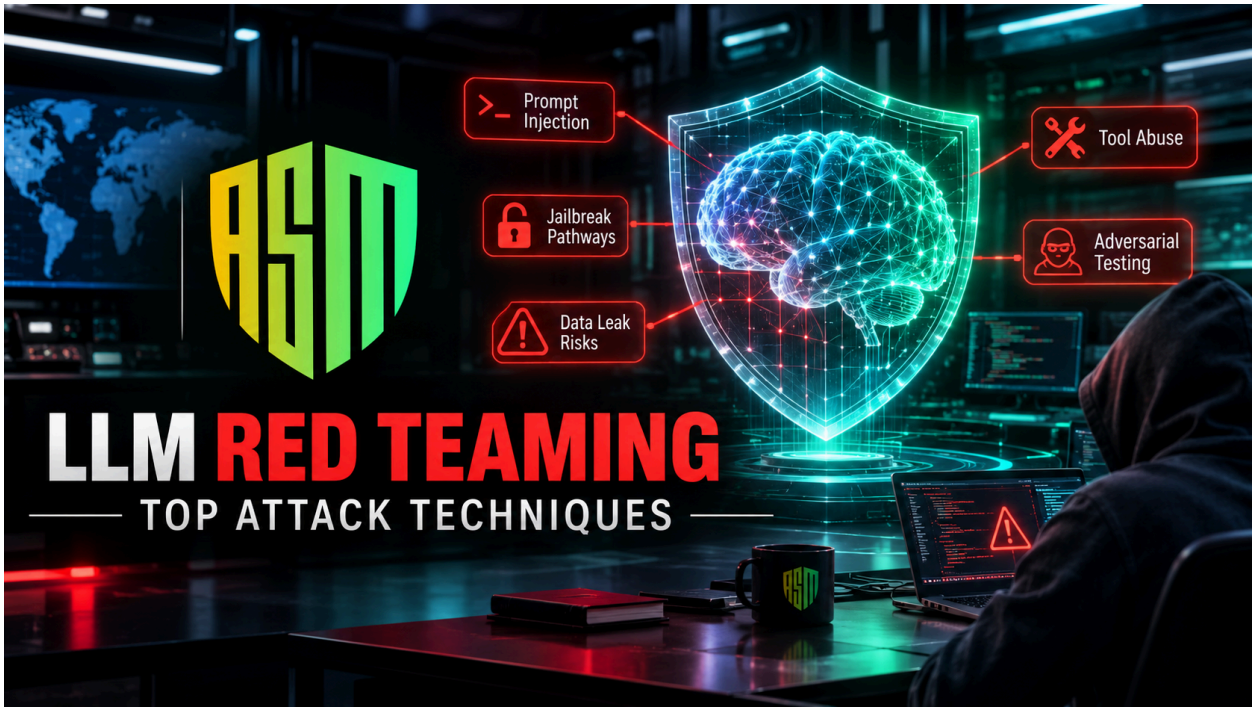
LLM Red TeamingTop Attack Techniques Explained

LLM red teaming tests a large language model application by simulating the attacks real adversaries use. The main techniques cover prompt injection, jailbreaks, system prompt leakage, sensitive data disclosure, training data poisoning, model evasion, adversarial prompts, insecure output handling and supply chain compromise. Each category has its own test ideas, warning signs and defenses and together they map closely to the OWASP Top 10 for LLM Applications.Ask five practitioners how to break a language model and you will get five different stories. Taken together, though, those stories fall into a small number of repeatable patterns. This article organizes LLM red teaming by vulnerability category rather than by tool or process, so you can see what each attack looks like, how testers probe for it and what defenders can do. The focus is on authorized testing of systems you own or are cleared to assess. For the broader discipline, see our overview of AI red teaming.



How Generative AI Red Teaming Differs From Classic Testing

Before the techniques, it helps to understand why generative AI red teaming needs its own playbook. Three properties change the game.Instructions and data share a channel. The model reads everything as text. There is no hardware enforced separation between what the developer wants and what the document says.

