OWASP Top 10 for LLM and Agentic Apps

Version LLM 2026 / ASI 2026 · verified 2026-11-02

Both OWASP GenAI lists, LLM01–LLM10 (2026) and ASI01–ASI10, mapped to the controls you build — capability budgets, tool policy, output encoding.

Contents

Mental model 2

The two lists and the Rule of Two 2

Prompt injection and goal hijack 3

Output handling and code execution 5

Excessive agency, tools and identity 6

Sensitive data and hidden context 7

Supply chain, poisoning and RAG 8

Memory, multi-agent and runaway loops 10

Humans in the loop, logging and evals 11

Further reading 12

Mental model

The model is an untrusted interpreter of everything in its context window. It cannot reliably tell your instructions from text inside a web page, an email or a tool result, so every input is potentially an instruction and every tool it can call is an attack surface. You cannot build a model that is never fooled; you build the system around it so that when it is fooled, nothing important breaks.

The two lists and the Rule of Two

Capability or combination	Meaning or guidance
A	Reads untrusted input
B	Touches sensitive data
C	Changes state or communicates out
Any two of A, B, C	Explicit residual-risk assessment
All three	Per-action human approval

OWASP's GenAI project publishes two lists: the **LLM Top 10 2026** (LLM01–LLM10, August 2026) for any LLM application, and the **Top 10 for Agentic Applications** (ASI01–ASI10, December 2025) for systems that plan and act. Each section below opens with its categories and the control that answers them. The LLM list adopts Meta's "Rule of Two" as the floor for budgeting agent capabilities.

```
type Caps = {
  untrusted: boolean; // A
```

```
  sensitive: boolean; // B
  acts: boolean;      // C
};

function policy(c: Caps) {
  const n = [c.untrusted, c.sensitive, c.acts]
    .filter(Boolean).length;
  if (n === 3) return "approve-each-action";
  if (n === 2) return "assess-residual-risk";
  return "ok";
}
```

> **Gotcha:** An agent that reads your inbox (A and B) and can send email (C) has all three. Prompt-injection filters do not change that classification — only removing a capability does.

Prompt injection and goal hijack

ID	Category	Control
LLM01	Prompt Injection	Bound the blast radius; filter every modality
ASI01	Agent Goal Hijack	Least privilege; confirm privileged actions

Direct injection comes from the user; indirect injection rides in on content the model reads — a web page, a PDF, a calendar invite, an image. OWASP's 2026 entry states plainly that no reliable prevention exists, so defense is architectural: filters and separate channels reduce success rates, while privilege limits cap what a successful injection achieves. Strip invisible Unicode at every ingest and render boundary.

```
// Tag blocks, variation selectors, zero-widths
const HIDDEN =
  /[\u{E0000}-\u{E007F}--]/gu;

function asData(source: string, text: string) {
  const clean = text.replace(HIDDEN, "");
  return `<untrusted source="${source}">\n` +
    `${clean}\n</untrusted>`;
}
```

> **Gotcha:** Delimiters like `<untrusted>` lower success only against attackers who do not know your scheme. OWASP cites adaptive attacks that beat such defenses most of the time — treat them as a speed bump, never as the control.

Output handling and code execution

ID	Category	Control
LLM10	Improper Output Handling	Treat output as user input; encode per sink
ASI05	Unexpected Code Execution (RCE)	No `eval`; sandbox generated code

Model output that reaches a browser, shell, SQL engine or terminal is attacker-controllable input. Encode it for the sink it lands in, use parameterized queries, and apply a strict CSP. Disable auto-rendering of Markdown images and link previews: a rendered image URL is a one-request exfiltration channel.

```
declare const db: {
  query(sql: string, args: unknown[]): unknown;
};

const IMG = /!\[[^\]]*\]\([^)]*\)/g;

function safeMarkdown(out: string) {
  return out.replace(IMG, "[image removed]");
}
// Never interpolate model text into SQL
db.query("SELECT * FROM t WHERE name = $1", [
```

```
  "model output here",
]);
```

> **Warning:** Strip ANSI escapes and other control characters before model output reaches a terminal or log file. Those sinks interpret the bytes, so a reply can rewrite what an operator sees.

Excessive agency, tools and identity

ID	Category	Control
LLM03	Excessive Agency	Fewest tools, narrowest functions
ASI02	Tool Misuse & Exploitation	Policy check in code at call time
ASI03	Identity & Privilege Abuse	Act with the user's own scoped token

Excessive Agency rose to third in 2026 because agentic deployments are where the damage lands. Give the agent only the tools the task needs, avoid open-ended tools (`run_shell`, `fetch_url`), and enforce authorization in code — complete mediation — rather than asking the model whether an action is allowed.

```
type Call = { tool: string; args: unknown };
type User = { id: string; token: string };
declare function approve(u: User, c: Call):
```

```
  Promise<boolean>;
const TOOLS: Record<string, "read" | "write"> =
  { read_mail: "read", send_mail: "write" };

async function gate(call: Call, user: User) {
  const kind = TOOLS[call.tool];
  if (!kind) throw new Error("tool not allowed");
  const ok = kind === "read" ||
    (await approve(user, call));
  if (!ok) throw new Error("rejected by user");
  return user.token; // never a shared admin key
}
```

> **Gotcha:** A mailbox-summarizer built on a generic mail API also gets send and delete. The model never needed them, but an injected instruction will happily use them.

