Prompt Injection Defense

Pattern reference and integration guide for LLM applications

> **This is the working reference behind our open-source middleware** (`prompt-injection-defense`, MIT). Everything documented here is implemented in the library and reproduced from its source, so you can verify each claim against the code rather than take our word for it.
>
> It covers the attack families the detector matches, the scoring model, the four public functions, and four integration patterns. Use it as a checklist even if you write your own filter.

1. What prompt injection actually is

An LLM does not distinguish between the instructions you wrote and text a user pasted. Both arrive as tokens in the same context. Prompt injection is any user-supplied text crafted to be read as instruction rather than as data. It is catalogued as **LLM01** in the OWASP Top 10 for LLM Applications.

The consequence depends on what your model can reach. A chatbot with no tools leaks its system prompt. A model wired to a database, a mailbox, or a payment API performs actions on behalf of whoever wrote the injected text.

2. Attack families and detection weights

The detector matches against six families. Each pattern carries a weight; the risk score is the mean weight of the patterns that matched, capped at 1.0. Input is flagged as an attack above 0.5.

Family	Example trigger	Weight
Instruction override	ignore all previous instructions	0.90
	disregard prior context	0.90
	forget everything	0.85
System prompt manipulation	you are now a …	0.80
	`[system]:`	0.95
Special token injection	`<|im_start|>`, `<|im_end|>`	1.00
Role hijacking	pretend to be a … / act as a …	0.70
	you have admin access	0.90
Data exfiltration	repeat your system prompt	0.95
	what were your original instructions	0.90
Whitespace exploit	five or more consecutive newlines	0.60
Markdown injection	`### instruction`	0.80

Confidence is derived from the same score: above 0.8 is `high`, above 0.5 is `medium`, anything else `low`.

3. Output-side credential leak patterns

Detection on the way in is only half the job. The library also scans model output for four credential shapes before it reaches your user or your logs:

```
/sk-[a-zA-Z0-9]{32,}/                       OpenAI-style secret keys
/api[_-]?key[\s:="']+[a-zA-Z0-9_-]{20,}/i   generic API keys
/password[\s:="']+[^\s]{6,}/i               passwords in plain text
/Bearer\s+[a-zA-Z0-9_-]{20,}/               bearer tokens
```

4. The four functions

detectPromptInjection(input)

```
{
  isAttack: boolean;      // riskScore > 0.5
  riskScore: number;      // 0.0 - 1.0
  patterns: string[];     // which families matched
  confidence: 'low' | 'medium' | 'high';
}
```

sanitizeInput(input, options?)

Strips matched patterns and truncates. Returns the cleaned string plus the list of what was removed, so you can log the attempt without logging the payload verbatim.

validateOutput(output)

Runs the credential patterns above against model output. Returns `isSafe`, the leaks found, and warnings.

promptInjectionDefense(options?)

Express middleware wrapping the three. Options: `maxLength` (default 2000), `aggressiveMode`, `blockOnDetection` (default true), `logAttempts` (default true).

5. Integration patterns

Express

```
import { promptInjectionDefense } from 'prompt-injection-defense';

app.post('/api/chat', promptInjectionDefense(), async (req, res) => {
  // req.body.message is sanitized before it reaches the model
});
```

Next.js route handler

```
const detection = detectPromptInjection(userMessage);
if (detection.riskScore > 0.7) {
  return Response.json({ error: 'Invalid input' }, { status: 400 });
}
const { cleaned } = sanitizeInput(userMessage);
```

n8n Function node

```
const detection = detectPromptInjection($json.message);
if (detection.isAttack && detection.riskScore > 0.8) {
  throw new Error('Potential prompt injection detected');
}
return { sanitizedMessage: sanitizeInput($json.message).cleaned };
```

Supabase Edge Function

```
import { detectPromptInjection } from 'npm:prompt-injection-defense';
// reject before the request ever reaches your model provider
```

6. Where to put the threshold

The threshold is a business decision, not a technical one, because it trades false positives against blast radius:

- **0.5** (library default for `isAttack`) suits a model with tool access, where a successful injection does real damage.
- **0.7-0.8** suits a support chatbot with no tools, where over-blocking a legitimate customer costs more than a leaked system prompt.
- **Log everything, block above threshold.** The log is what tells you where to move the line next month.

What this does not do, stated plainly. This is pattern matching over English-language attack shapes. It raises the cost of the obvious attacks; it does not make an application injection-proof. It will not catch obfuscated payloads, non-English phrasing, base64 or unicode-smuggled instructions, or indirect injection where the hostile text arrives through a document, a web page, or a retrieved chunk rather than through the user field.

We publish no benchmark accuracy figure for it, because the repository ships no test suite that would substantiate one. If you need a number for your own risk register, measure it against your own traffic.

It is one layer. The layers that matter more: least-privilege tool scopes, human confirmation before irreversible actions, output encoding, and treating every retrieved document as untrusted.

7. A ten-minute self-check

1. List every tool, function, or API your model can call. For each: what is the worst thing it can do if the caller is hostile?
2. Which of those need human confirmation and do not have it?
3. Does any retrieved content (document, web page, email body) enter the context unlabelled as untrusted?
4. Are model outputs scanned for credentials before they hit your logs?
5. If an injection succeeded right now, what in your logs would tell you?

An unanswered question in that list is worth more attention than tuning a regex threshold.

DeviDevs Technologies S.R.L. · AI agents, automation and AI security
Source: github.com/DeviDevs-Technologies/prompt-injection-defense (MIT)
devidevs.com · security@devidevs.com