Behavior is probabilistic. An attack that fails nine times might succeed on the tenth. Testers therefore think in rates, not binary outcomes.Capabilities extend beyond text. Retrieval, plugins,

code execution and agents let a persuaded model affect the real world.The OWASP Top 10 for LLM Applications captures these realities in a shared vocabulary. The categories below follow that spirit, though some are grouped for clarity.

Technique 1Prompt Injection

Prompt injection is the best known weakness in model powered software. The attacker supplies text that changes the model's behavior in ways the application owner did not intend.

Direct injection comes from the user's own inputIgnore previous instructions and... variations that try to override rules.

Indirect injection is more dangerous. The malicious instruction hides in content the model fetches a web page, a shared document, a calendar invite, a support ticket, or a tool response. The user never typed it, yet the model obeys.

How testers probe it:

- Plant concealed instructions in documents, web pages and emails the system will ingest
- Test whether hidden text, such as white on white or markup comments, is treated as instruction
- Chain injection with tool access to see whether it can trigger actions
- Vary language, encoding and formatting to test filter robustness

Defensive themes treat all retrieved content as untrusted, constrain tool permissions, require confirmation for sensitive actions, separate privileges between components and monitor outputs for anomalies. For a full breakdown of mechanics and mitigations, see our article on the prompt injection attack.

Technique 2Jailbreak Attacks on LLMs

A jailbreak tries to make the model abandon its safety training or policy. While injection targets the application's instructions, jailbreaks target the model's behavioral guardrails.

Common patterns include:

- Roleplay and fiction framing asking the model to play a character who is not bound by rules
- Incremental escalation moving across many turns from harmless to harmful requests
- Request splitting asking for pieces that look benign alone
- Obfuscation using encoding, unusual spelling, or other languages to slip past filters
- Hypothetical and research framingwrapping a request in a claimed legitimate purpose

How testers probe out build a graded set of policyviolating goals relevant to the product, try multiple framings per goal and record the success rate. Relevance matters. For a customer

service bot, the interesting jailbreak is not exotic content but making it grant refunds, reveal internal policy, or make binding promises.

Defensive themes layered controls that do not depend on a single refusal, independent output classifiers, strict limits on what the model is allowed to do even when persuaded and regular retesting after model updates.

Technique 3System Prompt Leakage

System prompts often contain business rules, tone guidance and sometimes things that should never be there, like credentials, internal URLs, or data access logic. Leakage occurs when a user can coax the model into revealing them.

How testers probe it:

- Ask the model to repeat, translate, summarize, or reformat its instructions
- Request the instructions as a poem or in a code block
- Use multiturn pressure and partial recall tricks
- Compare leaked text with the known configuration to measure completeness

Why it mattersa leaked prompt lowers the cost of every later attack because the adversary now knows the rules to bypass. If it contains secrets, the leak becomes an immediate incident.

Defensive themes never place secrets in prompts, assume the prompt will eventually be exposed and enforce important rules in code outside the model.

Technique 4Sensitive Data Disclosure

Applications connected to private knowledge bases, user histories, or internal tools can reveal information to the wrong person. Disclosure can happen through several channels:

- Retrieval leakageThe search layer returns documents the current user should not see and the model dutifully summarizes them.
- Crosssession leakage memory or caching features expose one user's content to another.
- Training data memorization the model reproduces fragments of data it was trained on.
- Overly helpful summaries the assistant includes personal or confidential details when asked broad questions.

How testers probe it log in as a lowprivilege user and request information belonging to a higher privilege user or a different tenant. Ask broad questions such as summarize everything you know about the finance team. Test whether access controls are enforced at the retrieval layer rather than only in the prompt.Defensive themes enforce authorization before data reaches the model, minimize what is stored, scrub sensitive fields and log access to confidential sources.

