AI Application Security Lab:
Data Leakage, Information Extraction, and Guardrails

Student Lab

Overview

In this lab, you will attack an instructor-provided, already-implemented retrieval-augmented study and quiz assistant. Your job is not to build a new app. Your job is to run the shipped application inside a Google Colab notebook, evaluate its behavior across several CLI paths, and determine whether it can be pushed into leaking information it was supposed to protect.

The lab combines three related attack styles into one evaluation:

- direct extraction of sensitive material from prompt context or retrieved context;
- bypassing weak guardrails that try to block disclosure;
- abusing attacker-controlled or seeded retrieval content to induce leakage or unsafe behavior.

The emphasis is on observing failure modes, documenting attack paths, and explaining why the system leaked information or resisted leakage attempts.

Learning Goals

By the end of the lab, you should be able to:

- identify how sensitive information can leak from prompt context, retrieved context, and model outputs in a RAG-style application;
- evaluate the difference between a nominal guardrail and an actually effective guardrail;
- demonstrate how attacker-controlled retrieval content can change model behavior or induce disclosure;
- document concrete attack transcripts and explain the trust failures they reveal;
- propose practical defenses that fit the architecture of the shipped application.

Setup

The starter package includes everything you need: a Colab notebook (`lab.ipynb`), a small Python application, a mixed-trust document corpus, prompt templates, and deliverable templates.

Default Backend: Llama 3.1 8B In Colab

The lab runs in Google Colab against an in-runtime Ollama server. There is no local Python install, no institutional API key, and no model download to manage outside Colab.

1. Unzip the starter and upload the whole `topic-07-ai-app-security-lab` folder to the top of your Google Drive (so it lives at `My Drive/topic-07-ai-app-security-lab`).
2. Open `lab.ipynb` from that Drive folder in Colab (https://colab.research.google.com/, then File → Open notebook → Google Drive).
3. From the menu, choose **Runtime → Change runtime type → T4 GPU**.
4. Run the cells top to bottom. Section 2 mounts your Drive and copies the starter into `/content/lab`; the first run also installs Ollama, starts it as a background process, and pulls `llama3.1:8b` (Meta Llama 3.1 8B Instruct, ≈4.7 GB, ≈3–5 minutes).

Ollama exposes an OpenAI-compatible API at `http://localhost:11434/v1`, and the lab's `llm_client.py` talks to it without modification. Full setup details and troubleshooting are in `COLAB_SETUP.md` inside the starter.

Persistence

Colab runtimes are ephemeral. The final notebook cell zips `attack_log.md`, `analysis.md`, and any optional patches into a download bundle. Your Google Drive is already mounted from Section 2, so you can also copy your deliverables back into a folder you control.

Smoke Test

The notebook runs these four commands automatically once Ollama is up:

```
python3 app.py --list-sources
python3 app.py --question "How can retrieved context leak sensitive information?"
python3 app.py --generate-quiz
python3 app.py --grade-question "Why are prompt-only guardrails weak?" \
  --student-answer "Because the model can still see internal retrieved context."
```

Application Under Test

The shipped application is a small RAG study and quiz assistant. Application paths you should test:

- `python3 app.py -list-sources`: inspect the mixed-trust corpus and its public/internal split;
- `python3 app.py -question "..."`: study-assistant path;
- `python3 app.py -generate-quiz`: quiz-generation path;
- `python3 app.py -grade-question "..." -student-answer "..."`: grading-feedback path.

The starter ships with a working application, a small document corpus, internally protected content (including obviously fake `LAB7-FAKE-*` canaries), and starter templates for the written deliverables.

The shipped system has at least one intentionally weak control: the student-facing retrieval filter in `rag_store.py` only penalizes internal documents instead of excluding them. If you choose the optional patch path, keep the patch small and explain why that patch point matters.

Tasks

1. Inspect The System And Attack Surface

Before attacking, review the provided materials and understand:

- what documents are available to the application;
- what information appears intended for normal student-facing use;
- what information appears internal, hidden, or protected;
- what guardrails or non-disclosure rules the application claims to enforce;
- where retrieval enters the quiz-generation or grading flow;
- which CLI paths behave differently.

Write a short system summary in `attack_log.md` so your later attacks are tied to the real architecture.

2. Perform A Direct Leakage Evaluation

Craft prompts that attempt to extract sensitive material from the system's prompt context or retrieved context without first modifying the corpus.

Examples of valid goals include:

- asking the model to reveal hidden instructions;
- asking for the exact context used to answer;
- asking for internal notes or excluded passages;
- inducing the model to quote or summarize protected material.

You must document at least one direct extraction attempt. It does not have to succeed fully, but you must explain what happened and why.

3. Perform A Guardrail-Bypass Evaluation

Test whether the application's explicit guardrails can be bypassed. These may include instructions such as "do not reveal internal instructions," "do not expose source passages," "do not provide the answer key," or "refuse requests for protected context."

Try multiple attack styles, such as:

- reframing the request as debugging or auditing;
- asking for transformation, summarization, or translation instead of direct disclosure;
- role-play or "act as" prompts;
- multi-turn probing that gradually escalates toward the protected content.

Document at least one guardrail-bypass attempt and explain whether the guardrail held, partly failed, or failed completely.

4. Perform A Retrieval-Abuse Evaluation

Evaluate whether attacker-controlled or seeded retrieval content can influence the model to leak information or behave unsafely. In the shipped starter, this should begin with the seeded poisoned chunk in `supplemental_notes_from_ta.txt`, which contains adversarial language about "audit," "verify," "blocked," "internal," or "quote."

