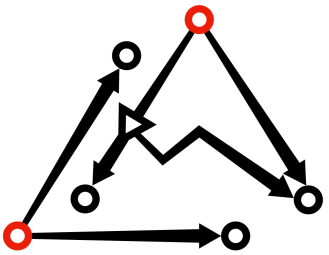

Least Authority

PRIVACY MATTERS

TEE Smart Contracts

Security Audit Report

Espresso

Final Audit Report: 1 April 2026

Table of Contents

Overview

Background

Espressosys has requested that Least Authority perform a security audit of its smart-contract layer, focusing on the Solidity contracts that verify Nitro TEE attestations and enforce batch authentication for Optimism integration.

Project Dates

- **February 18, 2026 - February 24, 2026:** Initial Code Review *(Completed)*
- **February 26, 2026:** Delivery of Initial Audit Report *(Completed)*
- **31 March, 2026:** Verification Review *(Completed)*
- **1 April, 2026:** Delivery of Final Audit Report *(Completed)*

Review Team

- Jasper Hepp, Security / Cryptography Researcher and Engineer
- Dominic Tarr, 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 smart contracts followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

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

- `espresso-tee-contracts` v1.1.0:
  - https://github.com/EspressoSystems/espresso-tee-contracts/blob/v1.1.0/src/EspressoNitroTEEVerifier.sol
  - https://github.com/EspressoSystems/espresso-tee-contracts/blob/v1.1.0/src/EspressoTEEVerifier.sol
  - https://github.com/EspressoSystems/espresso-tee-contracts/blob/v1.1.0/src/OwnableWithGuardiansUpgradeable.sol
  - https://github.com/EspressoSystems/espresso-tee-contracts/blob/v1.1.0/src/TEEHelper.sol

- `optimism-espresso-integration` v0.6.0:
  - https://github.com/EspressoSystems/optimism-espresso-integration/blob/v0.6.0/packages/contracts-bedrock/src/L1/BatchAuthenticator.sol
  - https://github.com/EspressoSystems/optimism-espresso-integration/blob/v0.6.0/packages/contracts-bedrock/src/L1/BatchInbox.sol

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

- `espresso-tee-contracts` v1.1.0:
  - `6b68d1a6fd356527aab9b03d476fede67c375eea64`

For the verification, we examined the following Git revision:

- `espresso-tee-contracts` v1.1.0:
  - `5e39b27cc7f0cee6cd9101ebf6f53ebd9915e7e8`

- `optimism-espresso-integration` v0.6.0:
  - `cd4c11a9d4756a56bc2e25e20bd3b34304e89fc9`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- `espresso-tee-contracts`: https://github.com/LeastAuthority/espresso-tee-contracts
- `optimism-espresso-integration`: https://github.com/LeastAuthority/optimism-espresso-integration

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:

- Website: https://www.espressosys.com
- Espresso Wiki: https://eng-wiki.espressosys.com
- To Be EXT - EspressoTEEVerifier Contract Governance - Celo: https://www.notion.so/espressosys/To-Be-EXT-EspressoTEEVerifier-Contract-Governance-Celo-3112431b68e980748248f8969d118475

In addition, this audit report references the following documents:

- EIP 712: https://eips.ethereum.org/EIPS/eip-712

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;
- Key management, including secure private key storage and management of encryption and signing keys;
- 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;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Espresso's TEE Contracts provide a mechanism to bridge Trusted Execution Environment outputs to the blockchain. This enables developers to execute complex code in a secure and trusted environment and receive an attestation of the output's correctness.

The audited contracts verify AWS Nitro Enclave attestations on-chain to register approved signers that enforce Espresso-confirmed batch posting for optimistic rollups. Because native verification of Nitro attestation certificate chains is prohibitively expensive on the EVM, verification is delegated to a zero-knowledge proof generated by a Succinct co-processor and verified through `Automata Network`'s contracts. The scope also included the OP Stack integration contracts, which authenticate batches by verifying ECDSA signatures against these TEE-registered signers. This batch-enforcement mechanism is orthogonal to dispute-resolution mechanisms, including fault proofs, and can be combined with them.

