Codebase Interrogator: Automate Your Search for Vulnerabilities at Scale Leveraging a New Tool

Bill Horn
March 7, 2024

Abstract

While recent advancements in large language models (LLMs) have paved the way for options that are more adept at analyzing source code and providing tailored remediation advice, the current options available to security professionals and software engineers involve time consuming techniques and risk missing areas of code. Additionally, an organization's legacy applications may not be slated for manual review any time soon. This paper presents a new tool that automates the process, performs holistic inspection, and greatly increases the speed of producing responses for red teams, blue teams, and security engineers. The paper also outlines setup, configuration, limitations, and how to get the most out of the tool, including how to customize it to meet specific needs. Other sections of the paper cover things to watch out for such as potential cost, performance, and privacy concerns.

Introduction

I've created a tool for inspecting entire source code repositories for potential security vulnerabilities that leverages the code analysis capabilities of specific large language models. While the tool has multiple potential use cases, I initially created it with the primary goal to automate security analysis across many files.

In the process of researching capabilities and building the tool, I also achieved the following additional goals:

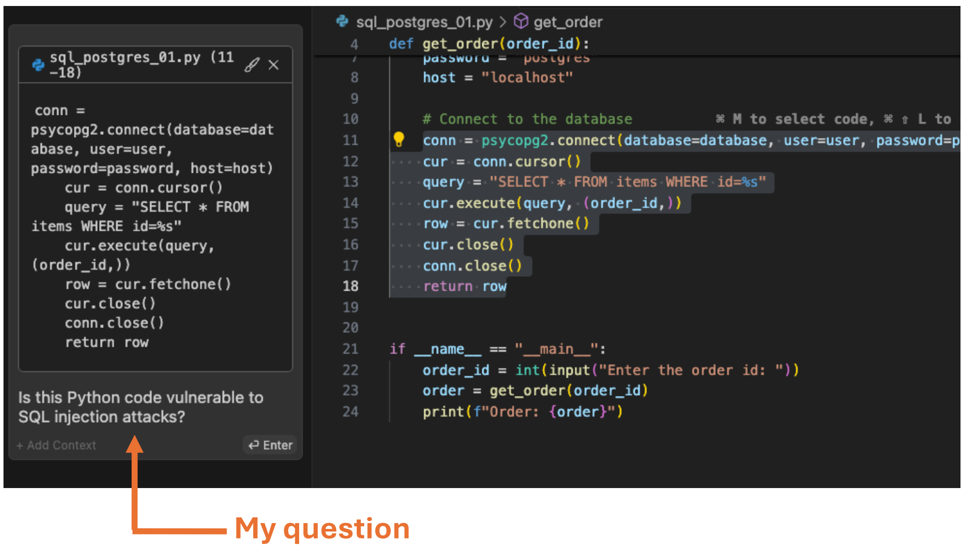
- The tool had to be effective at producing credible findings.
- Must be cost-effective to run.
- Can be implemented with relative ease by other colleagues.
- Flexible enough to allow for customization.
- Presents the results in an easy to read and consume format.
- Offers concrete remediation advice and examples.

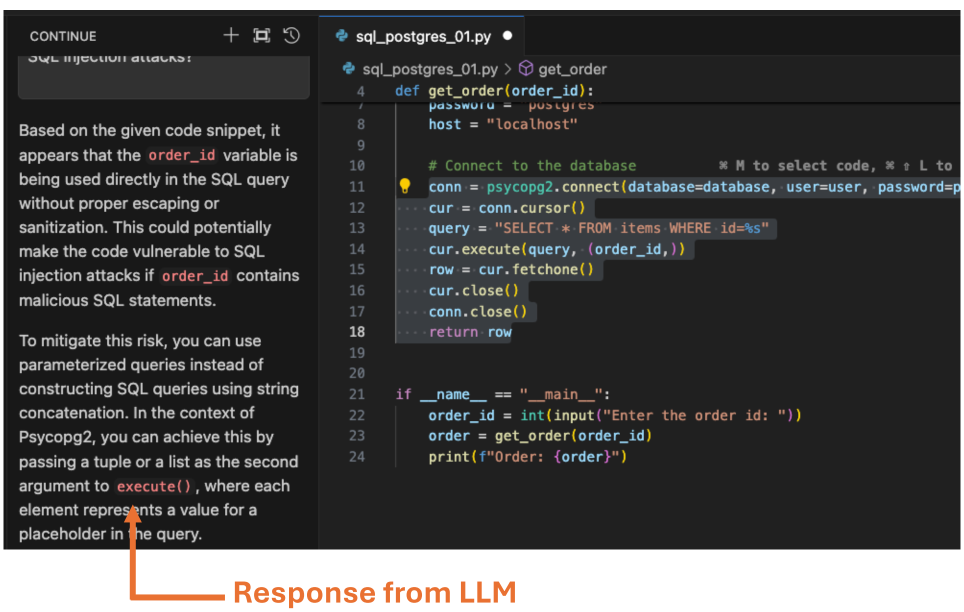
While the tool should not replace existing time-proven security offerings, it does offer a novel approach to aid security professionals and application engineers in identifying application-level security concerns. This type of implementation where the UI layer manages rapid requests and displaying responses has allowed for implementing new features with little effort.

