Espresso Optimism Integration

Customer:
Espresso Systems
2025-07-31
v1.0

Contact:
Jarred Parr, 0xkato
jarred@espressosys.com

Table of Contents

Executive Summary

This penetration testing engagement assessed the security posture of the Espresso Systems Global Confirmation Layer (GCL) integration with the Optimism Layer 2 blockchain infrastructure. Espresso Systems provides decentralized sequencing and accelerated transaction finality services, enabling faster confirmation times compared to traditional Ethereum Layer 1 settlement mechanisms.

The assessment scope encompassed the complete integration stack, including core streaming infrastructure, transaction batching services, blockchain deployment tools, node operations, and supporting cryptographic services. Our testing methodology combined static code analysis, dynamic security testing, and architectural security review to evaluate the integration's security controls and implementation quality.

Key findings from this assessment include 4 low-risk, with only one high-risk and no critical vulnerabilities identified. This finding profile demonstrates a mature security implementation with appropriate controls for a complex blockchain infrastructure integration. The development team has demonstrated strong security practices throughout the integration lifecycle, particularly in cryptographic implementations and service-to-service communications.

The Espresso-OP integration presents a robust security posture suitable for production deployment. All identified findings represent opportunities for security hardening and best practice implementation rather than fundamental security flaws. The absence of critical vulnerabilities indicates that the core security architecture effectively protects against common attack vectors while maintaining the performance and reliability requirements of a high-throughput blockchain system.

We recommend addressing the identified findings through the standard development cycle and implementing the security recommendations outlined in the detailed findings section to further strengthen the overall security posture of the integrated platform.

Strengthen Your Habits, Terminate Exploits.

Methodology and Scope

Scope Summary

Overview

This penetration testing engagement focused on the Espresso Systems integration with the Optimism (OP) Layer 2 blockchain infrastructure. Espresso Systems operates as a Global Confirmation Layer (GCL) that provides decentralized sequencing and fast finality for composable blockchain networks. The integration aims to enhance OP's transaction finality speed beyond traditional Ethereum Layer 1 checkpointing mechanisms.

Architecture Context

The Espresso GCL serves three primary functions within the assessed ecosystem:

Global Trust Layer: Acts as a universally trusted confirmation mechanism across participating l2s, with cryptoeconomic security backed by validator collateral and slashing conditions to prevent rollback attacks without significant economic penalties.

Accelerated Finality: Provides faster confirmation of blockchain input sequences compared to standard settlement layer processing, enabling reduced transaction finality times for OP network participants.

Cryptographic Verifiability: Generates succinct cryptographic proofs allowing efficient verification of confirmed inputs by any participating chain or external party.

Technical Scope

The assessment encompassed four primary components of the OP-Espresso integration:

Core Infrastructure Components

- **Espresso Streamer Module**: Real-time data streaming functionality with fuzzing coverage analysis revealing 70% code coverage across 345+ million execution cycles
- **Batch Buffer Implementation**: Transaction batching and buffering mechanisms for optimized throughput

OP-Batcher Integration

- **Configuration Management**: Batcher service configuration and parameter handling
- **Driver Logic**: Core batching orchestration and transaction processing workflows
- **Espresso-Specific Integration**: Custom integration logic for GCL communication protocols
- **Transaction Data Handling**: Input validation and data serialization processes

Supporting Services

- **Cryptographic Operations**: Espresso-specific cryptographic primitives and signature schemes
- **Transaction Management**: Transaction lifecycle management with GCL integration for fetching and verification
- **Service Configuration**: Command-line interface and configuration management utilities, including configuration management and checks

Assessment Methodology

The penetration testing methodology incorporated static code analysis, dynamic fuzz testing, and architectural review focusing on:

- Integration point security between OP and Espresso systems
- Input validation and data sanitization
- Inter-service communication security

Key Findings Summary

The assessment identified various security considerations and implementation details across the integrated codebase. All identified findings were classified as informational, low, or medium risk. **Notably, one high-risk and no critical vulnerabilities were discovered during this engagement.** The overall security posture of the Espresso-OP integration has very little risky code, and all found errors are logical flaws, and not failures of the architectural decisions as far as can be ascertained.