System Design

Espresso's TEE Smart Contracts are implemented as multiple layers of contracts and proxies. `EspressoTEEVerifier` provides a consistent interface and invokes `EspressoSGXTEEVerifier` and `EspressoNitroTEEVerifier`. These two contracts contain the service registration logic, which differs between `Nitro` and SGX. For verification of a trusted enclave run, they invoke third-party contracts provided by `Automata`, namely `V3QuoteVerifier` and `NitroEnclaveVerifier`, respectively. Proxies are deployed in front of `EspressoTEEVerifier`, `EspressoSGXTEEVerifier`, and `EspressoNitroTEEVerifier`, increasing the architecture from three layers to six. The inner layer, consisting of `EspressoSGXTEEVerifier` and `EspressoNitroTEEVerifier`, maintains a reference back to `EspressoTEEVerifier`, which complicates deployment by requiring an additional transaction to update `EspressoTEEVerifier` with references to the inner layer. We recommend simplifying the verifier architecture by removing redundant upgrade paths (Suggestion 6).

We systematically reviewed the Espresso TEE contracts across signature and cryptographic correctness, various replay-attack classes, access control and privilege persistence paths, TEE attestation integrity, upgradeability and initialization safety, batch authentication logic, and external dependency risks.

We found that the `BatchAuthenticator` contract does not follow the EIP-712 standard (Issue B), which allows for replay attacks across contexts. According to the Espresso team, replay attacks across time are mitigated by the derivation pipeline. We suggest documenting this assumption (Suggestion 5).

We also examined access control and privilege persistence. The `BatchAuthenticator` and `EspressoTEEVerifier` contracts introduce a parallel access-control authority plane that can diverge from ownership, which is the root cause of the two risk scenarios (Issue A). In addition, we assessed alignment between the code and the documentation of the `EspressoTEEVerifier` contract governance model and identified two minor deviations (Suggestion 7).

Moreover, we evaluated the `BatchAuthenticator` contract against the `SystemConfig` contract from Optimism (see documentation). We found that `BatchAuthenticator` implements its own guardian controls and does not inherit Optimism's `SuperchainConfig` `pause` gating (Suggestion 8). Additionally, Issue A highlights a deviation between the two configurations.

Finally, our team observed that not all functions in the verifiers emit events (Issue C).

Dependencies

The contracts inherit from several `OpenZeppelin` contracts and depend on external contracts maintained by `Automata`. `OpenZeppelin` is a well-established and widely used library suite, but it may introduce features that are not strictly necessary (Issue A). Dependence on external contracts also exposes Espresso to the risk that changes by `Automata` could disrupt Espresso's contracts. Espresso has included a method to update the addresses of the Automata contracts, allowing them to be redeployed and EspressoTEEVerifier to be redirected if necessary.

Code Quality

We performed a manual review of the repositories in scope and found the code to be well organized and of high quality. However, we identified a few areas for improvement (Suggestion 2, Suggestion 3, and Suggestion 4) by comparing the contracts against established best practices.

Tests

The repositories in scope include comprehensive test coverage. During the review, our team expanded coverage for the Espresso TEE contracts and added several property-based and fuzzing tests to supplement the assessment. None of these additional tests revealed any issues.

Documentation and Code Comments

The project documentation provided by the Espresso team was generally sufficient in describing the intended functionality of the system and proved helpful overall, although certain areas would benefit from additional depth. Moreover, the codebase is well commented and aided our understanding of most components. However, we recommend minor improvements to the NatSpec documentation (Suggestion 1).

Scope

The scope of this review was sufficient and included all security-critical components.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. 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: Split Role Administration Creates Two Risk Scenarios	High	Resolved
Issue B: Missing Domain Separation in Batch Authentication Signatures Allows Replay Across Verification Contexts	High	Resolved
Issue C: Missing Events for Sub-Verifier Address Updates	Low	Resolved
Suggestion 1: Close NatSpec Documentation Gaps	Informational	Resolved
Suggestion 2: Remove Unused using EnumerableSet Directive in SGX and Nitro Verifiers	Informational	Resolved