The advent of running LLMs from a local workstation or laptop paved the way for cost effective research, debugging, and testing of this type of project. Running LLMs locally also offer the option to clone repos to a local file system and run the tool at a later time while disconnected from the internet (e.g., during travel).

Background

I was recently performing a security code review. While manually inspecting part of a file in an IDE, I decided to use an LLM extension to ask if it could spot a vulnerability in a code block and hit '*Enter*'.





I realized that LLMs were becoming adept at spotting security and other issues in code and found the "second set of eyes" extremely useful. After going through this process of selecting code blocks and asking questions, I realized that I had quite a few files to go and thought, “it would be great if there was a way to automate this to auto-ask these questions for all code blocks in a large collection of files.”

I then began building an application that pulls all files in a code repository, breaking them into blocks that an LLM could handle one at a time, then responds with a security evaluation.

After experimenting with different models and prompts, and adding a UI, I realized that this tool could be very beneficial for other use cases too.

Use Cases

<u>Security Researchers</u>

- Testing open-source applications.
- Quick reconnaissance prior to white box testing.
- Preparation for bug bounty programs.

<u>Red Teams</u>

- Quick inspection of internal application to support red team exercises.

<u>Blue Teams</u>

- Inspect repo of in-scope applications to support defect discovery/remediation.

<u>DevSecOps Team Members</u>

- Inspect entire local copy of solution prior to committing changes to centralized source control.

Problem Statement

<u>Existing LLM based tools:</u>

Current solutions that employ LLM technology require someone to have the file open and select a segment of the code in a size that an LLM will accept. For some, this may be in a code editor with an extension that sends the code block to an API. For others, this may mean copying the code block and pasting it into an LLM interface. In either case, this can be a laborious process to cover lengthy files. This process also makes it easy to leave behind the many potential files that were created prior to these advancements.

Existing non-LLM based tools:
Organizations as well as independent researchers may have limited licenses for analysis tools that would leave little or no options for evaluating every repository in an automated way. Many of these tools offer generic or no advice on how to address a finding without providing a clear before and after of the code to remediate.

Research Process

In order to test the breadth of finding types, I created several files with purposely vulnerable code and at least one file for each programming language I wanted the tool to support. Here is a list of vulnerabilities I placed in the files:

- Command Injection
- CSRF
- File Injection
- Hardcoded secrets
- Insecure deserialization
- Insecure direct object reference
- Lack of error handling
- Lack of input validation
- SQL injection
- XSS

I initially set out to implement a tool that utilized a technique that would embed the contents of each file and then place the embedding and files in a local vector store followed by running quires for each vulnerability type against the entire vector store (e.g., "What are the files that are vulnerable to SQL injection?"). This approach had several issues:

- The embedding step ran very slowly taking upwards of 5 minutes for a sample repo consisting of only 7 files.
- The results almost never told me the file name where the findings were (even with a fair amount of coaxing in the query explicitly asking to return file names).
- After 4 to 5 queries, the tool was not returning results.
- Even if I were to overcome the issues above, I realized that I may not be able to come up with a comprehensive list of queries to cover all vulnerability types.

I then decided to take a different approach and queue up a query for each code block and build an interface that dynamically loads each result as they come in.

