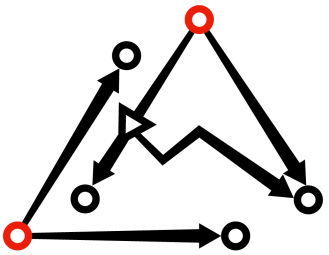

Least Authority

PRIVACY MATTERS

Derivation Pipeline Changes: OP Stack (Go and Rust)

Security Audit Report

Espresso Systems

Final Audit Report: 11 August 2026

Table of Contents

Overview

Background

Espresso Systems has requested that Least Authority perform a security audit of its derivation pipeline changes for the OP Stack (Go and Rust).

Project Dates

- **July 6, 2026 - July 17, 2026:** Initial Code Review *(Completed)*
- **July 20, 2026:** Delivery of Initial Audit Report *(Completed)*
- **August 11, 2026:** Verification Review *(Completed)*
- **August 11, 2026:** Delivery of Final Audit Report *(Completed)*

Review Team

- George Gkitsas, Security / Cryptography Researcher and Engineer
- Will Sklenars, Security Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the derivation pipeline changes followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories were considered in scope for the review:

- Rust:
  - `celo-kona` changes,
    - Corresponds to PR #220
  - Generalize EigenDADataSource over DataAvailabilityProvider
    - Corresponds to PR #2
- Go:
  - Espresso 2: derivation pipeline
    - Main changes to the derivation pipeline in the Go implementation
    - Corresponds to: PR #457
  - Espresso: drop dead calldata-source authentication + guard espresso_time >= ecotone_time
    - Dead code removal
    - Corresponds to PR #466

Specifically, we examined the following Git revisions for our initial review:

- `celo-org/optimism`: `095703380f5970a5febc642f7a9a1938596166ce`
- `celo-org/celo-kona`: `10ef4cf27a5d9775610c8b8d48e444a88fb67ced`
- `celo-org/hokulea`: `a57a866dd35b06e1e0565b226641c658001272bb`

For the verification, we examined the following Git revisions:

- `celo-org/optimism`: `9a3a81397fd059120745e2a5c8543019b09648d6`
- `celo-org/celo-kona`: `5ced403de1ee21c2d5aabd300061472a1df2f099`
- `celo-org/hokulea`: `a57a866dd35b06e1e0565b226641c658001272bb`

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Batch Authenticator and Derivation Pipeline Audit Scope Google Sheet *(shared with Least Authority via email on 18 May 2026)*.
- Website: https://www.espressosys.com

In addition, this audit report references the following documents:

- Derivation - OP Stack Specification: https://specs.optimism.io/protocol/ecotone/derivation.html
- Fault Proof - OP Stack Specification: https://specs.optimism.io/fault-proof/index.html
- EigenDA - Celo Specification: https://specs.celo.org/eigenda
- celo-integration-specs-31e9317.pdf *(shared with Least Authority via email on 15 June 2026)*

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Whether requests are passed correctly to the network core;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution;
- Protection against malicious attacks and other ways to exploit;
- Inappropriate permissions and excess authority;
- Maintenance risks and proactive dependency management;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a review of the integration of Espresso's confirmation layer with Celo, an OP Stack rollup. The integration is intended to provide faster assurance that the transaction batches a node observes will match those eventually finalized on Ethereum. In a standard OP Stack chain, a node can determine the canonical chain only once batch data is derived from L1. To shorten this delay, a batcher running in a Trusted Execution Environment (TEE) submits each batch to the Espresso network for fast

confirmation, and Celo's derivation pipeline is modified to accept only batches that a TEE batcher has authenticated on L1 through the `BatchAuthenticator` contract.