The access control principles in API security and the OWASP API risks apply directly, because broken objectlevel authorization is just as damaging when a model is the one fetching the object.

Technique Training Data Poisoning

Poisoning targets the learning process rather than the live conversation. An attacker who can influence training data, fine tuning examples, or even the documents a system retrieves may be able to plant backdoors, bias behavior, or degrade quality.

Examples of what poisoning can look like:

- A trigger phrase that makes the model produce a specific harmful output
- Subtle bias introduced into a fine tuning set
- Malicious content seeded in public sources that later get scraped
- Tainted entries in a retrieval index that steer answers

How testers probe out, review data provenance and pipelines, test for unusual behavior on rare trigger inputs, compare model behavior against a trusted baseline and examine who can write to training or retrieval stores.

Defensive themes vet and track data sources, restrict write access, validate fine tuning data and monitor for behavioral drift.

Technique 6Model Evasion Attacks

Evasion attacks craft inputs that cause a model to misjudge what it sees. In a classic example, tiny changes to an image cause a classifier to label it wrongly. In the language setting, evasion often targets the safety filters and detectors layered around the main model: a harmful message reworded or encoded so a moderation classifier misses it, or malware like content disguised to slip past an automated reviewer.

How testers probe intake inputs a detector correctly blocks and applies systematic transformations such as paraphrase, character substitution, spacing, language switching and encoding. Measure how quickly detection degrades.Defensive schemes do not rely on one detector, combine pattern checks with semantic analysis, retrain on discovered bypasses and apply rate limits to slow automated probing.Applications that let a model produce or influence structured documents should also be checked against XXE injection risks.

Technique 7 Adversarial Prompts

Adversarial prompts are inputs engineered, sometimes automatically, to push a model toward unintended behavior. Researchers have shown that optimization methods can generate oddlooking strings that reliably trigger unsafe responses across multiple models. Practical

testers also use LLMs to generate large volumes of attack variations, then score which ones succeed.

Useful categories to cover:

- Instruction conflict prompts that pit system rules against user requests
- Ambiguity prompts that exploit unclear policy boundaries
- Longcontext prompts that bury instructions in large inputs
- Multilingual prompts that exploit uneven safety coverage across languages

How testers probe to automate generation and scoring, keep a library of prompts that worked and rerun it after every significant change.Defensive adversarial training, input normalization, output verification and continuous regression testing.

Technique Insecure AI Output Handling

The model's output is untrusted data. When applications pass it to another component without validation, classic vulnerabilities return. If a generated string reaches a browser, you may get crosssite scripting. If it reaches a database layer, you may get an injection. If it reaches a template engine, shell, or parser, the consequences can be worse.



How testers probe aircraft prompts that cause the model to emit payloadlike strings, then trace where that output goes. Does the interface render it as HTML? Is it concatenated into a query? Is it executed as code? Practicing in a SQL injection lab builds the reflex to ask what happens when text reaches a query and studying server side template injection shows how rendered

content can become code execution. Defensive themes treat model output like user input, encode for the destination context, use parameterized queries, sandbox code execution and validate structured output against strict schemas.

Technique 9LLM Supply Chain Attacks

Modern AI systems assemble many third party parts per trained models, adapters, datasets, libraries, plugins and hosted services. Each is a potential entry point.

Typical risks:

- Tampered model files that execute code when loaded
- Malicious or typosquatted packages in the machine learning toolchain
- Compromised datasets that introduce poisoning
- Vulnerable plugins with excessive permissions
- Unvetted model providers with unclear security practices

How testers probe it, inventory every component, check hashes and signatures, review how models and packages are pulled and examine plugin permissions. A careful source code review lab approach helps when auditing loading code, dependency handling and integration glue.

Defensive themes maintain a bill of materials for models and data, pin versions, verify integrity, prefer safe serialization formats and apply least privilege to plugins.