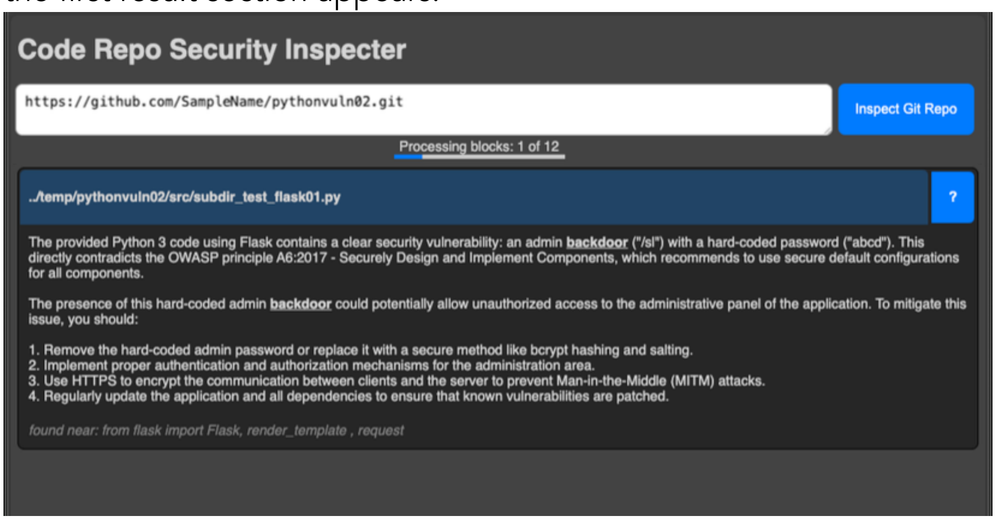
A New Tool

Let's begin with an example workflow utilizing the running tool.

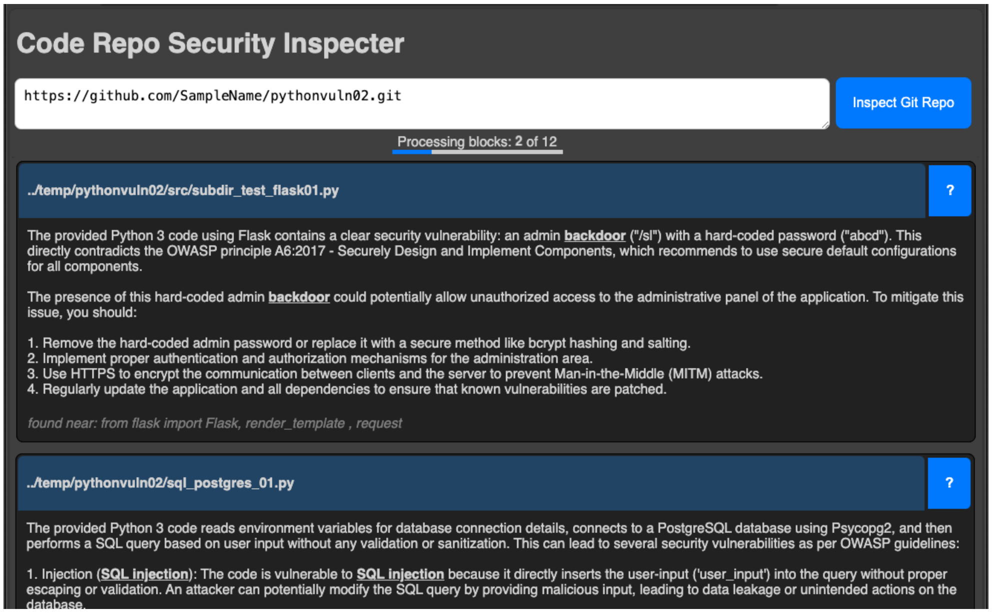
1. Provide the URL to a git repository.
   Click the "Inspect Git Repo" button.