Suggestion 3: Remove Redundant signer != address(0) Check After ECDSA.recover	Informational	Resolved
Suggestion 4: Update deleteEnclaveHashes Behavior for Non-Registered Hashes	Informational	Resolved
Suggestion 5: Document Derivation Pipeline Trust Model for Time-Based Commitment Replay Attacks	Informational	Resolved
Suggestion 6: Reduce Number of Proxy Layers	Informational	Resolved
Suggestion 7: Align Governance Documentation and Access Control for deleteEnclaveHashes and setEnclaveHash	Informational	Resolved
Suggestion 8: Align Batch Admission Governance with Optimism's Pause Domain	Informational	Resolved

Issue A: Split Role Administration Creates Two Risk Scenarios

Location
`src/OwnableWithGuardiansUpgradeable.sol`

`src/EspressoTEEVerifier.sol`

`openzeppelin-contracts-upgradeable/contracts/access/AccessControlUpgradeable.sol`

`openzeppelin-contracts-upgradeable/contracts/access/extensions/AccessControlEnumerableUpgradeable.sol`

`ethereum-optimism/optimism/packages/contracts-bedrock/src/L1/SystemConfig.sol`

Synopsis
The `BatchAuthenticator` and `EspressoTEEVerifier` contracts introduce a parallel authority plane through access-control roles that can decouple from ownership. This root-cause divergence creates two distinct risk scenarios.

Impact
High.

An attacker may regain control over guardian-gated trust-root actions through stale admin privileges, and the owner may lose operational control of guardian management following role-admin divergence.

Feasibility
Medium.

This state can arise during routine administration, for example through delegated role grants or by calling `renounceRole`, and exploitation becomes straightforward once role-admin and ownership diverge.

Severity
High.

Preconditions

A non-owner account must hold the variable `DEFAULT_ADMIN_ROLE` before ownership transfer acceptance for the first risk scenario, or the owner must call the function `renounceRole(DEFAULT_ADMIN_ROLE, owner())` for the second risk scenario.

Technical Details

The `BatchAuthenticator` and `EspressoTEEVerifier` contracts both inherit the contract `OwnableWithGuardiansUpgradeable`, which in turn inherits `AccessControlEnumerableUpgradeable` and exposes the functions `grantRole`, `revokeRole`, and `renounceRole`, as well as the role-admin graph through the variable `DEFAULT_ADMIN_ROLE` and the function `getRoleAdmin`. This model differs from Optimism's `SystemConfig` contract, which inherits `OwnableUpgradeable` and does not expose this additional direct role-admin plane. The core issue is divergence between role-admin and ownership. Addresses other than the return value of the function `owner` can hold or remove the variable `DEFAULT_ADMIN_ROLE`.

Risk scenario one is stale-admin takeover after ownership transfer. The function `_transferOwnership` revokes the variable `DEFAULT_ADMIN_ROLE` only from the previous owner and grants it to the new owner but does not remove additional holders. As a result, a stale admin can call `grantRole` to self-assign the variable `GUARDIAN_ROLE` and call `revokeRole` to remove admin privileges from the new owner, thereby regaining effective control over sensitive trust-root actions.

Risk scenario two is owner lockout from guardian lifecycle management. If the owner calls the function `renounceRole(DEFAULT_ADMIN_ROLE, owner())`, subsequent calls to `addGuardian` and `removeGuardian` can revert even though the caller is the owner, because these wrappers invoke `grantRole` or `revokeRole`, both of which require active role-admin authority.

Remediation

We recommend replacing external mutable management of `DEFAULT_ADMIN_ROLE` with ownership-coupled admin-role management that enforces the invariant that only the address returned by `owner` holds `DEFAULT_ADMIN_ROLE`.

Status

The Espresso team has resolved the issue. The contract now inherits only `Ownable2StepUpgradeable` and manages guardians through a custom `EnumerableSet.AddressSet`.

Verification

Resolved.

Issue B: Missing Domain Separation in Batch Authentication Signatures Allows Replay Across Verification Contexts

Location

`src/EspressoTEEVerifier.sol#L55-L74`

`optimism-espresso-integration/packages/contracts-bedrock/src/L1/BatchAuthenticator.sol#L108`

`optimism-espresso-integration/packages/contracts-bedrock/src/L1/BatchAuthenticator.sol#L147`

Synopsis
The function `authenticateBatchInfo` accepts signatures over a raw hash without typed domain binding, allowing replay in any verification context that accepts the same signer and digest.