The reviewed components were Celo's forks of Optimism (Go) and Kona (Rust), as well as Hokulea, which provides `EigenDA` support for the Rust pipeline. These components implement corresponding Go and Rust derivation pipelines for batches carried as `calldata`, EIP-4844 blobs, or `EigenDA` commitments, including the authorization checks the Espresso integration requires. These pipelines are the read path, reconstructing the L2 chain from L1 data. The Go pipeline runs in Celo nodes to derive the canonical chain, while the Rust pipeline runs under SP1 to reconstruct the same chain when generating the ZK fault proofs that resolve disputes over L2 state on L1.

The write path (the TEE batcher) and the `BatchAuthenticator` contract (which was covered in our prior TEE Smart Contracts audit) were out of scope. Our team compared the Go implementation against the provided specification, reviewed the Go and Rust implementations for parity, and assessed the code for security issues.

We found the two pipelines consistent with one another. Our findings relate to specification inconsistencies and operational hardening recommendations, together with one issue in a related but out-of-scope component.

System Design

Our team reviewed the system's design and implementation and found that security has been taken into consideration, as demonstrated by the requirement that batches be authenticated before derivation accepts them, the binding of each batch commitment to its sender to prevent replay by another party, and event processing that checks the emitting contract, receipt status, event format, commitment, and sender. Invalid configurations fail early, and the design separates keys through a dedicated signing service and supports fallback operation when Espresso is unavailable. In addition, the specification contains explicit security requirements with testable acceptance criteria. We recommend that the Espresso team continue using explicit requirements as the system evolves.

Post-fork, the derivation pipeline replaces the upstream sender-based check with event-based authorization. A batch is accepted only if a matching `BatchInfoAuthenticated` event, emitted by the configured `BatchAuthenticator` contract, appears in the L1 receipts within a fixed lookback window, and the batch's L1 sender equals the address that emitted it. The verified TEE signature establishes that the batch content was confirmed by Espresso regardless of the entity that posts it, while the sender-to-caller binding prevents a third party from replaying an authenticated batch as a resource-exhaustion vector. Our team noted two instances where the specification no longer matches this implementation: the lookback window is described as configurable but is a fixed compile-time constant (Suggestion 3), and the authorization check is attributed to both data sources but is now applied by the blob data source alone (Suggestion 2).

Because the two pipelines must derive identical results for the proven state roots to match the chain that nodes follow, our team reviewed them for parity and found no divergence. However, we noted that the fault-proof host does not yet support EIP-4844 blob retrieval (Suggestion 4). This agreement depends on identical configuration: the Espresso parameters (activation time and authenticator address) are maintained separately in the Rust proof program and the Go node, with no common source. This is an operational concern rather than a code defect. We recommend establishing a single source of truth for these parameters (Suggestion 5) and documenting a verifiable post-deployment procedure to confirm consistency (Suggestion 6).

The design supports continued operation when Espresso is unavailable through a fallback batcher and separates signing keys through a dedicated service so that the batcher's L1 key and the enclave's signing

key are held independently. Our team reviewed the interaction between batcher deactivation and the lookback window. We found that the specification states that deactivation leaves no window in which two batchers are effective, but the implementation continues to honor already-authenticated batches until their events age out of the lookback window (Suggestion 1). Separately, while examining the TEE verifier on which the pipeline depends, a component outside this engagement's scope, our team identified a governance inconsistency in enclave-hash revocation (Issue B).

Dependencies

We examined the dependencies implemented in the codebase and identified three concerns. Reachable denial-of-service vulnerabilities exist in the Go toolchain's TLS, `HTTP/2`, and `X.509` implementations (Issue A). Duplicated upstream Kona code presents maintenance and security risk (Suggestion 7). In addition, stronger controls are recommended for dependency pinning, updates, forks, and parallel dependency versions (Suggestion 8).

Code Quality

We performed a manual review of the repositories in scope and found the code to be well-organized and aligned with development best practices.

Tests

The repositories in scope include substantial unit and component tests. However, integration testing can be improved.