   (alternatively, you could provide the path to cloned repo on the local device)

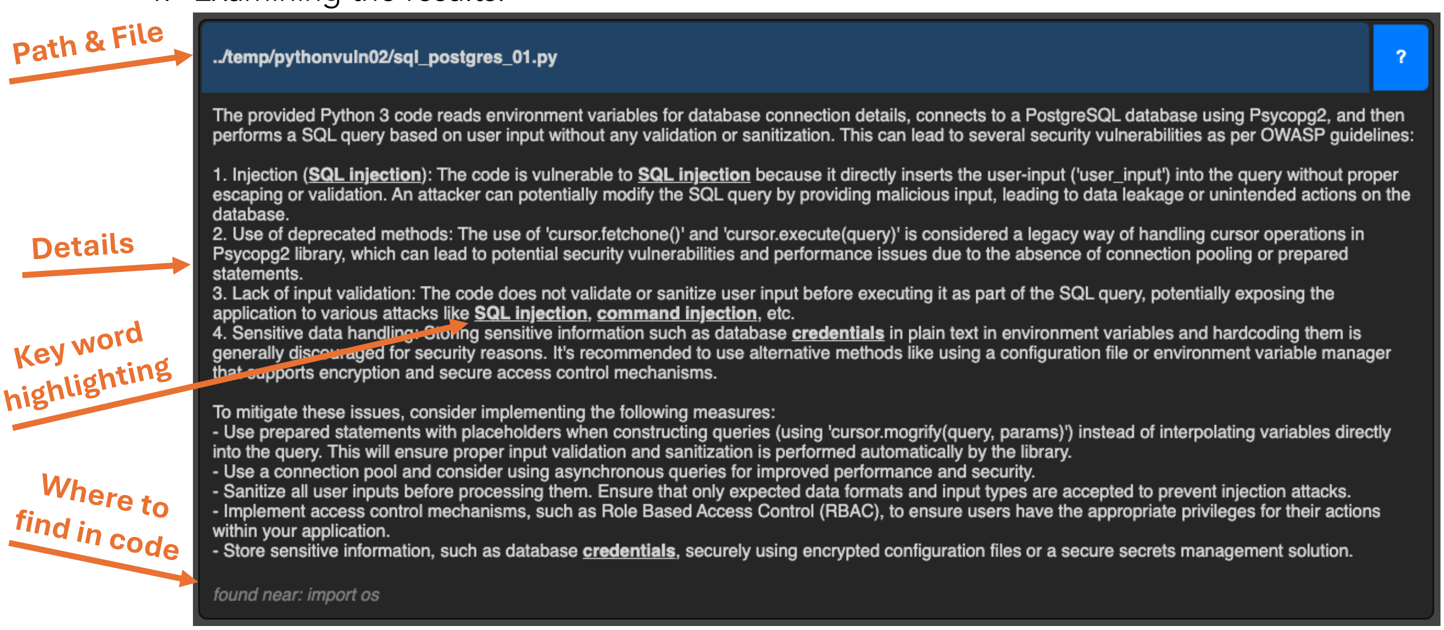
2. An indicator shows that 12 code blocks are being processed and the first result section appears.



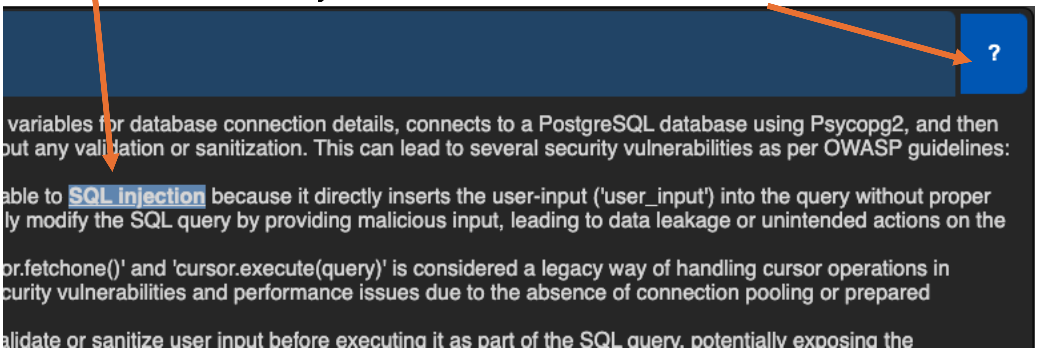
3. As more files or blocks are processed, they will begin to show below the first blocks rendered.



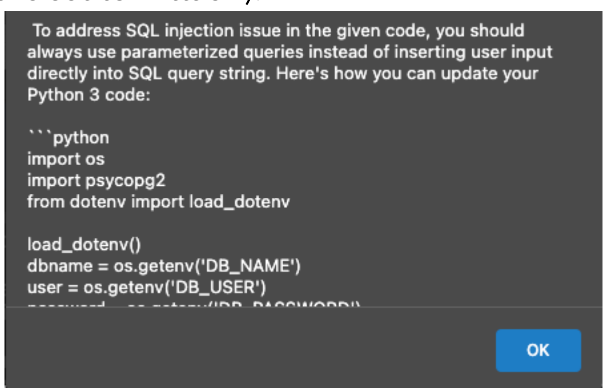
4. Examining the results.



