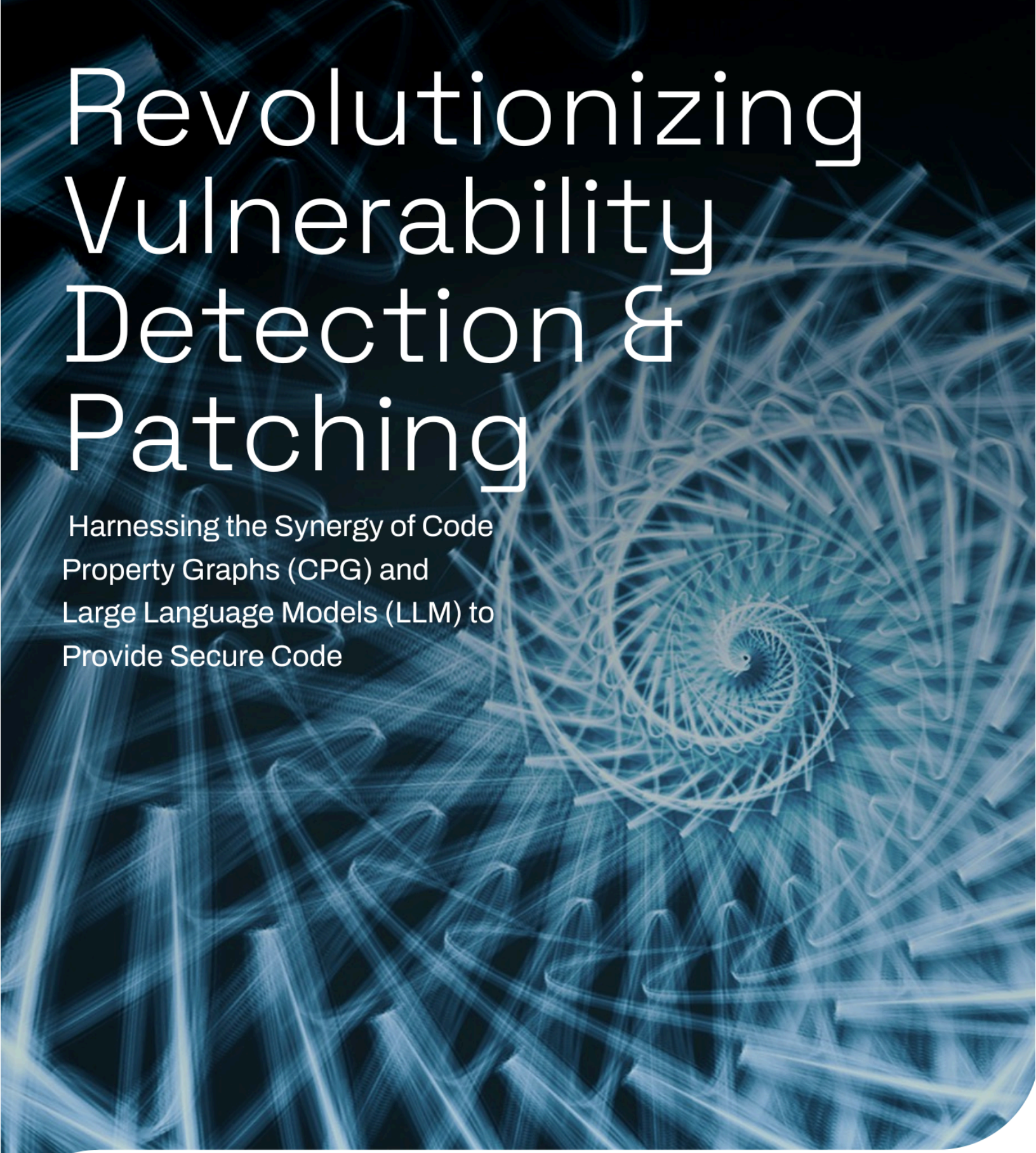


Revolutionizing Vulnerability Detection & Patching

Harnessing the Synergy of Code Property Graphs (CPG) and Large Language Models (LLM) to Provide Secure Code

qwiet AI

In the ever-evolving landscape of software development, ensuring the security of applications has become a paramount concern. As cyber threats continue to grow in sophistication, it is crucial for developers and security professionals to stay ahead of the curve. This article explores a groundbreaking approach that combines the power of Code Property Graphs (CPGs) and Large Language Models (LLMs) to revolutionize vulnerability detection and patching processes.