Sensitive data and hidden context

ID	Category	Control
LLM02	Sensitive Information Disclosure	Minimize context; authorize before retrieval
LLM08	Hidden Context Exposure	No secrets or rules in prompts

Hidden Context Exposure is the 2026 successor to System Prompt Leakage, widened to all context the user is not meant to see. Assume anything in the window can be read back by the user. Keep credentials in the tool layer, and enforce access rules where the data lives — chunk-level ACLs inside the vector query, not a filter after retrieval.

```
// Bad: model can be talked into repeating both
const bad = "Refund key: sk_live_123. " +
  "Only admins may refund.";

// Good: prompt describes; code decides
const system = "You can request refunds via the " +
  "refund tool. The tool decides eligibility.";
```

> **Note:** Scrub prompts, completions and reasoning traces before they reach logs and APM tools. LLM02 treats reasoning traces as output in their own right: never log them raw to unrestricted observability.

Supply chain, poisoning and RAG

ID	Category	Control
LLM04	Supply Chain	Verified sources, signatures, AI BOM
LLM05	Data and Model Poisoning	Lineage, signed artifacts, red-team triggers

ID	Category	Control
LLM09	Vector and Embedding Weaknesses	Per-tenant ACLs, provenance at ingest
ASI04	Agentic Supply Chain Vulnerabilities	Pin and review MCP servers and tools

Models, adapters, datasets, chat templates, MCP servers and tool descriptions are all code you did not write. Pin exact versions, verify hashes or signatures, and read tool descriptions: they go straight into the model's context and can carry instructions.

```
{
  "mcpServers": {
    "tickets": {
      "command": "npx",
      "args": ["-y", "@acme/tickets-mcp@2.3.1"]
    }
  }
}
```

> **Gotcha:** `npx some-mcp-server` without a version runs whatever was published most recently, on every start. An unpinned tool server is an auto-update channel for whoever takes over the package.

Memory, multi-agent and runaway loops

ID	Category	Control
LLM06	Unbounded Consumption	Token quotas, hard spend caps
ASI06	Memory & Context Poisoning	Treat memory writes as privileged
ASI07	Insecure Inter-Agent Communication	Authenticate and validate messages
ASI08	Cascading Failures	Circuit breakers, step limits
ASI10	Rogue Agents	Kill switch, monitoring, scoped identity

Agents amplify: one task fans out into dozens of tool calls, one poisoned memory entry steers every later session, one compromised agent feeds the next. Enforce step, depth, time and cost ceilings in the runner, not in the prompt, and make limits halt execution rather than raise an alert.

```
const LIMITS = { steps: 25, usd: 2 };
const seen = new Set<string>();

function check(step: number, usd: number,
  state: string) {
  if (step > LIMITS.steps) throw Error("steps");
  if (usd > LIMITS.usd) throw new Error("budget");
  if (seen.has(state)) throw new Error("loop");
```

```
  seen.add(state); // hash of tool + args
}
```

> **Tip:** Log the prompt that caused each memory write, and require approval before instruction-like text persists across sessions. A delayed-trigger memory write is how injected instructions outlive the conversation.

Humans in the loop, logging and evals

ID	Category	Control
LLM07	Misinformation	Ground claims; verify before acting
ASI09	Human-Agent Trust Exploitation	Show the exact action, not a summary

Approval is a control only if the human sees what will execute — the full rendered command, recipient and payload — because a confident summary is exactly what a manipulated agent produces. Log every tool call with its arguments and the approving user. Test defenses against adaptive attackers who know your guardrails, and keep an injection corpus in CI.

```
type Action = { tool: string; args: object };
```

```
function approvalText(a: Action) {
  return `${a.tool}\n` +
    JSON.stringify(a.args, null, 2);
}

function audit(a: Action, user: string,
  ok: boolean) {
  console.log(JSON.stringify({
    at: new Date().toISOString(), user, ok, ...a,
  }));
}
```

> **Gotcha:** Approval fatigue is real: a reviewer who clicks “approve” fifty times a day stops reading. Reserve prompts for irreversible or external actions and make everything else impossible by design.

Further reading

- OWASP GenAI LLM Top 10 2026
- OWASP Top 10 for Agentic Applications for 2026
- LLM Top 10 2026 source and framework mappings
- OWASP GenAI Security Project