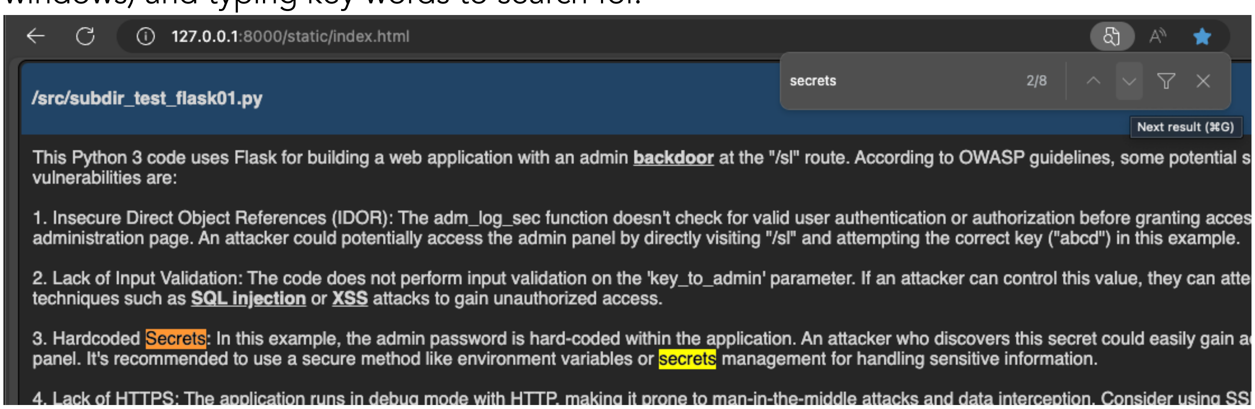
5. Select some text with your cursor in the detail and click the "?" button.



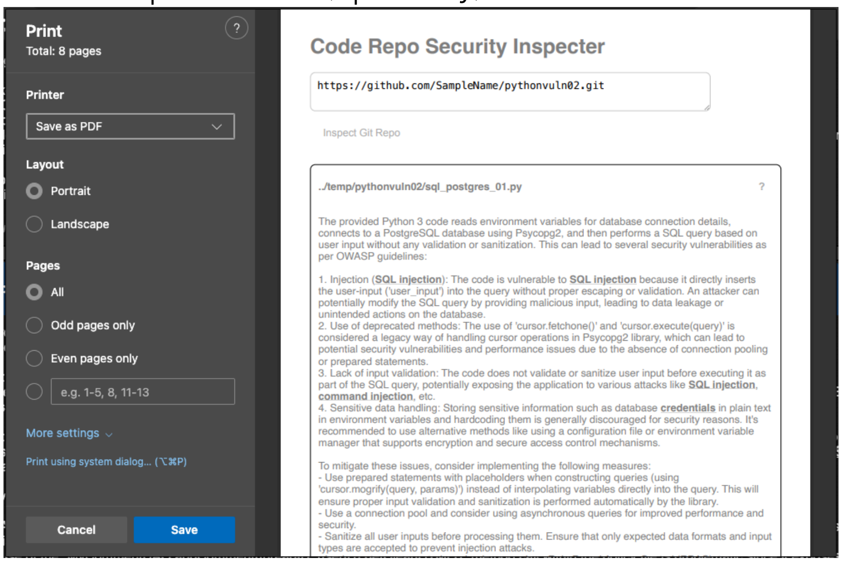
6. A scrollable message box will appear showing you an updated version of the code you can use to address the issue you selected (or potential ways to exploit if the mode is set to 'Attack').



7. You can search the result sections by pressing command+f (or control+f in windows) and typing key words to search for.



8. Print or export to PDF (optionally).



Requirements and Setup

- Python 3 (tested on Python version 3.11)
- A browser allowing JavaScript (tested on Edge version 122)
- Access to an LLM API (tested with OpenAI and Ollama version 0.1.25)
- Note: For running LLMs locally such as Ollama, this may work on a variety of machines; however, a realistic baseline for ensuring quality execution speeds would include 16 GB or more of RAM and GPU acceleration (e.g., Apple M1/M2).