Documentation and Code Comments

The project documentation sufficiently describes the system's intended functionality and is notable for stating explicit security requirements with testable acceptance criteria. Several of our team's suggestions concern keeping the specification aligned with implementation changes. Additionally, the codebase includes clear and useful comments, which aid in understanding the intended behavior of the relevant components.

Scope

The scope of this review was sufficient and included all security-critical components. Our team identified one issue (Issue B), which concerns code that was out of scope.

Specific Issues & Suggestions

We list the issues and suggestions identified during the review in descending order of severity. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	SEVERITY	STATUS
Issue A: Vulnerable Go Networking Libraries Allow L1 RPC Responses to Halt Derivation	Low	Resolved
Issue B: Enclave-Hash Revocation Is Reachable at a Lower Governance Threshold Than Intended (Out of Scope)	Informational	Resolved
Suggestion 1: Clarify REQ-CONTRACT-SAF-002 to Reflect the	Informational	Resolved

Post-Deactivation Decay-Window		
Suggestion 2: Update 3.3.2.1.5 to Attribute Espresso Authorization to the Blob Data Source Only	Informational	Resolved
Suggestion 3: Update the Specification to Describe the Lookback Window as a Fixed Constant, Not a Configurable Value	Informational	Resolved
Suggestion 4: Support EIP-4844 Blobs in the Celo Fault Proof Host	Informational	Resolved
Suggestion 5: Establish a Single Source of Truth for the Espresso Configuration	Informational	Unresolved
Suggestion 6: Develop a Playbook and/or Tools to Verify Espresso Configuration After Deployment	Informational	Partially Resolved
Suggestion 7: Add Automated Parity Checks for Duplicated Upstream Kona Derivation Code	Informational	Unresolved
Suggestion 8: Pin and Continuously Monitor Integration Dependencies	Informational	Unresolved

Issue A: Vulnerable Go Networking Libraries Allow L1 RPC Responses to Halt Derivation

Location

`optimism/go.mod#L3-L5`

`optimism/op-node/rollup/derive/batch_authenticator.go#L153-L198`

`optimism/op-service/sources/receipts_rpc.go#L73-L99`

`celo-org/op-geth/rpc/http.go#L216-L243`

`GO-2026-4870`

`GO-2026-4918`

`GO-2026-5037`

Synopsis

The project builds the Go derivation client with Go `1.24.13`, which contains three denial-of-service vulnerabilities reachable through the L1 remote procedure call requests used for Espresso batch authentication. An attacker who controls or intercepts the connection can cause a connection deadlock, an HTTP/2 transport loop, or excessive certificate verification work, preventing the node from completing derivation.

Preconditions

The node must be built with Go `1.24.13`, Espresso authentication must be active, and a cache miss must cause the derivation pipeline to request receipts or an ancestor header.

Exploitation also requires one of the following capabilities:

- Control of the configured HTTPS or secure WebSocket L1 RPC provider, its TLS proxy, or its certificate, allowing the attacker to send malicious TLS 1.3 `KeyUpdate` messages.
- Control of an L1 RPC connection using `HTTP/2`, allowing the attacker to send a `SETTINGS` frame with `SETTINGS_MAX_FRAME_SIZE` set to zero.
- The ability to intercept or redirect the L1 RPC connection, allowing the attacker to present an untrusted certificate containing a large DNS Subject Alternative Name list.

Likelihood

Low.

Production nodes commonly access L1 data through HTTPS endpoints, and the affected transport functions are used whenever an uncached receipt or header is requested. Two attack paths require control of the configured endpoint, while the certificate attack occurs before certificate trust is established and can be attempted by a network adversary.

Impact

Medium.

Successful exploitation consumes connection or processor resources and prevents the affected node from retrieving the L1 data required for authenticated batch derivation. This can halt the node and delay services that depend on its derived chain, but it does not directly permit invalid state transitions or loss of funds.

If multiple nodes use the same RPC, the attack scales accordingly.

Severity

Low.

