof_enhancer.go#L73–L87

- `validatecertificate_proof_enhancer` validates the proof before retrieving the message from the inbox, whereas `readpreimage_proof_enhancer` does not. Essentially, the order of checks between them differs.

```
func (e *ValidateCertificateProofEnhancer) EnhanceProof(ctx context.Context,
messageNum arbutil.MessageIndex, proof []byte) ([]byte, error) {
	// Extract the hash and marker from the proof
	// Format: [...proof..., certHash(32), marker(1)]
	minProofSize := CertificateHashSize + MarkerSize
	if len(proof) < minProofSize {
		return nil, fmt.Errorf("proof too short for ValidateCertificate
enhancement: expected at least %d bytes, got %d", minProofSize, len(proof))
	}
```

Figure B.2: Part of the EnhanceProof function in validatecertificate_proof_enhancer.go#L36–L42

```
// EnhanceProof implements ProofEnhancer for CustomDA
func (e *ReadPreimageProofEnhancer) EnhanceProof(ctx context.Context,
messageNum arbutil.MessageIndex, proof []byte) ([]byte, error) {
	batchContainingMessage, found, err :=
e.inboxTracker.FindInboxBatchContainingMessage(messageNum)
	if err != nil {
		return nil, err
	}
```

```
	if !found {
		return nil, fmt.Errorf("Couldn't find batch for message #%d to
enhance proof", messageNum)
	}
```

Figure B.3: Part of the EnhanceProof function in readpreimage_proof_enhancer.go#L39–L46

- `validateCertificate` does not check that the first byte of the certificate is the `DACertificateMessageHeaderFlag`.

```
validator := e.dapRegistry.GetValidator(certificate[0])
if validator == nil {
    return nil, fmt.Errorf("no validator registered for certificate type
0x%02x", certificate[0])
}
```

Figure B.4: Part of the EnhanceProof function in readpreimage_proof_enhancer.go#L104–107

- `validateCertificate` does not check the length of `MinCertificateSize`. Semantically, the highlighted check is equivalent to checking if `len == 0`, since `MinCertificateSize` is 1. However, if `MinCertificateSize` were to change, then this could become a problem.

```
// Validate certificate format
if len(certificate) < MinCertificateSize {
	return nil, fmt.Errorf("certificate too short: expected at least %d
bytes, got %d", MinCertificateSize, len(certificate))
}
```

Figure B.5: Part of the EnhanceProof function in readpreimage_proof_enhancer.go#L61–L64

- Different code styles are used in `validatecertificate_proof_enhancer` and `readpreimage_proof_enhancer`. The first one uses `offset` while the second one does not.

```
offset := originalProofLen
binary.BigEndian.PutUint64(enhancedProof[offset:], certSize)
offset += CertificateSizeFieldSize

// Add certificate
copy(enhancedProof[offset:], certificate)
offset += len(certificate)

// Add validity proof
copy(enhancedProof[offset:], validityProof)
```

Figure B.6: Part of the EnhanceProof function in validatecertificate_proof_enhancer.go#L110–L119

```
// Copy original proof up to the CustomDA marker data
copy(enhancedProof, proof[:markerDataStart])

// Add certSize
binary.BigEndian.PutUint64(enhancedProof[markerDataStart:], certSize)

// Add certificate
copy(enhancedProof[markerDataStart+CertificateSizeFieldSize:], certificate)

// Add custom proof
copy(enhancedProof[markerDataStart+CertificateSizeFieldSize+len(certificate):]
, customProof)
```

Figure B.7: Part of the EnhanceProof function in readimage_proof_enhancer.go#L126–L32

C. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On January 12, 2026, Trail of Bits reviewed the fixes and mitigations implemented by the Offchain Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, Offchain Labs has resolved all four issues disclosed in this report. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Missing validations in executeValidatePreimage	Undetermined	Resolved
2	Missing sequencer message length checks	Low	Resolved
3	Unclear why the blob decode error is not being propagated	Informational	Resolved
4	Unclear behavior during sequencer message parsing	Low	Resolved

Detailed Fix Review Results

TOB-EXTDA-1: Missing validations in executeValidatePreimage
Resolved in PR #398. Validations on the `preImageType` and `ptr` inputs have been added in the `executeValidatePreImage` function. Additionally, PR #4187 made changes to the `serialize_proof` function for the `ValidateCertificate` opcode on the `preimage_type`.

TOB-EXTDA-2: Missing sequencer message length checks
Resolved in PR #4214. A length check on the sequencer message has been added to the `recoverInternal` function of the `readerForBlobReader` struct before the message is indexed.

TOB-EXTDA-3: Unclear why the blob decode error is not being propagated
Resolved in PR #4182. A comment has been added explaining why the error is not propagated.

TOB-EXTDA-4: Unclear behavior during sequencer message parsing
Resolved in PR #4149. The `payload` is set to nil to correctly handle it as an empty batch.

D. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

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 assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up with our latest news and announcements, please follow @trailofbits on X or LinkedIn 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

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information 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 software controls and security properties. 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. A project's time and resource constraints also limit their use.