Vulnerability Overview

In the course of this penetration test **1 High** and **4 Low** vulnerabilities were identified:

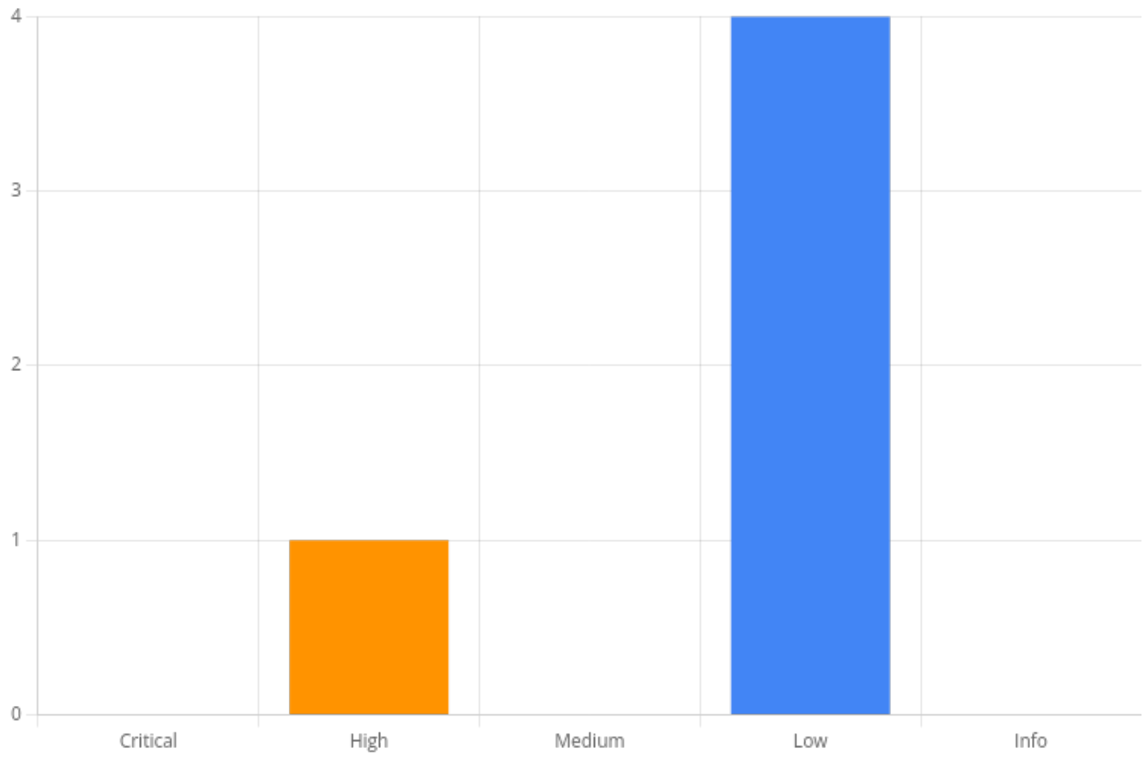


Figure 1 - Distribution of identified vulnerabilities

Vulnerability	**Criticality**
Changing loop iteratee during iteration	**High**
Brittle and inconsistent error checking	**Low**
Invalid length check	**Low**
Missing check for batch insertion on error.	**Low**
Multiple single points of failure with Query Service	**Low**

1. Changing loop iteratee during iteration

Criticality: High
CVSS-Score: 7.7
Affects: https://github.com/EspressoSystems/optimism-espresso-integration/blob/celo-integration-rebase-13.2/espresso/streamer.go#L352 **Recommendation:** Cache the keys and delete them all at once