TABLE OF CONTENTS

Understanding Code Property Graphs

Code Property Graphs are a powerful tool for representing and analyzing the intricate relationships within software code. By transforming source code into a graph structure, CPGs enable developers to gain deep insights into the flow of data, control dependencies, and other critical aspects of the codebase. This granular level of understanding is essential for identifying potential vulnerabilities that might otherwise go unnoticed.

At its core, a Code Property Graph is a unified representation of a software system that combines various aspects of the code, such as syntax, control flow, and data flow, into a single graph structure. Each node in the graph represents a code element (e.g., variables, functions, statements), while the edges represent the relationships between these elements.

Syntax Trees

The Building Blocks of CPGs The foundation of a CPG lies in the syntax tree, which captures the syntactic structure of the code. The syntax tree represents the hierarchical arrangement of code elements, such as functions, classes, and statements. By incorporating the syntax tree into the CPG, we can gain a granular understanding of the code's structure and identify potential vulnerabilities that arise from syntactic patterns.

For example, consider a code snippet that concatenates user input directly into an SQL query without proper sanitization:

```
sql_query = "SELECT * FROM users WHERE username = '" + user_input + "';"
```

Control Flow

Navigating the Execution Paths Control flow analysis is a crucial component of CPGs, as it helps in understanding the possible execution paths that the code can take. By representing control flow as edges in the graph, CPGs enable us to track the flow of execution from one code element to another.

Control flow analysis can uncover vulnerabilities that arise from improper handling of control structures, such as conditional statements or loops. For instance, consider a code snippet that performs authentication:

```
if (user.isAuthenticated())
          { // Grant access to sensitive resources }
else
          { // Deny access }
```

By analyzing the control flow, we can identify if there are any paths that bypass the authentication check or if there are any unintended paths that lead to unauthorized access.

Data Flow

Tracking the Flow of Information Data flow analysis is another essential aspect of CPGs, as it allows us to track how data moves through the codebase. By representing data dependencies as edges in the graph, CPGs enable us to trace the flow of information from its origin to its eventual use.

Data flow analysis is particularly useful in identifying vulnerabilities related to improper handling of user input or sensitive data. For example, consider a code snippet that reads user input and uses it to construct a file path:

```
String filePath = "/user/data/" + userInput; File file = new File(filePath);
```

By analyzing the data flow, we can determine if the user input is properly validated or sanitized before being used in the file path construction. If not, it could lead to vulnerabilities such as path traversal or arbitrary file access.

Extracting Exploitable Paths By combining the information from syntax trees, control flow, and data flow, CPGs provide a powerful tool for identifying potential vulnerabilities and extracting exploitable paths.

We can perform graph traversals and queries on the CPG to extract exploitable paths. For instance, we can search for paths that originate from user input sources (e.g., HTTP requests, command-line arguments) and flow into sensitive sinks (e.g., database queries, file system operations) without proper validation or sanitization. These paths represent potential attack vectors that an attacker could exploit.

Here's an example query that identifies paths from user input to SQL queries:

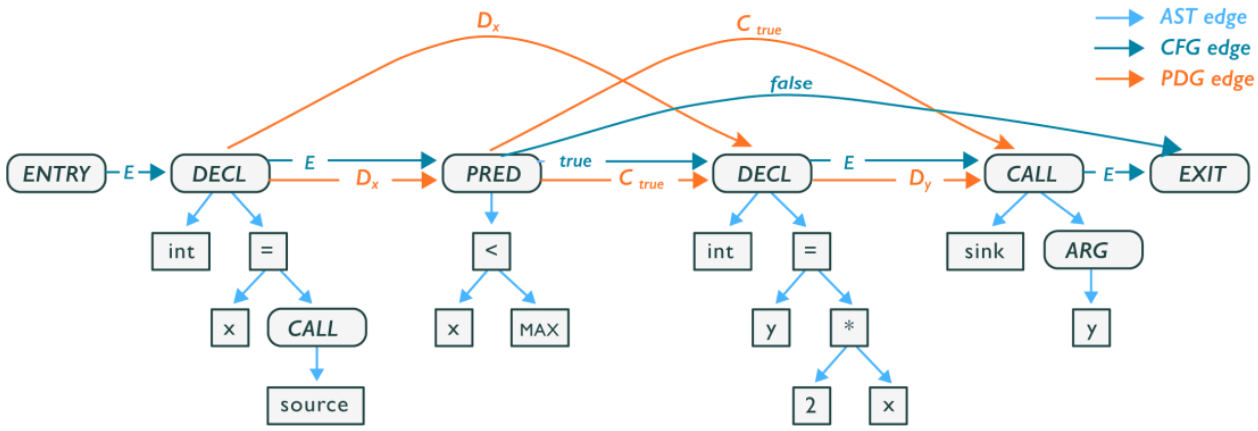
```
START n=node:UserInput()

MATCH (n)-[*]->(m:SQLQuery)

RETURN n, m
```