Technical Details

After Espresso activation, `dataAndHashesFromTxs` invokes `CollectAuthenticatedBatches`. On a cache miss, this function calls `FetchReceipts` for each required block and `L1BlockRefByHash` while traversing the authentication lookback window. The production implementation passes these requests through the Ethereum client and the Celo `op-geth` remote procedure call client, which submits them with `http.Client.Do`. HTTPS and secure WebSocket endpoints therefore exercise the affected `Go TLS`, `HTTP/2`, and `X.509` functions.

`GO-2026-4870` permits a TLS 1.3 peer to send multiple post-handshake `KeyUpdate` messages in one record. The connection can deadlock and retain resources instead of respecting the request timeout.

`GO-2026-4918` permits an `HTTP/2` server to send a zero `SETTINGS_MAX_FRAME_SIZE` value. The client transport then enters an infinite loop while writing continuation frames, which can pin processor and connection resources.

`GO-2026-5037` permits a peer to supply an untrusted certificate with many DNS Subject Alternative Name entries. Hostname verification repeats work for every entry and can consume quadratic processor time before certificate chain validation rejects the certificate.

Each attack prevents the receipt or header request from completing as expected. Because those requests are required to assemble the authenticated batch set, derivation cannot advance beyond the affected L1 block.

Remediation

We recommend upgrading the build toolchain to at least Go `1.25.12` or a newer supported release, and updating `golang.org/x/net` to at least version `0.53.0`. Every affected node binary should be rebuilt and redistributed. In addition, a continuous integration check should be added to reject unsupported Go releases and run `govulncheck` against production commands so future standard library vulnerabilities are identified before release.

Status

The Espresso team has resolved the issue in Commit `6e0f651` and PR 481.

Verification

Resolved.

Issue B: Enclave-Hash Revocation Is Reachable at a Lower Governance Threshold Than Intended (Out of Scope)

Location

This issue is located in the `espresso-tee-contracts` repository, which is outside the scope of this engagement:

`src/EspressoTEEVerifier.sol#L158`

Synopsis

The TEE verifier allows the owner or a guardian to mark an enclave hash `valid` or `invalid` via `setEnclaveHash`, and allows the owner to delete an enclave hash via `deleteEnclaveHashes`. `deleteEnclaveHashes` is deliberately restricted to the owner (a higher governance threshold). Its code comment states that deleting a hash "*breaks services that are using it, so it requires stricter governance than setting hashes.*"

However, calling `setEnclaveHash(hash, false)`, which is available to a guardian at the lower threshold, has the same effect. It marks the enclave hash `invalid`, which causes every signer registered under that hash to be treated as invalid. A single guardian can therefore revoke enclave signers and disrupt batch authentication through the lower-threshold function, bypassing the stricter governance that the delete path was intended to enforce.

Preconditions

The attacker must hold a valid guardian key.

Likelihood

Low.

The action requires a guardian key, so it can be performed only by a trusted guardian or through a compromised guardian key. In both cases, the required access is privileged and limited.

Impact

Low.

A guardian can invalidate an enclave hash and thereby revoke all TEE signers registered under it, disrupting batch authentication. The effect is recoverable because the owner can re-enable the hash, or governance can switch to the fallback batcher.

Severity

Informational.

Mitigation

We suggest monitoring for `setEnclaveHash` calls that set a hash to `false` and re-enabling the hash if required.

Remediation

We recommend restricting disabling an enclave hash to the owner and enforcing the owner check when `setEnclaveHash` is called with `valid == false`.

Status

The Espresso team has resolved the issue in `espresso-tee-contracts` PR #70.

Verification

Resolved.