Overview

This is a serious issue where the code modifies the map being iterated (`s.RemainingBatches`) by deleting keys during the range loop. In Go, modifying a map during range iteration can lead to unpredictable behavior, including skipped elements, panic conditions, or infinite loops. The `delete` operation occurs in multiple cases within the switch statement and also at the end of each iteration, making this a systematic problem that affects the reliability of batch processing. An example of this can be seen below:

```go
func main() {
        values := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
        fmt.Println("Original slice:", values)
        fmt.Println("Attempting to remove even numbers during iteration:")

        for i, v := range values {
                fmt.Printf("Processing index %d, value %d\n", i, v)
                if v%2 == 0 {
                        // Modify the slice during iteration, this will panic
                        values = append(values[:i], values[i+1:]...)
                        fmt.Printf("  Removed %d, slice is now: %v\n", v,
values)
                }
        }
        fmt.Println("Final slice:", values)
        fmt.Println()
}
```

Description

Go's range loop over maps creates an iterator at the start of the loop. When the underlying map is modified during iteration (via `delete` operations), the iterator's behavior becomes undefined. This can result in elements being skipped entirely, elements being processed multiple times, or runtime panics. The issue is compounded because deletion occurs in multiple code paths (both in switch cases and at the loop's end), making the iteration behavior highly unpredictable and potentially causing data loss or system instability

Recommendation

Collect keys to delete in a separate slice during iteration, then delete them after the loop completes, or use a different iteration approach that's safe for concurrent modification.

```
var keysToDelete []KeyType
for k, batch := range s.RemainingBatches {
    // ... processing logic ...
    keysToDelete = append(keysToDelete, k)
}
for _, k := range keysToDelete {
    delete(s.RemainingBatches, k)
}
```

2. Brittle and inconsistent error checking

Criticality: Low
CVSS-Score: 3.6
Affects:

- https://github.com/EspressoSystems/optimism-espresso-integration/blob/celo-integration-rebase-13.2/op-batcher/batcher/espresso.go#L237
- https://github.com/EspressoSystems/optimism-espresso-integration/blob/celo-integration-rebase-13.2/op-batcher/batcher/espresso.go#L248
- https://github.com/EspressoSystems/optimism-espresso-integration/blob/celo-integration-rebase-13.2/op-batcher/batcher/espresso.go#L346

Recommendation: Use better error types and explicit deserialization

Overview

This error handling mechanism exhibits extreme fragility through string-based error classification, creating a significant maintenance burden and potential for runtime failures. The function performs literal string matching against hardcoded error messages from Go's encoding/json package, which are implementation details subject to change across Go versions. This approach violates the principle of robust error handling in distributed systems where reliability is paramount. In a Web3 context where transaction processing failures can result in financial losses or network disruptions, such brittle error detection represents a critical reliability anti-pattern.

The current implementation only covers a subset of possible JSON marshaling errors, potentially misclassifying unknown JSON errors as retryable when they should be skipped. This could lead to infinite retry loops, resource exhaustion, and degraded system performance during edge cases involving malformed transaction data.

```go
if strings.Contains(msg, "json: unsupported type:") ||
    strings.Contains(msg, "json: unsupported value:") ||
    strings.Contains(msg, "json: error calling") ||
    strings.Contains(msg, "json: invalid UTF-8 in string") ||
    strings.Contains(msg, "json: invalid number literal") ||
    strings.Contains(msg, "json: encoding error for type") {
    log.Warn("json.Marshal fails, skipping", "msg", msg)
    return Skip
}
```

Description

The fundamental issue lies in the coupling between error handling logic and Go standard library implementation details. JSON marshaling errors in Go can manifest in numerous ways beyond the hardcoded strings, including custom type marshaling failures, circular reference errors, and numeric overflow conditions. The string-based approach creates a maintenance nightmare where Go version updates could silently break error classification, leading to incorrect retry behavior.

Recommendation

Implement a comprehensive error taxonomy system that classifies HTTP errors by their underlying cause and appropriate remediation strategy. Use Go's typed error system with custom error types for different failure modes.