This query starts from nodes representing user input and follows any path that leads to nodes representing SQL queries. The returned paths highlight the flow of user input into SQL queries, allowing us to identify potential SQL injection vulnerabilities.

By leveraging the rich information captured in CPGs, we can perform complex analyses and uncover vulnerabilities that might be difficult to detect through manual code review or traditional static analysis techniques.



(fig a - representation of a Code Property Graph - src Wikipedia)

For more in-depth information on Code Property Graphs, please see the original paper by Fabian Yamaguchi.

The Power of Graph-based Machine Learning (ML) and Deep Learning Techniques Over CPG

The graph-based nature of CPGs makes them an ideal candidate for applying Graph-based Machine Learning and Deep Learning techniques. These techniques can help uncover hidden patterns, predict potential vulnerabilities, and provide insights that traditional analysis methods might overlook.

Node Classification

Node classification is a fundamental task in graph-based machine learning, where the goal is to assign labels or categories to nodes based on their features and the structure of the graph. In the context of CPGs, node classification can be used to identify code elements that are likely to contain vulnerabilities.

For example, let's consider a CPG where nodes represent functions in a codebase. By extracting features such as the number of parameters, cyclomatic complexity, and the presence of certain API calls, we can train a machine learning model to classify nodes as "vulnerable" or "safe." This classification can help prioritize code review efforts and focus attention on the most critical parts of the codebase.

One popular approach for node classification is using Graph Neural Networks (GNNs). GNNs can learn representations of nodes by aggregating information from their neighboring nodes and edges. For instance, a Graph Convolutional Network (GCN) can be used to learn embeddings for each function node in the CPG, capturing both the local code features and the structural information of the graph. These embeddings can then be fed into a classifier to predict the vulnerability likelihood of each function.

Link Prediction

Link prediction is another powerful application of graph-based machine learning on CPGs. The goal of link prediction is to estimate the likelihood of the existence of an edge between two nodes based on the graph's structure and node attributes. In the context of CPGs, link prediction can be used to identify potential data flow or control flow dependencies that may lead to vulnerabilities.

For example, consider a scenario where a zero-day vulnerability is discovered in a widely-used library. The vulnerability arises from an attacker-controlled input being passed to a sensitive function

without proper validation. By analyzing the CPG of the affected codebase, we can use link prediction techniques to identify other potential instances of this vulnerability.

To achieve this, we can train a link prediction model using the existing edges in the CPG as positive examples and generate negative examples by randomly sampling non-existent edges. The model learns to predict the likelihood of an edge existing between two nodes based on their attributes and the graph's structure. In the case of the zero-day vulnerability, we can use the model to predict potential data flow paths from user input sources to sensitive functions, helping identify other potential instances of the vulnerability.

Graph Embedding Techniques

Graph embedding techniques play a crucial role in enabling graph-based machine learning on CPGs. These techniques aim to learn low-dimensional vector representations (embeddings) of nodes or entire graphs, capturing their structural and semantic information.

One popular graph embedding technique is Node2Vec, which learns embeddings by performing random walks on the graph and optimizing a skip-gram objective. By applying Node2Vec to a CPG, we can learn embeddings that capture the code structure and semantic relationships between code elements. These embeddings can then be used as input features for downstream tasks such as node classification or link prediction.

Another powerful approach is Graph Convolutional Networks (GCNs), which learn node embeddings by aggregating information from neighboring nodes using convolutional operations. GCNs can capture both the local code features and the graph structure, making them well-suited for vulnerability detection tasks on CPGs.

Example: Detecting a Zero-Day Vulnerability using CPG and Graph-based ML Let's consider a real-world example to illustrate the potential of leveraging CPGs with graph-based machine learning for detecting a zero-day vulnerability.