Suggestions

Suggestion 1: Clarify REQ-CONTRACT-SAF-002 to Reflect the Post-Deactivation Decay-Window

Location

`Specification: REQ-CONTRACT-SAF-002 (and its AC-1)`

Synopsis

`REQ-CONTRACT-SAF-002` states that after `setActiveIsEspresso` deactivates a batcher, "*in-flight batch-data transactions from the now-deactivated batcher are dropped by derivation, leaving no window in which two batchers are simultaneously effective.*"

However, the derivation pipeline does not enforce this. In both the Go and Rust implementations, a payload is authorized if (1) its commitment appears in a `BatchInfoAuthenticated` event within the lookback window (`BATCH_AUTH_LOOKBACK_WINDOW`, 100 L1 blocks) and (2) the data transaction's sender equals the event's caller. Neither implementation consults the batcher's activation state. Consequently, a payload authenticated by the Espresso batcher before it is deactivated remains valid, and will be derived, if its data transaction is posted within the lookback window after deactivation. The two implementations are consistent with each other. The discrepancy lies between the specification text and the implementations.

There is therefore a bounded window of up to `BATCH_AUTH_LOOKBACK_WINDOW` (approximately 20 minutes) during which the deactivated batcher's pre-authenticated payloads and the newly active fallback batcher's payloads are both effective, contrary to the "no window" claim.

The implemented behavior is the appropriate design. Gating derivation on the authentication event, rather than on activation state, is consistent with `REQ-PIPELINE-M-001`, which requires the pipeline to authorize by scanning receipts rather than reading contract state. Determining the batcher's activation state at the moment a data transaction was posted would require reading contract state, and reconstructing it from `BatcherSwitched` events would require an unbounded scan.

Two additional points should be considered. First, `AC-1` is ambiguous. Its `Arrange` describes a payload that was authenticated and then had its batcher deactivated. However, its `Assert` addresses only data "whose authentication reverted," which is a different scenario, and does not test the case described by the requirement's headline. Second, the requirement refers to an authenticated "batch," but the unit actually authenticated is the payload of a single inbox transaction (a set of frames), bound to those exact bytes by the commitment.

Mitigation

We recommend revising `REQ-CONTRACT-SAF-002` to state that authentications emitted before deactivation would continue to be honored by derivation until their events leave the lookback window (100 L1 blocks). The revised requirement should also clarify that deactivation would prevent new authentications because subsequent `authenticateBatchInfo` calls from the now-deactivated batcher would revert without emitting an event, but would not retroactively invalidate authentications already emitted within the window.

Status

During the engagement, the specification author provided our team with a revised `REQ-CONTRACT-SAF-002` that is consistent with the implementations.

Verification

Resolved.

Suggestion 2: Update 3.3.2.1.5 to Attribute Espresso Authorization to the Blob Data Source Only

Location

`Specification 3.3.2.1.5`

Synopsis

Section 3.3.2.1.5 attributes the authorization check and commitment computation to "both the calldata source and the blob data source." After PR #466, both reside only in the blob data source, which handles calldata-carrying and blob-carrying transactions post-Ecotone.

Mitigation

We recommend updating Section 3.3.2.1.5 by replacing "both the calldata source and the blob data source" and "both data sources" with "the blob data source" and removing "identically," which no longer applies with a single source.

Status

The Espresso team has updated the specification.

Verification

Resolved.

Suggestion 3: Update the Specification to Describe the Lookback Window as a Fixed Constant, Not a Configurable Value

Location

- `REQ-PIPELINE-F-001`: configured lookback window
- 3.3.2.1.5, Key properties: default lookback of 100 blocks

- Variability Guide: event lookback window | block count (default 100)
- Decision Log: configurable event lookback window (default 100 L1 blocks)
- Go implementation: op-node/rollup/derive/params.go#L32
- Rust implementation: crates/kona/genesis/src/rollup.rs#L15

Synopsis

Several sections of the specification describe the lookback window as configurable. However, in both implementations, it is a hardcoded compile-time constant of 100.

Mitigation

We recommend updating the specification to describe the lookback window as a fixed consensus constant.

Status

The Espresso team has updated the specification.

Verification

Resolved.