3. Invalid length check

Criticality: Low
CVSS-Score: 3.6
Affects: https://github.com/EspressoSystems/optimism-espresso-integration/blob/celo-integration-rebase-13.2/op-batcher/batcher/service.go#L141 **Recommendation:** Change the length check to be > 1

Overview

The length check in the service init method is incorrect. Below is the snippet from the function.

```
	if len(cfg.EspressoUrls) > 0 {
		bs.EspressoPollInterval = cfg.EspressoPollInterval
		client, err :=
espressoClient.NewMultipleNodesClient(cfg.EspressoUrls)
```

However, the MultipleNodesClient requires a minimum number of URLs to be greater than 1, leading to failures through unintentionally forced user error.

```
func NewMultipleNodesClient(urls []string) (*MultipleNodesClient, error) {
	if len(urls) <= 1 {
		return nil, IncorrectUrlAmountErr
	}
	nodes := make([]*Client, len(urls))
	for i, url := range urls {
		nodes[i] = NewClient(url)
	}
	return &MultipleNodesClient{nodes: nodes}, nil
}
```

Description

The config value loads the urls from the user-sourced configuration. There needs to be more than 1 url, else it is not a `MultipleNodesClient`.

Recommendation

Simple change the length check to check for 1 instead.

4. Missing check for batch insertion on error.

Criticality: Low
CVSS-Score: 3.6
Affects: https://github.com/EspressoSystems/optimism-espresso-integration/blob/celo-integration-rebase-13.2/espresso/streamer.go#L322 **Recommendation:** Don't insert the BatchFuture value

Overview

The code handles an unexpected BatchFuture case by logging an error but continues execution without preventing the batch from being inserted. When `CheckBatch` unexpectedly returns `BatchFuture`, the code logs this as an error condition but doesn't halt processing or skip the problematic batch. This allows potentially invalid or unexpected batches to be inserted into the batch buffer, which could lead to inconsistent state or processing errors downstream.

```
case BatchFuture:
    // The function CheckBatch is not expected to return BatchFuture so if we enter this case there is a problem.
    s.Log.Error("Remaining list", "BatchFuture validity not expected for batch", batch)
    // Code continues to the batch buffer insert
```

Description

The switch statement lacks proper error handling for the unexpected `BatchFuture` case. After logging the error, execution falls through to the insertion logic outside the switch statement. This means batches that shouldn't exist according to the system's expectations are still processed normally, potentially corrupting the batch processing pipeline or causing undefined behavior in downstream components.

Recommendation

Add a `continue` statement after the error log to skip insertion of batches with unexpected validity states.

5. Multiple single points of failure with Query Service

Criticality: Low
CVSS-Score: 3.6
Affects:

- https://github.com/EspressoSystems/optimism-espresso-integration/blob/celo-integration-rebase-13.2/op-batcher/batcher/espresso.go#L609
- https://github.com/EspressoSystems/optimism-espresso-integration/blob/celo-integration-rebase-13.2/espresso/streamer.go#L296

Recommendation: Users should be logged out automatically after a certain period of inactivity.

Overview

The current architecture exhibits a critical single point of failure through its reliance on a singular Espresso Query Service instance for transaction hash lookups. This design creates multiple attack vectors and reliability issues that fundamentally compromise the deployment when interfacing with untrusted instances. A motivated attacker, given the open source nature of our code, could figure out the url we're querying, and target the single node to deny service to the OP instance.

Description

Querying a single instance of a service is problematic as it opens up an easy vector for denial of service. Since we're querying trusted nodes, this is less of an issue, but could become a problem if an attacker DoSes a node.

Recommendation

Implement a query system that polls multiple independent Espresso Query Services and uses some type of reconciliation method to determine if any are faulty (for example, take the majority response).

List of Changes

Version	Date	Description	Author
0.1	2025-07-30	Added the initial executive summaries and high-level report info.	Jarred and 0xkato
0.2	2025-07-30	Added the findings	Jarred and 0xkato
1.0	2025-07-31	Couple of typo fixes	Jarred

