

AIPatch: LLM Assisted Patch Copilot for Critical Open Source Infrastructure

Joel Christoph

Defensive Acceleration Hackathon Submission

Abstract

Open source software underpins modern AI systems and critical infrastructure. A single widely used library can sit underneath thousands of applications. When a vulnerability is discovered in such a project, the time between disclosure and patching across the dependency graph creates a large attack window.

Many of these libraries are maintained by one or two volunteers who have limited time and limited access to security expertise. At the same time, large language models have become strong at code understanding and synthesis. AIPatch is a prototype defensive tool that uses these capabilities to help maintainers understand, prioritise, and patch known vulnerabilities faster.

Given a repository and a set of known or suspected issues, AIPatch combines lightweight static analysis with LLM based reasoning to summarise each vulnerability in plain language, propose candidate code patches as diffs, and generate a structured report that maintainers can turn into pull requests or advisories. The core design principle is to strengthen the shield without adding new offensive capability.

This report describes the motivation, threat model, system design, hackathon prototype, and roadmap for AIPatch as a reusable piece of defensive infrastructure for the open source ecosystem that supports AI.

1 Motivation

Modern software stacks are built on deep chains of open source dependencies. In AI pipelines these chains include data processing libraries, numerical kernels, model serving frameworks, and glue code around accelerators. When a vulnerability exists anywhere inside that stack, attackers can target it in order to exfiltrate models, escalate privileges, or move laterally inside an organisation.

Large companies increasingly have security teams and internal tools that scan, triage, and patch vulnerabilities as part of a continuous process. In contrast, the maintainers of widely used open source projects are often volunteers. They may receive vulnerability reports and scanner output, but still struggle to decide which issues matter most, how to implement safe fixes, and how to communicate the changes to downstream users. From a defensive acceleration perspective this is a serious bottleneck. Offensive actors can already automate scanning and exploit development, while defensive capacity in the long tail of open source projects has not kept up.

The central idea behind AIPatch is simple. If we can give maintainers an LLM assisted copilot for security patches that fits into their existing workflow, then we can reduce the time and expertise required to move from “a vulnerability is known” to “a good patch has been merged and clearly documented.”

2 Threat Model and Problem Statement

AIPatch focuses on a realistic and common setting. The project is an open source library or application with at least some security sensitive downstream users. One or more vulnerabilities are already known to the maintainer, either through a public advisory, an issue report, or automated scanning. Attackers can read the same advisories, scan the code, and use automation to generate and weaponise exploits, while maintainers have limited time and limited security expertise.

The tool deliberately avoids the setting in which it would be used to discover new zero day vulnerabilities in arbitrary closed code bases. Instead it tries to reduce the time and effort required to move from a known issue to a good quality patch in projects that already want to fix security problems.

More concretely, the problem is the following. We are given a code repository and a list of known or suspected issues from scanners or advisories, maintained by people who want to fix them but are resource constrained. We would like a system that can explain each vulnerability in clear natural language, propose candidate patches as unified diffs grounded in local code context, surface information that helps prioritise issues by likely impact and exploitability, and produce a short report that can be pasted into pull requests, changelogs, or advisories. The system should clearly improve defensive capacity without creating new offensive capability.

3 System Design

3.1 Architecture

The AIPatch prototype has four main components.

First, a *repository ingestion* module accepts a local path or a Git URL. AIPatch clones or reads the repository, parses the directory structure, and collects candidate files for analysis. In the hackathon prototype we focus on Python projects, which keeps parsing simple and makes demonstration easier.

Second, an *issue collection and normalisation* layer gathers information about vulnerabilities. Issues can come from multiple sources. For the prototype we support a simple built in static checker for insecure patterns in Python and a minimal text file of known issues keyed by path and line numbers. These inputs are normalised into a small schema with a file path and line span, a code snippet that shows the local context, and a natural language description or scanner message where available.

Third, an *LLM assisted patch reasoning* component takes each normalised issue and prepares the prompt for the model. The prompt combines the issue description and metadata, the code snippet and surrounding lines, and any notes on the project such as the target Python version. The model is asked to explain the issue in maintainer friendly language, propose a candidate patch as a unified diff that fixes the issue, and flag obvious backward compatibility risks. Prompts also specify that exploit code or proof of concept attacks should not be generated.