Suggestion 4: Support EIP-4844 Blobs in the Celo Fault Proof Host

Location

`celo-kona/crates/kona/derive/src/blobs.rs#L93-L159`

`celo-kona/crates/kona/derive/src/blobs.rs#L201-L244`

`optimism/rust/kona/crates/proof/proof/src/l1/blob_provider.rs#L39-L56`

`celo-kona/bin/host/src/single/handler.rs#L122-L126`

Synopsis

The Celo derivation pipeline accepts authenticated EIP-4844 batch transactions, while the Celo fault proof host rejects the `L1Blob` required to retrieve their blob contents. A batch accepted by full nodes can therefore stop proof execution at the same L1 block, preventing the proof system from reproducing the accepted chain.

Specifically, the `CeloBlobSource` recognizes `TxEnvelope::Eip4844`, applies batch authorization, records each blob versioned hash, and passes the hashes to `get_and_validate_blobs`. In the fault proof client, `OracleBlobProvider::get_blob` sends an `L1Blob` hint so that the host prepares the commitment and field element preimages.

The Celo host handles the same `L1Blob` hint by returning an error based on the assumption that Celo uses only `EigenDA` or `calldata`. The client and host therefore enforce different data availability policies: the client accepts the batch and requests its blob, but the host cannot provide the data required to continue proof execution.

This behavior conflicts with the OP Stack fault proof specification, which requires hosts to process `l1-blob` hints for EIP-4844 verification. It also conflicts with the Celo Integration Specifications section 3.3.2.1.5, `REQ-PIPELINE-F-001`, which applies authentication to blob batches, and `REQ-SIDECAR-F-002`, which requires support for signing and submitting EIP-4844 batch transactions.

An authorized submitter can trigger the issue by posting a valid blob batch. Standard derivation would accept the transaction, but proof execution would stop when the host rejects the resulting blob request.

Mitigation

We recommend implementing `L1Blob` handling in the Celo host. The handler should retrieve the requested blob by versioned hash and L1 timestamp, verify it, and populate the commitment and field element preimages expected by `OracleBlobProvider`. An integration test should also be added that submits an authenticated blob batch and completes fault proof execution for the resulting chain segment.

Status

The Espresso team has prepared a solution in PR 482 and PR 289.

Verification

Resolved.

Suggestion 5: Establish a Single Source of Truth for the Espresso Configuration

Location

`celo-kona/crates/kona/proof/src/boot.rs#L214`

`optimism/op-node/rollup/types.go#L183`

Synopsis

The Espresso activation parameters (`espresso_time` and `batch_authenticator_address`) exist independently in two locations, embedded into the Rust proof program and read at runtime by the Go `op-node` from `rollup.json`. This poses a potential operational risk because, if the parameters do not agree across the two pipelines, the derivation pipelines would produce divergent outputs.

Mitigation

We recommend introducing a single canonical source of truth for the Espresso parameters, keyed by chain ID, that both the `op-node` configuration and the prover's embedded constants derive from or are validated against. The validation could be a build-time assertion that the embedded constants match the canonical file.

Status

The Espresso team has chosen not to implement this suggestion.

Verification

Unresolved.

Suggestion 6: Develop a Playbook and/or Tools to Verify Espresso Configuration After Deployment

Synopsis

Activating Espresso is a multi-step, cross-component change involving `rollup.json`, the prover's embedded constants, a recompiled prover, a new verification key, and the on-chain verifier, coordinated manually. A configuration or sequencing mistake during activation could cause the two derivation pipelines to diverge. This is an operational risk rather than a code defect. After deployment, the chain

operator (the party running the rollup's sequencer, proposer, batcher, prover, and contract deployments) should verify that the Espresso configuration is consistent across the components it controls.