Disclaimer

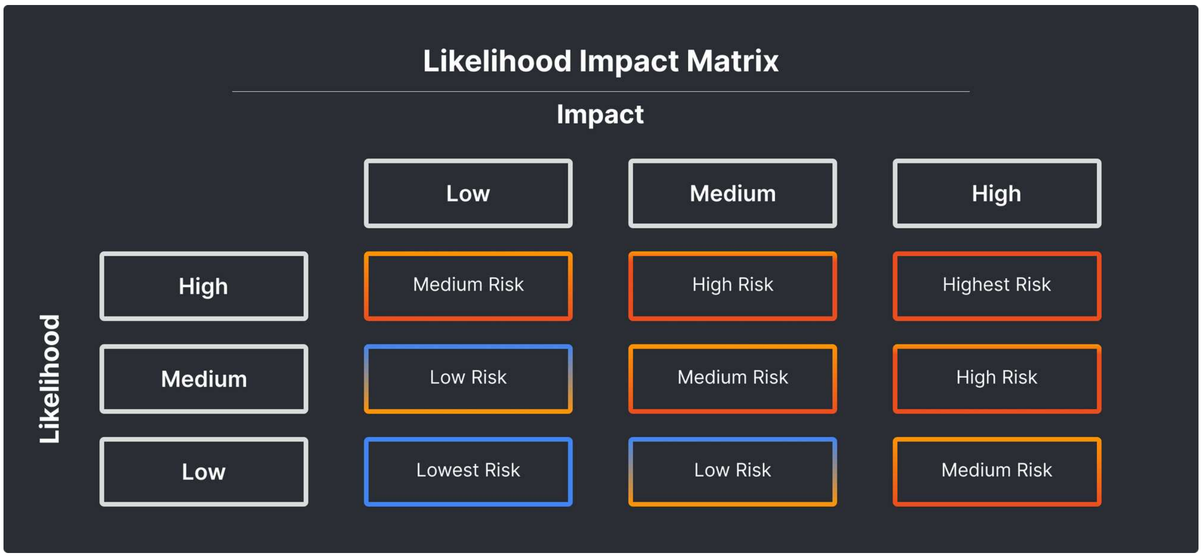
This report is a comprehensive review of in-scope items and should not be considered an endorsement or disapproval of any facet of the Espresso Systems software platform. It does not guarantee or provide any warranty regarding the absolute continuity of the analyzed technologies. Nor does it reflect the economics, value, business model, or legal compliance of such products. This report should not be used to make any investment decisions, nor does it constitute investment advice. This report serves as a comprehensive evaluation of the technology and constituent components, aimed at improving the overall security posture of the platform.

The Brewtforce Measurement Scale

The Brewtforce measures severity using in-house expertise along the The Brewtforce likelihood-impact matrix. Furthermore, we leverage industry standard methodologies as outlined in the Open Web Application Security Project (OWASP) and the Common Vulnerability Scoring System (CVSS).

The severity of each finding in this report is established after the conclusion of our rigorous testing, in accordance with the threat model outlined in the statement of work. We evaluate the severity of findings in isolation, irrespective of other findings. A vulnerability's overall business and technical impact is correlated to its score, where a higher score corresponds to higher business and technological impact, and lower scores are less severe.

Critical	Vulnerabilities that score “high” but additionally pose a severe business or security impact. Examples of this could be threats to human safety, major financial loss, or existential risks to operations. These require immediate remediation and are disclosed under the The Brewtforce 0-day disclosure policy.
High	Vulnerabilities that pose a significant risk to the business, such as unauthorized access to sensitive data or administrative functionality. These require prompt remediation but are not immediately existential to business operations.
Medium	Vulnerabilities that do not cause immediate harm but may be used to escalate privileges, pivot within a network, or gather information useful for future attacks. These warrant timely attention to reduce long-term risk.
Low	Vulnerabilities with minimal security impact that are unlikely to be exploited but may still improve overall posture if addressed. Examples include weak password policies, outdated libraries, or directory listing on web servers.
Informational	Issues that do not represent exploitable vulnerabilities but provide insight into the system, such as version disclosure, internal IP addresses, or verbose error messages. These may aid an attacker during reconnaissance but are not threats by themselves.



Likelihood Impact Matrix

Likelihood \ Impact	Low	Medium	High
High	Medium Risk	High Risk	Highest Risk
Medium	Low Risk	Medium Risk	High Risk
Low	Lowest Risk	Low Risk	Medium Risk

Imprint

The Brewtforce
Your Harddrive
The, Internet
United States of America