Impact
High.

An attacker may transfer authentication decisions across chains, contracts, or entry points when signer overlap exists, potentially pre-authorizing batch commitments outside the original context.

Feasibility
Medium.

Exploitation does not require privileged access but requires observable signatures and signer reuse across the source and target verification domains.

Severity
High.

Preconditions
For this issue to be possible, the following must be in place:

- The same signer address must be valid in more than one target context.
- A valid `(digest, signature)` pair must be observable and capable of being resubmitted by an attacker.

Technical Details
The function `authenticateBatchInfo` normalizes the recovery byte and then calls `ECDSA.recover(commitment, signature)` directly. The function `verify` in `EspressoTEEVerifier` follows the same raw digest recovery pattern. Neither path computes an EIP-712 typed-data digest, because no domain separator with `chainId` and `verifyingContract` is applied and no type-scoped `hashStruct` is applied. Signature validity is therefore bound only to the signer key and raw `bytes32` digest, not to a specific contract or chain.

A captured signature can be replayed to another authenticator instance that accepts the same signer and digest, thereby setting `validBatchInfo` for that digest in the target context. The function `verify` in `EspressoTEEVerifier` also uses raw digest recovery, so replay impact across entry points depends on signer and service overlap in the consuming integration.

Remediation
We recommend replacing raw digest recovery in `authenticateBatchInfo` and `verify` with EIP-712 typed-data verification. The implementation should recompute the digest on-chain from validated message fields, namely `commitment` or `userDataHash`, `teeType`, and `service`, using a domain separator that binds `chainId` and `verifyingContract`.

Status
The Espresso team has resolved the issue and implemented EIP-712 typed-data verification across the full signature path.

Verification
Resolved.

Issue C: Missing Events for Sub-Verifier Address Updates

Location
src/EspressoTEEVerifier.sol#L143-L167

src/EspressoTEEVerifier.sol#L55-L137

src/EspressoSGXTEEVerifier.sol#L203-L209

src/EspressoNitroTEEVerifier.sol#L119-L125

Synopsis
The functions `setEspressoSGXTEEVerifier` and `setEspressoNitroTEEVerifier` update core verifier addresses without emitting dedicated events.

Impact
Low.

A malicious owner or a compromised owner key may alter the verification route while reducing event-based detection and response visibility.

Feasibility
Medium.

Severity
Low.

Preconditions
For this issue to arise, the following must apply:

- An attacker must control the `owner` account.
- Operational monitoring must rely on emitted logs from the `entrypoint` contract rather than direct storage polling.

Technical Details
The functions `setEspressoSGXTEEVerifier` and `setEspressoNitroTEEVerifier` validate only a nonzero input and then write the new contract reference to storage, and neither function emits an event. These references are consumed by `verify`, `registerService`, `isSignerValid`, and `registeredEnclaveHashes`, so the write modifies the authentication trust boundary for both SGX and `Nitro` flows. In contrast, the related administrative paths `_setQuoteVerifier` and `_setNitroEnclaveVerifier` emit explicit events.

If an attacker obtains owner privileges, verification can be routed to an attacker-controlled contract by calling a setter function without a dedicated entry point event signaling that state transition. This attack path primarily degrades observability and incident response rather than bypassing authorization checks.

Remediation
We recommend replacing silent storage updates in `setEspressoSGXTEEVerifier` and `setEspressoNitroTEEVerifier` with updates that emit dedicated events including the previous and new verifier addresses.

Status

The Espresso team has resolved the issue by introducing events in the aforementioned functions.

Verification

Resolved.

Suggestions

Suggestion 1: Close NatSpec Documentation Gaps

Location

src/EspressoTEEVerifier.sol#L37-L46

src/TEEHelper.sol#L22-L27

src/TEEHelper.sol#L142-L148

optimism-espresso-integration/packages/contracts-bedrock/src/L1/BatchAuthenticator.sol#L108-L128

Synopsis

Several architecturally relevant functions have incomplete NatSpec documentation. The function `initialize` in `EspressoTEEVerifier` omits `@param` tags, the functions `_layout` and `_setTEEVerifier` in `TEEHelper` omit NatSpec blocks, and `authenticateBatchInfo` omits parameter documentation.

