

Emerging C2 Threats in GenAI

SSE Abuse in MCP-Enabled Systems

Matthew Schwartz

Contents

Why This Is a Concern	1
How SSE-Based Malware Works	2
1. Why Malware Can Successfully Use SSE	2
2. How SSE Could Be Abused for Command and Control	2
Detection Strategies — Challenges and Solutions	2
Actionable Detection Strategy	2
Key Indicators of Malicious SSE Behavior (With Detection Actions)	4
SSE vs HTTP/3 Server Push — Why SSE Is Still a Distinct Risk	4
Operational Playbook for SecOps and DevOps Teams	4
Final Thoughts	5

Why This Is a Concern

With the rapid adoption of GenAI, containerized services, and edge-hosted inference models across cloud-scale infrastructure, attackers are adapting too. Malicious actors are beginning to explore abuse of model-serving protocols such as the **Model Context Protocol (MCP)** - a protocol designed for managing and contextualizing large language model (LLM) interactions. MCP traffic often leverages real-time, streaming-friendly protocols like Server-Sent Events (SSE) or WebSockets to support incremental generation and token streaming. SSE has already become the de-facto transport for real-time token streaming in Bedrock JumpStart, SageMaker Real-Time Inference, and most OSS model servers (TGI, vLLM). Its wire image over HTTPS is nearly identical whether the payload is benign or adversarial, eroding protocol-level security controls.

This creates new opportunities for threat actors to blend malicious communication into GenAI inference pipelines, making it harder to distinguish command-and-control (C2) traffic from legitimate LLM output. SSE is particularly attractive in this context due to its simplicity, persistence, and similarity to legitimate model serving patterns.

Server-Sent Events (SSE) is a lightweight, long-lived HTTP protocol originally intended for real-time data streams (e.g., stock tickers, notifications). However, its properties make it highly suitable for malware:

- Low overhead and unidirectional (server → client only) push: Ideal for silently receiving commands.
- Common usage in modern UIs: Attackers can mimic legitimate SSE patterns (e.g., dashboards, analytics streams).
- Hard to inspect: Encrypted (TLS) traffic combined with persistent streams makes DPI and signature detection difficult. Plus, ALB/NLB idle-timeout defaults (60s for CLB, 350s for ALB) are long enough for malware to maintain beacons without renegotiation, so flow-count anomaly engines miss the churn.
- Misleading similarity to GenAI inference streams: Inference workloads often emit continuous SSE updates (e.g., token-by-token streaming), which attackers can blend into.
- Limited visibility in browsers and proxies: SSE traffic is not commonly inspected or blocked by web proxies and lacks deep observability tools.

- Evasion through HTTP headers and TLS obfuscation: SSE can spoof headers to look like WebSockets or typical HTTP streams.

As organizations scale AI and microservices deployments, especially over HTTP(S) using SSE and WebSockets for efficiency, malicious use of SSE can hide in the noise. It creates a blind spot for defenders, especially in environments lacking full-layer observability or threat modeling of application-layer protocols.

How SSE-Based Malware Works

1. Why Malware Can Successfully Use SSE

Tactic	Description
SSE's Low Overhead	Lightweight protocol for persistent connection with little overhead — perfect for C2.
Mimicry of Legitimate Data	Malware mimics stock prices, logs, GenAI token streams. SSE can look totally benign.
Inspection Limitations	Browsers and proxies rarely inspect or log SSE content. <i>Note: While SSE traffic is less scrutinized by traditional security tools, modern solutions with deep packet inspection (DPI) or application-layer awareness are increasingly capable of detecting anomalies in SSE streams. However, these tools may not be universally deployed, so the risk remains significant.</i>
Encrypted Channels	HTTPS + <code>text/event-stream</code> means content is opaque.
Runs in Legitimate Clients	Python/Node clients look like real users unless tightly controlled.

