FROM EXPERIMENTAL PILOTS TO PRODUCTION ATTACK SURFACE: WHY LLM SECURITY IS NOW A PLATFORM CONCERN

Generative AI Risks: Securing LLM Applications and Prompt Injection Defense

Executive briefing — Application security and platform engineering leaders | ~20 min

Why This Matters Now

- LLM features are shipping into production faster than security review cycles can adapt
- Traditional AppSec controls (input validation, WAF rules, static analysis) do not fully cover model-driven behavior
- LLMs blur the line between data and instructions, creating a new class of injection vulnerability
- Industry-reported range: a meaningful share of enterprise LLM pilots reach production without a formal security review — treat as directional, not verified
- This briefing frames the risk landscape and a practical path to reduce exposure before broader rollout

The LLM Risk Landscape, At a Glance

- Prompt injection — attacker-controlled input overrides intended model instructions
- Insecure output handling — model output trusted and executed or rendered without validation
- Training data and context leakage — sensitive data surfaced through model responses
- Excessive agency — models granted tool or plugin access beyond what a task requires
- Supply chain risk — third-party models, fine-tunes, and datasets of unknown provenance

Direct Prompt Injection

- A user directly submits crafted input designed to override the system prompt or safety instructions
- Goal is typically to bypass guardrails, extract system prompts, or force unintended actions
- Effective against models with weak separation between system, developer, and user instruction layers
- Illustrative scenario: a customer support bot instructed to "ignore prior instructions" and issue unauthorized refunds — not a verified case study
- Direct injection is the most tested class of attack but remains difficult to fully eliminate through prompting alone

Indirect Prompt Injection

- Malicious instructions are embedded in content the model retrieves or ingests — documents, emails, web pages, API responses
- The attacker never interacts with the application directly; the model processes the payload on the victim's behalf
- Particularly dangerous in RAG pipelines, browsing agents, and email or document assistants
- Illustrative scenario: a hidden instruction inside a shared PDF causes a document-summarization agent to exfiltrate other files in the workspace — not a verified case study
- Requires treating all retrieved and third-party content as untrusted input, not just user-typed text

Sensitive Data Exposure Through Model Outputs

- Models can surface data from training corpora, fine-tuning sets, or long-lived context windows
- Retrieval-augmented systems risk leaking documents across tenants or permission boundaries if access control is not enforced at retrieval time
- Prompt and completion logs themselves can become a repository of sensitive data if not governed
- Outputs can inadvertently reveal system prompts, internal tool names, or business logic to end users
- Data minimization and strict retrieval-scoping are more reliable controls than relying on the model to "know better"

Insecure Plugin and Tool-Use Architectures

- Tool-calling and plugin frameworks give models the ability to take real-world actions — file access, code execution, API calls, payments
- Excessive agency occurs when a model is granted broader permissions than the current task requires
- Chained tool calls can compound risk: an injected instruction in step one can drive an unauthorized action in step three
- Illustrative scenario: an agent with unrestricted file-system write access modifies configuration files based on injected content — not a verified case study
- Design principle: scope tool permissions per session and per task, not per application

Jailbreaking and Content Policy Bypass

- Adversarial prompting techniques attempt to override safety training or content restrictions
- Techniques evolve continuously — role-play framing, encoding tricks, multi-turn escalation, and instruction-hierarchy confusion
- Jailbreak resistance is a moving target; no single model or prompt defense should be treated as permanent
- Bypassed guardrails can expose the organization to reputational, compliance, and downstream liability risk
- Requires layered defenses rather than reliance on model provider safety tuning alone

Supply Chain Risk in Models and Fine-Tunes

- Third-party base models, open-weight models, and community fine-tunes carry unknown training provenance
- Fine-tuning on unvetted or poisoned datasets can embed backdoors or biased behavior that surfaces only under specific triggers
- Model and dataset provenance should be tracked with the same rigor as open-source software dependencies
- Vendor and hosting risk applies to inference APIs as well — data residency, retention, and logging terms vary widely
- Treat model selection as a procurement and security decision, not solely an ML performance decision

Guardrails and Output Filtering Architectures

- Input-side guardrails: instruction-hierarchy enforcement, content classifiers, and injection-pattern detection before the prompt reaches the model
- Output-side guardrails: response classifiers, PII redaction, and policy filters before output reaches the user or downstream system
- Defense in depth — guardrails should not depend on a single model call or a single layer to catch every case
- Guardrails add latency and false positives; tuning requires ongoing measurement, not a one-time configuration
- Guardrail logic should be externalized and version-controlled, separate from the model provider's built-in safety layer

Sandboxing Tool and Function Calls

- Execute model-initiated actions in isolated, least-privilege environments — not directly against production systems
- Apply allowlists for callable functions, parameter validation, and rate limits on tool invocations
- Require human approval or staged confirmation for high-impact actions (financial transactions, data deletion, external communication)
- Isolate code-execution sandboxes from the network and from credentials not required for the specific task
- Sandboxing converts a successful prompt injection into a contained event rather than a system compromise

Logging and Monitoring LLM Application Behavior

- Capture prompts, retrieved context, tool calls, and outputs as first-class security telemetry, not just application logs
- Monitor for anomalous patterns: unusual tool-call sequences, repeated injection attempts, or output classifier flags
- Establish baselines for normal usage so deviations are detectable rather than lost in volume
- Logs containing sensitive data require the same access controls and retention policy as other regulated data
- Feed LLM telemetry into existing SIEM and incident response workflows rather than building a parallel process

Red-Teaming LLM Applications Before Launch

- Adversarial testing should probe prompt injection, jailbreaks, data exfiltration, and tool-misuse paths specific to the application
- Include indirect injection scenarios using realistic retrieved content, not only direct chat-based attacks
- Red-team findings should map to the guardrail and sandboxing controls in place, closing gaps before production
- Repeat red-teaming on a cadence tied to model updates, prompt changes, and new tool integrations, not as a one-time gate
- Industry-reported range: organizations with formal LLM red-teaming programs report catching a substantial share of exploitable issues pre-launch — treat as directional, not a specific verified figure

A Practical Security Roadmap for Teams Shipping LLM Features

- Phase 1: Inventory all LLM use cases, data flows, and tool integrations across the organization
- Phase 2: Apply baseline controls — input/output guardrails, least-privilege tool scoping, and structured logging
- Phase 3: Red-team high-risk applications before launch and establish a recurring testing cadence
- Phase 4: Integrate LLM telemetry into existing security operations and incident response
- Phase 5: Formalize model and vendor vetting as part of standard procurement and change-management processes

Next Steps and the Ask

- Approve a cross-functional LLM security working group spanning AppSec, platform engineering, and data governance
- Fund a baseline guardrail and sandboxing pattern that teams can adopt by default, rather than building bespoke controls per project
- Mandate pre-launch red-teaming for any LLM feature with tool access, external data retrieval, or customer-facing output
- Require LLM telemetry integration into existing security monitoring within the current planning cycle
- Decision needed: sponsor and initial budget allocation to stand up the working group and baseline controls this quarter