Mitigation

We recommend replacing partial NatSpec annotations with complete NatSpec blocks that include `@notice` and `@param` tags for `initialize`, `_layout`, `_setTEEVerifier`, and `authenticateBatchInfo`.

Status

The Espresso team has updated the NatSpec documentation as recommended.

Verification

Resolved.

Suggestion 2: Remove Unused using EnumerableSet Directive in SGX and Nitro Verifiers

Location

src/EspressoSGXTEEVerifier.sol#L15-L29

src/EspressoNitroTEEVerifier.sol#L12-L22

Synopsis

The contracts `EspressoSGXTEEVerifier` and `EspressoNitroTEEVerifier` declare `using EnumerableSet for EnumerableSet.AddressSet`, but neither contract uses the `EnumerableSet.AddressSet` type in local storage or function logic, leaving a redundant directive and import pair in both files.

Mitigation

We recommend removing the redundant `using EnumerableSet for EnumerableSet.AddressSet` declarations in the two verifier contracts and relying solely on the parent-contract guardian set implementation where needed.

Status

The Espresso team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 3: Remove Redundant signer != address(0) Check After ECDSA.recover

Location

optimism-espresso-integration/packages/contracts-bedrock/src/L1/BatchAuthenticator.sol#L117-L120

openzeppelin-contracts/contracts/utils/cryptography/ECDSA.sol#L191

openzeppelin-contracts/contracts/utils/cryptography/ECDSA.sol#L202

Synopsis

The function `authenticateBatchInfo` checks `signer != address(0)` after calling `ECDSA.recover`, but the OpenZeppelin ECDSA implementation already treats `address(0)` as an invalid signature and reverts, which renders the subsequent check unreachable.

Mitigation

We recommend removing the redundant `require(signer != address(0))` check in `authenticateBatchInfo` and relying on the failure semantics of `ECDSA.recover`.

Status

The Espresso team has resolved the suggestion by replacing the entire signature verification flow.

Verification

Resolved.

Suggestion 4: Update deleteEnclaveHashes Behavior for Non-Registered Hashes

Location

src/TEEHelper.sol#L120-L134

src/EspressoTEEVerifier.sol#L194-L204

Synopsis

The function `deleteEnclaveHashes` validates each input hash with `require($.registeredEnclaveHashes[service][enclaveHash], "Enclave hash not registered")` before deletion. If any hash in the batch is not currently registered, the full call reverts,

rendering administrative deletion non-idempotent when state changes between preparation and execution.

Mitigation
We recommend replacing hard revert behavior for already unregistered hashes in `deleteEnclaveHashes` with skip semantics for missing entries so that batch deletion remains idempotent.

Status
The Espresso team has implemented the mitigation as recommended.

Verification
Resolved.

Suggestion 5: Document Derivation Pipeline Trust Model for Time-Based Commitment Replay Attacks

optimism-espresso-integration/packages/contracts-bedrock/src/L1/BatchAuthenticator.sol#L126-L185

optimism-espresso-integration/op-node/rollup/derive/data_source.go#L109-L137

optimism-espresso-integration/op-node/rollup/derive/batches.go#L82-L111

Synopsis
The function `authenticateBatchInfo` in `BatchAuthenticator` stores commitments in the variable `validBatchInfo` as append-only authorization state, while replay and freshness controls are enforced in the derivation pipeline through receipt-status filtering and batch-validity checks. While no issue was identified with this architecture, the trust boundary is implicit and may be misinterpreted if the dependency between the contract layer and derivation layer is not stated explicitly.

Mitigation
We recommend documenting the implicit cross-component assumption between the contracts and the derivation pipeline.

Status
The Espresso team has implemented the mitigation as recommended.

Verification
Resolved.