Suppose a critical zero-day vulnerability is discovered in a popular open-source library. The vulnerability allows an attacker to perform a remote code execution by exploiting a flaw in the library's deserialization mechanism.

To detect potential instances of this vulnerability in a large codebase, we can follow these steps:

1. Construct a CPG of the codebase, representing code elements (e.g., functions, variables) as nodes and their relationships (e.g., function calls, data flow) as edges.
2. Identify the relevant code patterns and characteristics associated with the vulnerability, such as the use of the vulnerable deserialization function, the presence of user-controlled inputs, and the data flow paths leading to the vulnerable code.

3. Train a graph-based machine learning model, such as a GCN, on the CPG. The model learns to classify nodes as "vulnerable" or "safe" based on their code features and the graph structure.
4. Apply the trained model to the CPG of the codebase, identifying potential instances of the vulnerability. The model can flag functions or code snippets that exhibit characteristics similar to those of known vulnerable patterns.
5. Perform manual code review on the flagged instances to confirm the presence of the vulnerability and take appropriate remediation actions.

By leveraging the CPG's graph-based substrate and applying graph-based machine learning techniques, we can efficiently identify potential instances of a zero-day vulnerability across a large codebase. This approach can significantly reduce the time and effort required for manual code review and help prioritize remediation efforts.

The Power of Generative AI - Large Language Models

Large Language Models, such as GPT-3, have emerged as a game-changer in the field of natural language processing. These models, trained on vast amounts of text data, possess an extraordinary ability to understand and generate human-like text. By harnessing the power of LLMs, we can leverage their linguistic capabilities to assist in the generation of code patches and security fixes.

Synergizing CPGs, Predictive AI on CPGs and LLMs

The true magic happens when we combine the insights derived from Code Property Graphs with the generative capabilities of Large Language Models. By training LLMs on data extracted from CPGs, we can create a powerful system that not only detects vulnerabilities with high precision but also suggests targeted patches to mitigate those vulnerabilities.

Fine Tuned Training

The training process involves carefully preparing and structuring the CPG data, selecting relevant features, and annotating code snippets with appropriate labels. This curated data is then fed into the LLM, enabling it to learn the intricacies of code structure, dependencies, and vulnerability patterns.

Few-Shot Prompting

The Key to Targeted Patch Generation Once the LLM is trained, the real power lies in its ability to generate accurate and contextually relevant code patches. This is achieved through a technique called few-shot prompting. By providing the model with a few examples of vulnerabilities and their corresponding fixes, along with specific details from the CPG analysis, we can guide the LLM to generate highly targeted patches.

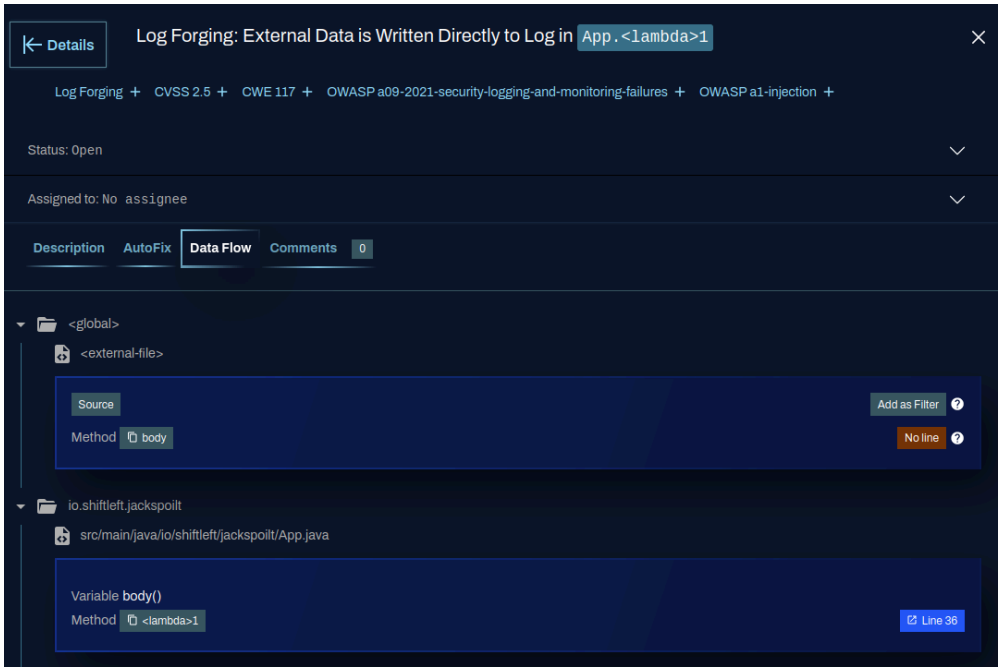
The key to effective few-shot prompting lies in the precision and relevance of the examples provided. By incorporating detailed context from the CPG, such as the specific vulnerability type, location in the code, and data flow information, the LLM can generate patches tailored to each vulnerability's unique characteristics.