1. Use Python's '`pip`' utility to install the required packages from the `requirements.txt` file.

2. Open the `.env` file.

```
.env
API_KEY = 'if-using-openai-or-other-put-key-here'
BASE_URL = 'http://localhost:11434/v1'
LLM_MODEL = 'mistral:7b'
QUESTION_BUTTON_MODE = 'DEFEND' # or 'ATTACK'
```

   a. Ensure that the `BASE_URL` setting points to the API you plan to use.
   b. Specify the `LLM_MODEL` you plan to use.
   c. If using a service that requires a key (e.g., OpenAI), set the `API_KEY` setting with the key provided by your service.
   d. Optionally, switch the question button mode from providing remediation help ('`DEFEND`') to how the code block could be exploited ('`ATTACK`').

NOTE: Attack mode may require using an LLM that allows this type of information to be returned; these are often listed as 'uncensored' and may necessitate requesting permission from your organization, if applicable.

3. Start a 'uvicorn' web server from the directory containing the `main.py` file.
   a. (e.g., `uvicorn main:app --port 8000`)

4. Navigate your browser to this location on your freshly started 'uvicorn' server:
   a. http://127.0.0.1:8000/static/index.html

About Models

Currently there are many options to choose from both categories of models, commercial and free. Each option comes with their own strengths, weaknesses, and tradeoffs. You will want to select a model that best suits your needs and potential restrictions. The table below maps my observations of using several different models during the creation and testing of the tool.

Model	Parameter Size	Findings	Speed (per block)	Price (per block)
Mixtral-8X7	47 B*	22 / 24	2 s	$0.0003
Nous-Hermes-2S	11 B	21 / 24	57 s	--
GPT-4 Turbo	--	20 / 24	25 s	$0.006
Gemma	7 B	20 / 24	13 s	--
Mistral	7 B	20 / 24	60 s	--
Deepseek-coder	7 B	18 / 24	12 s	--
GPT-3.5 Turbo	175 B	18 / 24	2 s	$0.0003
stablelm-zephyr	3 B	15 / 24	5 s	--
TinyLlama	1 B	8 / 24	5 s	--

* *Mixtral has 46.7B total parameters but only uses 12.9B parameters per token*

It's not surprising that the models built with more parameters scored higher in finding vulnerabilities out of the 24 test cases. I tried several other models in the 1 to 11 billion parameter range for running locally. I did not list them above due to not performing on par with the model listed above with the same parameter size. Many of the sub-three billion parameter models tested did not produce any findings, output gibberish, or were stuck in a repeating output loop.

NOTE: There are models available that are listed as 'optimized for code'. Some of these models may only include support for certain programming languages and may not produce results if the code you are trying to inspect was not used in the creation of that specific model.

At the time of this writing, the GPT-4 Turbo model produces results in a way that it wants to point out potential issues and best practices to keep in mind (e.g., "Be sure to sanitize input") even when asking it to only provide feedback for found vulnerabilities.

New models are released nearly on a daily basis, and I hope to update this white paper in the coming weeks with additional models and providers (e.g., Claude 3, StarCoder2).

Languages Supported

Currently the tool supports the following programming languages. Please see the 'Tips on Customization' section of this document for how to add additional languages.

- C#
- Go
- Java
- JavaScript
- PHP
- Python
- TypeScript

Lessons Learned

I found that simply asking the models to find vulnerabilities in a given block of code would often result in missing findings.

Including descriptions of code vulnerability types (e.g., SQL injection, XSS) in the query often resulted in even more missing findings than not listing out the types.

Modifying the code to include the associated programming language in each prompt dramatically improved the results ("… analyze the following PHP code for …").

Finally, adding the phrase, "… using OWASP guidelines …", in the prompt (when combined with a high performing model) resulted in the tool finding all the vulnerabilities I placed in the test files nearly every test.

```
messages=[
    {
        'role': 'user',
        'content': 'Give a short and concise response. Analyze the following ' + code_language +
        ' code for security vulnerabilities using OWASP guidelines and report any issues found in a brief summary. \
        + texts_global[inspectionTask.iterationNum].page_content
    }
],
model = os.getenv("LLM_MODEL"), # "stablelm-zephyr:latest", "gpt-3.5-turbo",
)
```

This combination also produced responses that included background information about the code block (e.g., "… connects to a PostgreSQL database using"), and even included advice on code quality or optimizations that were not directly security issues.