2. How SSE Could Be Abused for Command and Control

While SSE is widely used for legitimate purposes such as streaming GenAI outputs, log data, or analytics events, its design makes it a viable candidate for malware to establish covert command-and-control (C2) channels. The following abuse scenarios are technically feasible and should be considered in threat models, even if not yet observed at scale.

Use Case	Description
Data Exfiltration	Steals credentials, documents, or telemetry via small SSE “data:” messages.
C2 Command Pulling	Receives commands from the C2 server via SSE without polling.
Polymorphism	SSE streams deliver obfuscated code or payloads dynamically.
Beaconing	Maintains a heartbeat with C2 by checking for new SSE messages regularly.

Detection Strategies — Challenges and Solutions

Practical SSE detection requires balancing precision, scale, and context awareness. Below is a layered detection approach with actionable guidance for both application and detection engineering teams.

Actionable Detection Strategy

To operationalize detection and reduce false positives, security and platform teams should implement explicit SSE awareness into both service metadata and deployment pipelines.

A “**service registry**” is a centralized database, API, or catalog that tracks deployed services and their declared behaviors (e.g., endpoints, protocols, team ownership). It can be implemented using tools like AWS Cloud Map, Backstage, or even custom YAML/JSON schemas in Git repositories.

Here’s an example of how SSE might be declared in a service registry:

YAML (Backstage-style):

```
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: genai-stream-api
  annotations:
    team: ml-platform
    sse-enabled: "true"
    declared-endpoints: "/events,/tokens"
```

JSON (custom registry format):

```
{
  "service": "text-stream-api",
  "team": "llm-infra",
  "sse_enabled": true,
  "declared_endpoints": ["/events", "/token"]
}
```

Infrastructure as Code (IaC) refers to managing cloud infrastructure via machine-readable configuration files (e.g., Terraform, CloudFormation, AWS CDK). These can also be used to declare SSE usage explicitly.

Technique	Practical Implementation Guidance	Priority
Endpoint SSE Registration Audits	Require application teams to explicitly register SSE usage in Infrastructure as Code (IaC) (e.g., Terraform, CloudFormation), service metadata, or API specifications (e.g., OpenAPI). Alert on any text/event-stream responses from services not declared.	High
Proxy-Layer Header Inspection	Use AWS Gateway, CloudFront, or service mesh (e.g., Istio, Envoy) to log and inspect <code>Accept: text/event-stream</code> and unusual combinations of missing headers (User-Agent, Origin). Block if User-Agent is empty or if it’s non-whitelisted for known services.	High
Persistent Connection Monitoring	Instrument ALBs or NLBs with connection telemetry. SSE misuse often keeps connections open longer than normal REST usage (60s+). Alert on unknown services holding open idle connections.	Medium
Unusual POST Pairing with GET /events	SSE-based C2 commonly follows a GET /events followed by a POST / with command output. Detect such patterns using VPC Flow Logs + HTTP metadata.	High
SSM Agent Abuse Correlation	Correlate processes making outbound SSE connections with unexpected CLI or subprocess activity (e.g., Python executing shell commands). Useful in ECS/Fargate/EKS.	Medium
Allowlist + Rate-Limit SSE at Ingress	Rate-limit <code>text/event-stream</code> connections at ALB/NLB or CloudFront layer to prevent stealth C2. Use an SSE domain allowlist by team/service.	Medium
Use CloudTrail & eBPF for Post-Connect Behavior	Even if SSE traffic is allowed, observe if the process is executing shell commands or opening file descriptors shortly after SSE establishment.	High
Deploy Canary Services	Create SSE honeypot endpoints that should never receive traffic. Alert when accessed. Works well inside VPC or in multi-tenant container clusters.	Medium

Key Indicators of Malicious SSE Behavior (With Detection Actions)