Few-Shot Prompting with Precision

Once a vulnerability is identified via CPG, the next step involves guiding an LLM to suggest an appropriate patch. Here's how this can be effectively accomplished using few-shot prompting:

1. Example: Construct a few-shot prompt that includes examples of similar vulnerabilities, paired with secure coding practices. For example:
    - Example 1: "User input directly passed to exec() function. Vulnerable code: `exec('ping ' + user_input)`. Fixed code: `exec('ping ' + sanitize(user_input))`."
    - Example 2: "User input used in file path without validation. Vulnerable code: `open(user_input, 'r')`. Fixed code: `open(sanitize_path(user_input), 'r')`."
2. Vulnerability Context: Provide the LLM with the specific context of the detected vulnerability using CPG analysis. For instance, "Detected command injection risk where user input from form field user_data flows directly into exec() function."
3. Prompt for Solution: Ask the LLM to generate a patch based on the example structure and the specific vulnerability context: "Given the above examples and the detected vulnerability in the exec() function call using user_data, suggest a secure patch."
4. Patch Generation: The LLM uses the detailed input from the few-shot examples and the specifics of the vulnerability to suggest a precise fix, such as introducing a sanitization function specific to command injections: exec('ping ' + sanitize(user_input)).

How does this work in practice? In the example below, we see the CPG highlighting the data flow of a Log Forging vulnerability in an application.



The resulting fix based on Few-Shot prompting combined with the detailed CPG information provides a thorough fix that addresses the issue.

```java
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.commons.io.FileUtils;
import java.io.File;
import java.io.IOException;

public class App {
    private static final ObjectMapper mapper = new ObjectMapper();
    private static final File logFile = new File("application.log");

    public static void main(String[] args) throws IOException {
        // Assume that `request` and `response` are already defined
        // Assume that `deserialize` and `serializer` are already defined

        logRequest(request);

        Account account = deserialize(request);
        if (account != null) {
            Account res = addAccount(account);
            response.status(201);
            response.body(mapper.writeValueAsString(res));
        } else {
            response.status(400);
            response.body("Invalid content");
        }
    }

    private static void logRequest(HttpRequest request) throws IOException {
        FileUtils.writeStringToFile(logFile, "/accounts -> " + request.body(), true);
    }

    private static Account deserialize(HttpRequest request) throws IOException {
        return mapper.readValue(request.body(), Account.class);
    }

    private static Account addAccount(Account account) {
        // Add account logic here
    }
}
```

Steps to address the vulnerability:

1. Import necessary libraries for JSON serialization and file handling.
2. Create a static `ObjectMapper` for JSON serialization and a static `File` object for the log file.
3. Modify the `main` method to use the `logRequest` method to log the request body.
4. Create a `deserialize` method to convert the request body into an `Account` object.
5. Implement the `addAccount` method to handle adding the account to the system.

Conclusion

The integration of Code Property Graphs and Large Language Models represents a significant leap forward in vulnerability detection and patching. By leveraging the deep insights provided by CPGs and the generative capabilities of LLMs, we can create a powerful system that not only identifies vulnerabilities with high precision but also suggests accurate and contextually relevant patches.

This synergistic approach has the potential to revolutionize the way we approach software security. It accelerates the vulnerability management process, reduces manual effort, and ultimately enhances the overall security posture of software applications. As we continue to refine and advance these techniques, we can look forward to a future where software vulnerabilities are swiftly detected and patched, ensuring a safer digital ecosystem for all.

About Qwiet AI

Qwiet AI is the AI-enhanced application security testing platform that provides SAST, SCA, Container Scanning, and Secrets Detection all in one fast and comprehensive scan. Qwiet AI customers benefit from targeted results with scans that are 10x faster and 12x more accurate than traditional application security tools.