Mitigation

We recommend developing a playbook and/or tools to verify the configuration on the consensus-critical components that the chain operator deploys and runs. This playbook should include:

- Reading the on-chain `programVKey` from the verifier, recompiling the prover, and confirming that the keys match (assuming reproducible builds), then checking `espresso_time` and `batch_authenticator_address` against the canonical source-of-truth file if they match, as described in Suggestion 5.
- Verifying that the `BatchAuthenticator` contract is correctly deployed, including the proxy and implementation, at the configured authenticator address.
- Querying the sequencer's `op-node` and any verifier nodes within the chain operator's control through `optimism_rollupConfig`, and verifying `espresso_time` and `batch_authenticator_address` against the canonical source-of-truth file.

Status

The specification documents an activation and upgrade procedure (`DEC-op-011`) and a deploy-time variability guide. It does not, however, describe the post-deployment verification steps or tooling to confirm that the Espresso configuration is consistent across the prover, the `BatchAuthenticator` contract, and the op-nodes.

Verification

Partially Resolved.

Suggestion 7: Add Automated Parity Checks for Duplicated Upstream Kona Derivation Code

Location

`celo-kona/crates/kona/derive/src/blob_data.rs#L1-L6`

`celo-kona/crates/kona/derive/src/blobs.rs#L1-L4`

`celo-kona/crates/kona/derive/src/ethereum.rs#L1-L10`

`optimism/rust/kona/crates/protocol/derive/src/sources/blob_data.rs`

`optimism/rust/kona/crates/protocol/derive/src/sources/blobs.rs`

`optimism/rust/kona/crates/protocol/derive/src/sources/ethereum.rs`

Synopsis

The `celo-kona` derivation implementation copies or closely mirrors upstream Kona blob decoding and data source logic because the required upstream definitions are not publicly reusable. The copies do not automatically receive upstream correctness or security fixes. A future dependency update could change the upstream derivation path while leaving the Celo implementation unchanged. Because derivation must remain deterministic across node and fault proof implementations, implementation drift could produce inconsistent handling of `calldata`, blobs, resets, or errors.

This is a maintenance risk. At the reviewed revision, our team identified no mismatches.

Mitigation

We recommend documenting the exact upstream revision and source locations for each copied component. Continuous integration checks should be added that compare the copies with the pinned upstream version, and an explicit parity review should be required whenever the Kona dependency changes. Equivalence tests should also be added for `calldata` processing, blob decoding, reset behavior, error handling, and Espresso authenticated batches. We also recommend contributing reusable extension interfaces upstream so that Celo can integrate Espresso authentication without maintaining independent copies of consensus-critical derivation logic.

Status

The Espresso team has chosen not to implement this suggestion.

Verification

Unresolved.

Suggestion 8: Pin and Continuously Monitor Integration Dependencies

Location

`celo-kona/Cargo.toml#L290-L295`

`celo-kona/Cargo.lock#L2989-L2996`

`optimism/go.mod#L286-L287`

`optimism/.github/dependabot.yml#L1-L14`

`hokulea/.github/dependabot.yml#L12-L22`

`hokulea/canoe/verifier/Cargo.toml#L13-L20`

`hokulea/Cargo.lock#L1465-L1469`

`hokulea/Cargo.lock#L10109-L10127`

Synopsis

The reviewed integration has several dependency maintenance risks that share the absence of unified dependency governance controls:

- Celo Kona selects an Alloy overflow fix through a movable Git branch.
- The Go derivation code resolves `go-ethereum` imports to a Celo `op-geth` fork while routine Go module version-update pull requests are disabled.
- The Rust client carries separate Reth and REVM stacks for Celo execution and Hokulea certificate verification.

These conditions increase the chance that lockfile regeneration, missed upstream backports, or uneven upgrades introduce unreviewed behavior or leave a component behind on maintenance fixes.

Mitigation

We suggest the following measures to mitigate this issue:

- Pinning the Alloy patch to a full commit revision and requiring locked dependency resolution in release and proof builds.
- Maintaining an inventory of forked dependencies and parallel dependency versions that records each upstream base, required patches, and backport status.