Finally, a *reporting and export* component assembles the outputs into a form that maintainers can use. The current prototype writes a Markdown report with one section per issue and a concise explanation and patch. This format is designed to be easy to copy into a pull request or advisory. The same information can also be written to a machine readable format if integration with other tools becomes useful.

3.2 User workflow

The intended maintainer workflow is deliberately simple. A maintainer runs AIPatch on a repository with a short command line call. Once the analysis step is complete, they open the generated report, read the vulnerability summaries and candidate patches, and decide which ones to adopt. They then apply patches in a branch, run their existing test suite, and refine or reject suggestions as they see fit. If they decide to merge a patch, they can reuse the explanatory text for the pull request description, changelog, or advisory.

AIPatch is meant to be an assistant rather than an autonomous patch deployment system. Maintainership remains the locus of responsibility and control.

4 Hackathon Prototype

During the Defensive Acceleration Hackathon we implemented a minimal but end to end prototype that demonstrates this workflow.

The prototype accepts a Git repository URL or a local path and scans Python files for a small set of unsafe patterns. It then constructs prompts that combine scanner messages with code context and calls a large language model to generate explanations and proposed diffs. Finally it writes out a Markdown report and a patch file.

For the purposes of demonstration we tested AIPatch on small open source style Python projects where it is easy to introduce and then fix insecure patterns. These projects are fast to clone, the diffs stay short enough to inspect at a glance, and unit tests can be run quickly by maintainers after applying patches.

In a typical demo scenario we show a repository with a deliberately introduced security issue, run AIPatch, inspect the generated summary and diff, apply the patch, rerun tests, and present the resulting report as the basis for a pull request. This provides a concrete and intuitive picture of how AIPatch can help in real maintainer workflows.

5 Impact Assessment

AIPatch targets a specific defensive leverage point, namely the time and attention of under resourced maintainers whose projects sit inside security critical stacks.

If the tool works well at scale, it should reduce the time from a known vulnerability to a merged fix in widely used packages. It should make communication between security researchers and maintainers easier by providing clear summaries and structured patch descriptions. It should also reduce the cognitive overhead of reading raw scanner output, which can increase the willingness of maintainers to engage with responsible disclosures rather than ignoring them.

In the near term, success can be measured by pilots with real projects where maintainers report time savings and reduced friction in dealing with vulnerabilities. We can also track how often AIPatch generated text or diffs are used in pull requests and advisories, and gather qualitative feedback on where in the patching workflow LLM assistance is most valuable.

In the longer term, one can imagine AIPatch integrating with vulnerability databases and package registries. Such integration would make it easier to track which advisories already have high quality reviewed patches and which still need attention, and could provide ecosystem level metrics about the health of the defensive surface.

6 Safety Considerations and Limitations

Security tools must be designed in a way that does not create new risks. For AIPatch this has two parts.

First, the intended user is the maintainer or an authorised contributor to the project. The tool operates on vulnerabilities that are already known to the maintainer or public, rather than scanning arbitrary closed code bases in search of new weaknesses. Prompts to the model explicitly request defensive patches rather than exploit proof of concept code, and documentation emphasises responsible disclosure norms and the importance of avoiding unnecessary information hazards.

Second, we acknowledge the limitations of the current prototype. The present version focuses only on Python projects and uses simple pattern based static analysis for issue discovery. It does not provide automated verification beyond the maintainer’s own test suite, and it does not attempt to analyse system wide security guarantees. These limitations are acceptable for a hackathon prototype but they define a clear agenda for further work.

7 Roadmap

After the hackathon, there are several natural steps for AIPatch.

One priority is to extend support to C and C plus plus, where memory safety problems are especially important and where a good patch copilot could be particularly valuable. A second step is to integrate AIPatch into continuous integration systems so that maintainers can run it automatically on new issues or on security related branches. A third step is to partner with a small set of maintainers of critical packages to evaluate the tool in real workflows and iterate on the interface and prompts. Finally, it would be useful to explore links to existing initiatives in software supply chain security and to open source security foundations so that AIPatch can complement, not duplicate, existing efforts.

Acknowledgements

This prototype was developed for the Defensive Acceleration Hackathon organised by Apart Research with support from Halcyon Futures. The broader AIPatch direction is informed by work supported by the OpenAI Cybersecurity Grant Program.