Suggestion 6: Reduce Number of Proxy Layers

Location
`scripts/DeployAllTEEVerifiers.s.sol`

Synopsis
Espresso's Tee Smart Contracts employ multiple updatable-contract techniques concurrently. This approach is unnecessary, increases deployment complexity, and introduces the risk of misconfiguration, for example by allowing the top-level contract to reference invalid subcontracts.

Technical Details

`EspressoTEEVerifier` maintains references to the subcontracts (`EspressoNitroTEEVerifier` and `EspressoSGXTEEVerifier`), which manage the service registration logic, and also provides methods to update these references to different contracts (`setEspressoSGXTEEVerifier` and `setEspressoNitroTEEVerifier`). This allows a new subcontract to be deployed and its address updated in `EspressoTEEVerifier` if a bug is identified. However, the subcontracts are also deployed behind proxy contracts (`TransparentUpgradableProxy`), which allow a subcontract implementation to be replaced and the proxy redirected without changing the proxy address. Accordingly, maintaining both a proxy layer and address-update methods is redundant.

Mitigation

We recommend removing either the proxy layer or the address-change methods. Preferably, `EspressoTEEVerifier`, `EspressoSGXTEEVerifier`, and `EspressoNitroTEEVerifier` could be consolidated into a single updatable contract. None of the contracts are large, and the two subcontracts differ only in their `registerService` method. Because the top-level contract also exposes methods from the lower-level contracts while wrapping their functions, combining them would slightly reduce code size and gas usage, eliminate the risk of misconfiguration, and simplify deployment.

Status

The Espresso team has addressed the suggestion by removing the proxy layer.

Verification

Resolved.

Suggestion 7: Align Governance Documentation and Access Control for deleteEnclaveHashes and setEnclaveHash

Location

`src/EspressoTEEVerifier.sol#L176-L185`

`src/EspressoTEEVerifier.sol#L194-L205`

`https://www.notion.so/espressosys/To-Be-EXT-EspressoTEEVerifier-Contract-Governance-Celo-3112431b68e980748248f8969d118475`

Synopsis

The governance document describes the functions `setEnclaveHash` and `deleteEnclaveHashes` as "non-public functions," but both are declared `external` and callable by any address that satisfies the `onlyGuardianOrOwner` modifier. This language refers to access restriction rather than Solidity visibility. In addition, a comment-thread amendment in the governance document records agreement to move `deleteEnclaveHashes` to `onlyOwner`; however, the code retains `onlyGuardianOrOwner` for both functions. In governance terms, owner actions require a 2/2 approval threshold, while guardian actions require a 1/2 threshold. Because `deleteEnclaveHashes` remains callable by guardians, a single guardian side can execute enclave-hash revocation through the 1/2 guardian path even when the other side does not approve.

Mitigation

We recommend updating the documentation and the code accordingly.

Status

The Espresso team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 8: Align Batch Admission Governance with Optimism's Pause Domain

Location

ethereum-optimism/optimism/packages/contracts-bedrock/src/L1/SystemConfig.sol#L604

packages/contracts-bedrock/src/L1/BatchInbox.sol#L23

packages/contracts-bedrock/src/L1/BatchAuthenticator.sol#L85

packages/contracts-bedrock/src/L1/BatchAuthenticator.sol#L210

src/OwnableWithGuardiansUpgradeable.sol#L81-L86

Synopsis

We identified the following deviation when comparing the `BatchAuthenticator` contract with Optimism's `SystemConfig` contract, referred to as OP below:

In the upstream OP reference implementation, the functions `guardian` and `paused` in `SystemConfig` are read-throughs to `SuperchainConfig`, such that guardian authority and `pause` state are defined within a single external control plane. This pattern is not enforced in `BatchInbox` or `BatchAuthenticator`. By contrast, `BatchInbox` forwards directly to `validateBatch`, and `validateBatch` does not check `SystemConfig.paused`. `BatchAuthenticator` enforces admission through `validateBatch`, `validateTeeBatch`, and `validateNonTeeBatch`, none of which reference `SystemConfig.paused`. As a result, the OP `pause` state is not directly enforced at this admission gate.

Mitigation

We recommend documenting and clarifying this deviation.

Status

The Espresso team has addressed the suggestion.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

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

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up 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.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.