Your goal is to determine whether untrusted retrieved content can:

- override or confuse the intended policy;
- cause the model to expose protected content;
- distort grading feedback or quiz-generation behavior in a way that reveals internal information.

Document at least one retrieval-based attack attempt.

5. Compare The Three Attack Paths

After gathering evidence, compare the three categories:

- Which attack path was most effective?
- Which one required the least attacker effort?
- Which one exposed the most sensitive information?
- Which failures were caused by prompt design versus retrieval trust versus output handling?

Record this comparison in `analysis.md`.

6. Propose Defenses

Propose at least two realistic defenses. Examples:

- separating protected documents from student-query retrieval;
- attaching document-level access controls or metadata filters;
- improving prompt isolation between instructions and untrusted context;
- removing or reducing raw passage disclosure;
- changing output handling to avoid verbatim quoting of protected material.

Document the defense ideas in `analysis.md`. You do not need to fully implement a hardened system in this lab.

7. Optional Small Patch And Re-Test

If you choose to go beyond the required analysis work, you may implement one small patch and re-test a relevant attack. A good patch is tightly scoped, clearly connected to one observed failure mode, and easy to explain in terms of trust boundaries and tradeoffs. One intended patch point is the student-facing retrieval filter in `rag_store.py`; you may choose another small patch if you can justify it.

Deliverables

Required submission:

- `attack_log.md` with at least three documented attack attempts (one direct leakage, one guardrail bypass, one retrieval abuse);
- `analysis.md` with your comparison of the three attack paths, root-cause analysis, at least two defense ideas, and any optional patch and re-test notes;
- `agent_prompts.md` only if AI assistance materially shaped your work.

Optional:

- `bonus_agents_bypass.md` if you pursue the `AGENTS.md` bonus challenge;
- a small code patch if you choose the optional patch path.

Expected Use Of AI Assistance

AI assistance is allowed, but only as bounded help. You may use an AI assistant to brainstorm candidate attack prompts, help categorize observed failures, critique whether your explanation of a trust boundary is clear, or compare possible mitigations after you have gathered your own evidence.

You may not use AI assistance as a substitute for running the evaluation yourself or for writing a fabricated attack log. Your submission must be grounded in the actual behavior of the provided system.

If AI assistance materially shaped your work, complete `agent_prompts.md` with the prompts, transcript excerpts, or concise summaries that influenced your submission.

The starter includes an `AGENTS.md` file. Its purpose is to steer coding assistants toward bounded help rather than one-shot completion. You can technically modify that file, but please do not unless the lab explicitly asks you to evaluate or patch it.

Safety And Scope Notes

- Keep your work scoped to the instructor-provided lab environment.
- Do not attack external services, accounts, or systems outside the lab materials.
- Do not fabricate attack evidence. Partial failures are acceptable if they are analyzed honestly.
- Focus on information extraction and guardrail failure, not on destructive system compromise.
- If the system reveals seeded canary material, treat it as evidence, not as something to share outside the lab workflow.

Optional Bonus: `AGENTS.md` Bypass

The starter ships with an `AGENTS.md` file. For extra credit, you may try to craft a prompt that causes a coding assistant to work around that guidance in a meaningful and reproducible way.

Bonus credit is available as follows:

- +10 points if you provide a reproducible prompt or prompt sequence that reliably gets around the `AGENTS.md` guidance;
- +10 points if you also propose a plausible patch or improvement to `AGENTS.md` or the surrounding workflow that would make your bypass harder to reproduce.

To receive bonus credit, complete `bonus_agents_bypass.md` with the exact prompt or prompt sequence used, the observed agent behavior or output that shows the bypass worked, a short explanation of what part of the `AGENTS.md` guidance was bypassed, and enough detail for staff to reproduce the result safely.

Rubric

Category	Points	What strong work demonstrates
Direct leakage evaluation	20	A clear, well-documented direct extraction attempt tied to the real system, with strong reasoning about why it succeeded or failed.
Guardrail-bypass evaluation	20	A concrete attempt to evade explicit guardrails with careful, accurate explanation of the observed behavior.
Retrieval-abuse evaluation	20	A convincing evaluation of attacker-controlled or seeded retrieval content as a source of leakage or unsafe behavior.
Root-cause analysis and defense reasoning	25	Analysis distinguishes prompt, retrieval, and output-handling failures; proposes realistic defenses; includes a thoughtful re-test or defense evaluation.
Documentation, honesty of evidence, and agent-use disclosure	15	Submission is organized, evidence-based, complete, and includes `agent_prompts.md` when AI assistance materially shaped the work.

Total: 100 points.

AI Assistance Acknowledgment

This lab was developed and tested with substantial assistance from AI coding and writing tools. AI assistance contributed to drafting the lab write-up, generating the starter scaffolding, building the synthetic mixed-trust corpus and seeded canary content, producing the reference solution and instructor materials, and exercising the application's CLI paths during testing.

The instructor reviewed and revised the materials, made the final design decisions, and is responsible for the lab's content, security framing, and pedagogical choices. AI was used as a bounded development aid, not as a replacement for that judgment.

If you notice unclear instructions, install issues, factual errors, or rough edges, please report them so the lab can be improved.

Source Acknowledgment

The shipped application and overall lab structure are adapted from the GW course *AI Application Security* (Lab 4), with a local-LLM-first backend.