Offchain Labs Stylus Permissioned Deployer

Security Assessment (Summary Report)

November 6, 2026

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

Prepared by: **Jaime Iglesias**

Table of Contents

Table of Contents **1**
Project Summary **2**
Project Targets **3**
Executive Summary **4**
Summary of Findings **5**
Detailed Findings **6**
1. Fragments must be deployed and verified before signing 6
A. Vulnerability Categories **8**
About Trail of Bits **10**
Notices and Remarks **11**

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 consultant was associated with this project:

Jaime Iglesias, Consultant
jaime.iglesiasbotas@trailofbits.com

Project Timeline

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

Date	Event
November 6, 2026	Delivery of final summary report

Project Targets

The engagement involved reviewing and testing the following target.

nitro-contracts-private

Repository	https://github.com/OffchainLabs/nitro-contracts-private/
Version	3c31811787912aacdb17de44feb1f67ea85d04b2
Type	Solidity
Platform	EVM

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review its Stylus permissioned deployer contract. This contract changes Stylus program activation from a permissionless process to a permissioned one on Arbitrum's first-party chains (One, Nova, and Sepolia). Its main use is to prevent hand-crafted Wasm programs from being deployed on the network.

The review covered the `StylusPermissionedDeployer`, `ActivationGasSetter`, and `InstallStylusPermissionedDeployerAction` contracts.

We focused on verifying a number of assumptions and invariants documented by the Offchain Labs team, looking for any edge cases or undesired behavior.

One consultant conducted the review on October 2 and from October 4 to October 5, 2026, for a total of three engineer-days of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

We identified only one issue during the review, of informational severity: signing a fragmented Stylus program before its fragments are deployed and verified could allow activation of code that differs from what the signer expected.

Recommendations

- **Remediate the finding disclosed in this report.** This finding 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	Fragments must be deployed and verified before signing	Data Validation	**Informational**

Detailed Findings

1. Fragments must be deployed and verified before signing	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SDEP-1
Target: `src/stylus/StylusPermissionedDeployer.sol`	

Description

An activation signature commits to a program's code hash and a deadline. For a non-fragmented program, that code hash identifies the complete deployed runtime. For a fragmented program, it identifies only the root, which contains the fragment addresses but not their code. Therefore, signing the root before its fragments are deployed and verified could allow activation of code that differs from what the signer expected.

The `deployBySig` function passes the result of `payloadCodehash` into the signed digest, while the `activateBySig` function uses the existing program's code hash. As shown in figure 1.1, the `payloadCodehash` function hashes the runtime program contained in a standard deployment payload.

```
function payloadCodehash(
    bytes calldata payload
) public pure returns (bytes32) {
    if (
        payload.length < PAYLOAD_PRELUDE_LENGTH || payload[0] != 0x7f
            || uint256(bytes32(payload[1:33])) != payload.length -
PAYLOAD_PRELUDE_LENGTH
            || bytes9(payload[33:42]) != PAYLOAD_COPY_AND_RETURN
    ) {
        revert NonStandardPayload();
    }
    return keccak256(payload[PAYLOAD_PRELUDE_LENGTH:]);
}
```

Figure 1.1: The standard payload's runtime program is hashed for authorization. (nitro-contracts-private/src/stylus/StylusPermissionedDeployer.sol#L267-L278)

The same code-hash authorization has a different meaning for fragmented programs. The contract documentation explicitly states that the root code hash covers the fragment addresses, not the code stored at them (figure 1.2).

```
///         Fragments of a split program cannot be activated directly, so each one
is deployed without this
///         contract. The root, which lists the fragment addresses, is deployed and
activated like any other
///         program. Its codehash covers the fragment addresses it lists, not the
code at them.
```

Figure 1.2: The root code hash does not cover fragment code. (nitro-contracts-private/src/stylus/StylusPermissionedDeployer.sol#L32-L34)

Program Form	What the Signed Code Hash Identifies	Required Assumption
Non-fragmented	The complete deployed runtime code	The deployed runtime matches the signed hash.
Fragmented	Root metadata and the ordered fragment addresses	Every fragment is deployed, and its live code is verified before the root is signed.

Because deployed fragment code is immutable, following a deploy-and-verify-first flow makes the signed address list refer to fixed code. Signing a root containing predicted or otherwise unverified fragment addresses would break this assumption because the signature would not constrain the code later deployed at those addresses.

Exploit Scenario

A signer approves a fragmented root before its fragments are deployed and verified. The party controlling the fragment deployments places different code at the addresses listed by the approved root and then submits the root for activation. The root code hash still matches the signature even though the referenced fragment code differs from what the signer expected.

Recommendations

Short term, make the required order explicit and mandatory: deploy every fragment, verify that the live code at each listed address matches the expected fragment, and only then sign the root. Signing tooling should reject roots containing missing or unverified fragments.

Long term, consider implementing an alternative verification process for fragmented programs, such as including the ordered fragment code hashes in the signed commitment and checking them against live code before activation.

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**
Not Applicable	This issue is of informational severity and does not pose an immediate risk, so difficulty does not apply.
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.

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

© 2026 by Trail of Bits, Inc.

All rights reserved. Trail of Bits hereby asserts its right to be identified as the creator of this report in the United Kingdom.

Trail of Bits considers this report public information; it is licensed to Offchain Labs under the terms of the project statement of work and has been made public at Offchain Labs' request. Material within this report may not be reproduced or distributed in part or in whole without Trail of Bits' express written permission.

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through sources other than that page may have been modified and should not be considered authentic.

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.