Indicator	Malicious Use Case	Differentiation from Valid SSE	Recommended Action
Content-Type: text/event-stream from unknown services	C2 channel mimicking real-time data	Valid SSE typically comes from known app subdomains or service tags	Require explicit declaration via service registry or IaC
No Origin or Referer headers	curl/Python clients don't mimic browser behavior	Browsers usually include these headers; their absence is more suspicious in user-facing traffic. Internal GenAI services may omit them	Monitor per-client-type and flag unexpected patterns
Suspicious User-Agent (e.g. python-requests)	Malware often uses basic clients	Real services use SDK-defined headers	Alert or block based on UA + path correlation
GET /events followed by POST /	Classic reverse shell behavior	Legitimate apps usually POST to structured APIs, not root	Correlate flow logs for suspicious GET/POST cycles
Small, frequent POSTs	Beaconing + exfil in short messages	Normal SSE apps send larger, structured payloads less frequently	Alert on POST frequency per client/connection
Connections open >60s with little data	SSE C2 waiting for commands	Legit SSE either streams or reconnects periodically	Alert on idle, long-lived connections from undeclared SSE services

SSE vs HTTP/3 Server Push — Why SSE Is Still a Distinct Risk

While HTTP/3 (QUIC) also supports server push, there are critical differences from SSE:

Feature	SSE	HTTP/3 Server Push
Communication	Server → Client	Server → Client
Granularity	Whole message stream	Multiple individual assets (typically in bursts)
Content Type	text/event-stream	None (inferred via headers)
Detection Ease	Easier (has headers)	Harder (deep QUIC inspection needed)
Behavioral Footprint	Linear stream	Multiplexed bursts
Usage Context	Apps, dashboards, real-time UI	Primarily preloading of static assets, though theoretically broader

Note: SSE is compatible with HTTP/3 and QUIC, as it operates at the application layer and does not depend on specific transport-layer details. The comparison here focuses on the differences between SSE and HTTP/3 server push as mechanisms for server-to-client communication.

Why use SSE for malware C2:

- Easier to control
- Fewer defenses against it
- Looks like benign app telemetry

In contrast, QUIC server push is more complex and application-integrated — harder to abuse, but also harder to monitor without specialized tooling.

Operational Playbook for SecOps and DevOps Teams

1. Inventory First-Party SSE Usage

- Require all SSE endpoints to be declared via tagging, service catalog, or IaC annotations.
- Example: Add `Metadata: SSE=true` to CloudFormation modules.

2. Enforce Gateway-Level Header Inspection

- CloudFront, AWS ALB, and API Gateway should block or log SSE requests lacking `Origin/User-Agent`.

3. Alert on Undeclared SSE Use

- Any `text/event-stream` response from a service not in the allowlist should trigger a telemetry record or alert.

4. Enable Connection Telemetry

- In NLB/ALB logs or VPC Flow Logs, flag connections that:
 - Exceed 60 seconds
 - Transmit <10KB of data
 - Repeat every few seconds

5. Threat Hunt Regularly for GET/POST Loops

- Use Athena, CloudWatch Logs Insights, or OpenSearch queries to detect:

```
filter http.uri = '/events' and response.content_type = 'text/event-stream'
group by source_ip, dest_service
where following request is POST /
```

6. Deploy Canary SSE Targets

- Host `/fake-events` endpoint behind a dummy CloudFront service and alert on access.

Final Thoughts

Detecting malicious SSE traffic requires context-aware inspection and behavioral correlation. It's not enough to block or inspect based on content alone — modern SSE abuse is about blending in, not brute force.

If you're not explicitly using SSE in your infrastructure, any persistent `text/event-stream` connections should be treated as high risk.

To stay ahead of adversaries:

- Know which teams are using SSE and why
- Monitor usage against your inventory
- Detect patterns like POST-after-SSE, idle long-lived connections, and header anomalies
- Embed detection in both your DevSecOps pipeline and production observability stack

SSE-based C2 is subtle, scalable, and dangerous — and your GenAI infrastructure is already a target.