Putting the Techniques Together

Real attacks rarely stay inside one category. A realistic chain might look like this:

1. An attacker plants a hidden instruction on a web page (indirect injection).
2. A research assistant visits the page and follows the instructions.
3. The instruction tells it to reveal its system prompt and then to search internal files (leakage and disclosure).
4. The assistant sends the results to an attacker controlled address through a tool (excessive agency).

Each step on its own might look minor. Together they form a serious incident. Good testers therefore combine techniques and score the chain, not just the individual links. Mapping chains is easier after a short threat modeling session that lists entry points, privileges and trust boundaries. Pairing this approach with a broader web application hacking lab practice reinforces the habit of thinking in attack paths rather than single bugs.

A Quick Coverage Table

Technique	Primary target	Typical impact	First test to run

Prompt injection	Application instructions	Unauthorized actions, data exposure	Hidden instruction in an uploaded file
Jailbreaks	Model safety behavior	Policy violations	Roleplay framing against a key policy
System prompt leakage	Configuration	Easier follow on attacks	Ask for instructions in a different format
Data disclosure	Access boundaries	Privacy breach	Lowprivilege user requests restricted data
Data poisoning	Training or retrieval data	Backdoors, bias	Review data write access and provenance
Model evasion	Detectors and filters	Bypassed moderation	Paraphrase and encode a blocked input
Adversarial prompts	Model robustness	Unsafe output at scale	Automated variation generation
Output handling	Downstream components	Injection, code execution	Trace model output to its sink
Supply chain	Models, packages, plugins	Compromise of the pipeline	Inventory and verify components

Tips for Running TechniqueBased Tests

- Start with the categories that match your system's real capabilities. A readonly chatbot needs less agency testing than an autonomous agent.
- Record exact inputs, model versions and settings so developers can reproduce the issue.
- Measure rates by repeating each test, because randomness can hide weaknesses.
- Always test within authorization and avoid generating or storing harmful content beyond what the assessment requires.
- Retest after every fix and rerun the full suite when the model or prompt changes.

Conclusion

LLM red teaming becomes manageable once you break it into categories. Prompt injection and jailbreaks attack behavior, leakage and disclosure attack confidentiality, poisoning and supply chain issues attack trust in the pipeline, evasion and adversarial prompts attack robustness and insecure output handling reconnects AI to the oldest vulnerabilities in software. Knowing each pattern gives you a concrete set of tests and a clear sense of what to fix.The strongest

programs treat these techniques as a living checklist. They run them on every significant change, combine them into realistic attack chains and keep the underlying application security fundamentals sharp. Start with the categories that match your system's capabilities, measure what you find and keep improving. Models will change, but the habit of testing like an attacker will keep your defenses honest.To see how these topics connect with the wider application security path, browse the learning resources on the AppSecMaster homepage.

Frequently Asked Questions (FAQs)

What is LLM red teaming?

LLM red teaming is authorized adversarial testing of applications built on large language models. Testers simulate attacks such as prompt injection, jailbreaks and data extraction to expose weaknesses in the model, its prompts, its data connections and its tools, so the issues can be fixed before release.

What are the most common attacks on LLM applications?

Prompt injection, jailbreaks, sensitive data disclosure, system prompt leakage, insecure output handling and excessive tool permissions appear most often. Data poisoning and supply chain compromise are less visible but can have severe, long lasting effects when they occur.

What is the difference between prompt injection and a jailbreak?

Prompt injection manipulates the application by smuggling instructions into the model's input, often through external content. A jailbreak targets the model's own safety behavior to make it produce restricted output. They often combine, but the defenses differ.

How does the OWASP Top 10 for LLM Applications help red teams?

It provides a shared list of risk categories, which helps teams plan coverage, communicate findings in common terms and confirm that no major class of weakness is skipped. It works best alongside a system specific threat model.

Can LLM red teaming be automated?

Partly. Automation is excellent for generating prompt variations, measuring success rates and catching regressions. Human testers remain essential for creative, multistep and contextspecific attacks and for judging real business impact.