Tips on Customization

<u>Adding new languages:</u>

The Python library used for splitting files into code blocks works best when you specify the language for both the '`splitter`' and '`parser`':

```
if ext == "py":
    loader = GenericLoader.from_filesystem(
        repo_path + "/",
        glob="**/*",
        suffixes=[".py"],
        exclude=["**/non-utf8-encoding.py"],
        parser=LanguageParser(language=Language.PYTHON, parser_threshold=500),
    )
    documents = loader.load()
    python_splitter = RecursiveCharacterTextSplitter.from_language(
        language=Language.PYTHON, chunk_size=2000, chunk_overlap=200
    )
    texts = python_splitter.split_documents(documents)
```

However, the 'parser' does not support all the same languages that the 'splitter' does and will throw an error for some languages when trying to use it. To get around this, you must omit the specified language from the '`parser`':

```
elif ext == "java":
    loader = GenericLoader.from_filesystem(
        repo_path + "/",
        glob="**/*",
        suffixes=[".java"],
        exclude=["**/non-utf8-encoding.java"],
        parser=LanguageParser(parser_threshold=500),
    )
    documents = loader.load()
    java_splitter = RecursiveCharacterTextSplitter.from_language(
        language=Language.JAVA, chunk_size=2000, chunk_overlap=200
    )
    texts = java_splitter.split_documents(documents)
    texts_all_types = texts_all_types + texts
```

Adding / Changing Keywords to Highlight
In the UI code (`index.html`), you can modify the terms and phrases in the '`wordsToHighlight`' function:

```
<script>

    var wordsToHighlight = ['SQL Injection', 'File Injection', 'Command Injection', 'Code Inject

    function highlightWordsInTd(element) {
```

Saving Money on API Usage Fees
When making changes and additions to the code you will likely make many calls to LLM APIs in the process of debugging and testing your work. If you do not have the option to use a free LLM (e.g., Ollama), here are some options to greatly reduce your costs:

- Implement a mock API call that returns some sample data in the same format that an actual API call would and temporarily use the mock function until you are comfortable with the changes.
- Switch to a less expensive model during testing (e.g., GPT-3.5 Turbo vs. GPT-4).
- Use a test repo with only a small number of files.

Limitations

The ability to run the tool against a very large codebase will depend greatly on factors such as the amount of RAM in your workstation, and what other processes are currently running. You may need to consider running the tool in multiple iterations once each for top level directories.

As with other instances of using LLMs, it's possible to get different variations of responses even when sending the same prompt and data.

The tool only inspects file types listed in the '`extensions`' variable. This means that if a filetype with an extension that is not in the list is encountered, the tool will not inspect it.

Some models may come with a license that prohibits commercial use without obtaining permission from the creators.

Roadmap

I'm currently investigating/working on the following feature additions:

- Excluding, hiding, or dimming responses with a sentiment of "*no findings in this block*".
- Confirmation message prior to processing batches over a certain size ("*You are about to make 100 or more calls to the LLM provider, do you want to continue?*")
- Button for implementing "Import to Jira (or other tracking system) backlog".

Privacy / Compliance Considerations

It should be noted that using LLM tools hosted by external companies could lead to you uploading a company's source code to an entity that you do not have an agreement with to safeguard your data. Aside from the external provider having source code that should not have been shared, the data may be used in a way that could lead to exposing this data to other users of the system. To avoid this concern, you may need to consider utilizing one of the following:

- Entering an agreement with a provider (e.g., Azure OpenAI Service)
- A company hosted solution that has been vetted for privacy and security.
- Locally running LLM.

References

- GitHub Copilot
  https://github.com/features/copilot
- LangChain Retrieval
  https://python.langchain.com/docs/modules/data_connection/
- Mixtral of experts
  https://mistral.ai/news/mixtral-of-experts
- Ollama
  https://github.com/ollama/ollama
- OpenAI API Documentation
  https://platform.openai.com/docs/overview
- Python 3
  https://www.python.org/downloads/
- uvicorn
  https://pypi.org/project/uvicorn/