- Enabling scheduled Go module updates, narrowing Hokulea's exclusion of Cargo patch and minor updates, and running dependency advisory checks against the production feature graphs in continuous integration.
- Treating the two Reth and REVM stacks as an explicit compatibility risk, preventing unexpected additional versions, testing equivalent behavior at that boundary, and defining when the older verification stack can be retired.

Status

The Espresso team has chosen not to implement this suggestion.

Verification

Unresolved.

About Least Authority

Founded in 2011 and based in Berlin, Germany, since 2016, Least Authority provides security consulting for privacy-preserving technologies and decentralized systems. We are committed to transparency, openness, and advancing secure digital infrastructure.

We believe that people have a fundamental right to privacy and that secure solutions enable people to use the internet and other connected technologies more freely. Our security consulting services help teams make their solutions more resistant to unauthorized access and unintended system manipulation. We support teams from the design phase through production launch and beyond.

Our team has experience reviewing code for security vulnerabilities and specific attack vectors in multiple languages, including Rust, Go, Python, C, C++, Solidity, Cairo, Circom, Noir, Haskell, OCaml, TypeScript, Swift, ZoKrates, and others. We have reviewed implementations of cryptographic protocols and distributed system architecture in blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, we use various tools to scan code and networks and build custom tools as necessary.

For more information about our security consulting, please visit:
https://leastauthority.com/security-consulting.

Our Methodology

We use a transparent review process that provides the project team visibility into our findings, assumptions, and remediation considerations throughout the audit. The goals of our security audits are to improve the quality of the systems we review and support sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Code Review

In manually reviewing all of the code, we identify potential issues, including those related to code logic, error handling, cryptographic implementations, and correctness with respect to specifications, standards, and best practices. We also evaluate areas where more defensive programming could reduce the risk of future mistakes and accelerate future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

When appropriate, we also use artificial intelligence (AI) to support the efficiency of our security audits without reducing rigor or reliability. AI-assisted outputs are carefully guided, reviewed, and validated by our team so that conclusions remain accurate and defensible. This supports stronger audit decisions, not only faster review.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and white box penetration testing, supplemented by various tools, including AI, that support the code review process. Discussions with the development team help clarify their vision for the software. Installing and using the relevant software helps us understand user interactions and roles, while also informing our analysis of threat models and attack surfaces. Design documentation, relevant papers, other audit results, similar projects, source code dependencies, and open issue tickets may also be reviewed to inform the analysis. We identify potential vulnerabilities, and, when appropriate, create issue entries. For each entry, we follow the issue investigation and remediation process described below.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and supporting successful remediation. Whenever a potential issue is identified, we raise it with the client through a dedicated secure communication channel to discuss its validity and impact, to the extent these can be determined at that stage. This process allows potentially relevant findings to be reviewed with the client early, before they are documented in greater detail in the initial audit report. For each confirmed issue, we assess impact and feasibility and use those ratings to determine the overall severity, following the OWASP Risk Severity Matrix.

Suggested Solutions

We identify immediate and comprehensive mitigations available to live deployments and then suggest remediation requirements for future releases. When code is already deployed in production, we include mitigation strategies in the report to provide interim support until appropriate remediation is available. The mitigation and remediation recommendations should be reviewed by the development team and deployment engineers. After delivering the Initial Audit Report, we remain available to support implementation of the proposed mitigations and remediations before the verification review.

Development teams should address findings in an order and with an urgency that prioritizes the security of their users. Findings should be addressed as soon as possible, but remediation should be carried out pragmatically and should not cause unnecessary disruption to project timelines.

Although issues must be handled on a case-by-case basis, we work toward a resolution timeline that balances the impact on users and the needs of the project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess whether the issues and suggestions have been sufficiently addressed. When this analysis is complete, we update the report and provide a Final Audit Report that can be published in full. If any critical issues remain, we recommend that the Final Audit Report not be published and that users and other stakeholders be alerted to the impact. We encourage all findings to be addressed and the Final Audit Report to be shared publicly to support transparency and advance security knowledge within the industry.