Politechnika Wrocławska
Wydział informatyki i Telekomunikacji

Kierunek: **Cyberbezpieczeństwo (CBE)**
Specjalność: **Bezpieczeństwo danych (CBD)**

PRACA DYPLOMOWA
INŻYNIERSKA

Zastosowanie dużych modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa i podatności w kodzie aplikacji webowych

Patryk Fidler

Opiekun pracy
Dr. hab. inż. Maciej Piasecki

Słowa kluczowe: modele językowe, Sztuczna Inteligencja, statyczna analiza kodu

WROCŁAW 2024

STRESZCZENIE

Praca inżynierska zatytułowana "Zastosowanie dużych modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa i podatności w kodzie aplikacji webowych" koncentruje się na zastosowaniu zaawansowanych modeli językowych, takich jak GPT-3.5, GPT-4, a w przyszłości także modele otwartoźródłowe, takie jak Mistral, do automatycznego wykrywania i naprawiania błędów bezpieczeństwa w kodzie oprogramowania i aplikacji webowych.

Motywacja tej pracy wynika z rosnącej roli dużych modeli językowych (LLM) w różnych dziedzinach, w tym w cyberbezpieczeństwie. W kontekście tych działań, badana jest możliwość wykorzystania tych modeli do wykrywania i naprawiania podatności takich jak XSS, SQL Injection, CSRF, Buffer Overflow i tym podobne.

Punktem wyjścia dla pracy dyplomowej jest artykuł napisany w 2021 roku "Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?" - Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, Brendan Dolan-Gavitt (https://arxiv.org/pdf/2112.02125v1.pdf). Autorzy podkreślają znaczący potencjał tych modeli, a niniejsza praca ma na celu kontynuację tych badań i poszerzenie ich zakresu.

Planowane działania obejmują przygotowanie zbioru danych zawierającego podatne bazy kodu źródłowego, testowanie zdolności detekcji błędów przez modele językowe OpenAI, oraz porównanie tych wyników z istniejącymi rozwiązaniami, oferowanymi przez firmę Snyk.

Szczególny nacisk zostanie położony na wykorzystanie technik uczenia się w kontekście (in-context learning) oraz generowanie wspomagane pobieraniem (RAG - Retrieval Augmented Generation), które mogą pomóc w udoskonaleniu detekcji i wyników, nawet przy ograniczonych zasobach. Praca przewiduje implementację autonomicznego agenta AI zdolnego do analizy kodu, wykonania testów bezpieczeństwa i podejmowania decyzji na podstawie wyników tych testów i kontekstu.

Opcjonalnie, badane będą możliwości detekcji błędów przez otwarte modele językowe, a w dalszej perspektywie, możliwości specjalizacji modeli w zakresie cyberbezpieczeństwa za pomocą fine-tuning'u. Wszystkie te działania mają na celu nie tylko badanie, ale także poprawienie możliwości LLM w kontekście cyberbezpieczeństwa.

Głównym celem pracy jest zwiększenie świadomości na temat potencjału dużych

modeli językowych w cyberbezpieczeństwie oraz proponowanie praktycznych rozwiązań, które mogą pomóc programistom w tworzeniu bardziej bezpiecznych aplikacji.

ABSTRACT

The engineering thesis titled "Application of Large Language Models for Detecting and Fixing Security Bugs and Vulnerabilities in Web Application Code" focuses on the utilization of advanced language models, such as GPT-3.5, GPT-4, and if possible, Falcon, for automated detection and rectification of security bugs in the code of software and web applications.

The motivation for this work stems from the growing role of Large Language Models (LLMs) in various fields, including cybersecurity. In the context of these efforts, the possibility of using these models to detect and fix vulnerabilities such as XSS, SQL Injection, CSRF, Buffer Overflow, and the like is being investigated.

The starting point for this dissertation is the 2021 article "Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?" by Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, Brendan Dolan-Gavitt (https://arxiv.org/pdf/2112.02125v1.pdf). The authors emphasize the significant potential of these models, and this work aims to continue this research and broaden its scope.

Planned activities include preparing a dataset containing vulnerable source code databases, testing the error detection capabilities of OpenAI language models, and comparing these results with existing solutions offered by Snyk.

Particular emphasis will be placed on the use of soft-prompting and in-context learning techniques, which can help improve detection and results, even with limited resources. The work also envisages the implementation of an autonomous AI agent capable of analyzing code, performing security tests, and making decisions based on the results of these tests and context.

Optionally, the possibilities of error detection by open language models will be explored, and in the longer term, the possibilities of specializing models in the field of cybersecurity through fine-tuning. All these actions aim not only to understand but also to improve the capabilities of LLMs in the context of cybersecurity.

The main goal of the work is to increase awareness of the potential of large language models in cybersecurity and to propose practical solutions that can help developers create more secure applications.

Spis treści

WPROWADZENIE

Niniejsza praca inżynierska nosi tytuł "Zastosowanie dużych modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa i podatności w kodzie aplikacji webowych". Celem tej pracy jest zbadanie, w jaki sposób zaawansowane modele językowe, takie jak GPT-3.5, GPT-4 oraz modele otwarto-źródłowe - Mistral 7B, Falcon-7B-instruct, mogą być wykorzystane do automatycznego wykrywania i naprawiania błędów bezpieczeństwa w kodzie oprogramowania i aplikacji webowych. W tym celu zostanie opracowane i zaimplementowane narzędzie do statycznej analizy kodu, które będzie wykorzystywać modele językowe do detekcji podatności i naprawy błędów. Narzędzie to zostanie przetestowane i porównane z innymi rozwiązaniami, takimi jak Snyk, które oferują podobne funkcjonalności, a także z tradycyjnymi skanerami podatności. W pracy zostaną przedstawione wyniki badań, które mają na celu odpowiedzieć na pytanie, czy modele językowe mogą być wykorzystane do tego celu, oraz jak skuteczne są one w porównaniu z innymi rozwiązaniami. W ramach pracy zostaną również zbadane ograniczenia i wyzwania związane z wykorzystaniem tych technologii w kontekście cyberbezpieczeństwa.

PYTANIA BADAWCZE

W ramach pracy stawiam następujące pytania badawcze:

1. Czy duże modele językowe mogą być wykorzystane do wykrywania i naprawiania błędów bezpieczeństwa w kodzie aplikacji webowych?
2. Jak skuteczne są te modele w porównaniu z innymi rozwiązaniami?
3. W jakim stopniu metody wzbogacania generacji (RAG) i uczenia się w kontekście (in-context learning) mogą poprawić skuteczność tych modeli?
4. Jakie są ograniczenia i wyzwania związane z wykorzystaniem tych technologii w kontekście cyberbezpieczeństwa?

HIPOTEZY

Hipotezy pracy to:

1. Duże modele językowe, dzięki swojej zdolności do analizy i generowania kodu, mogą skutecznie identyfikować i naprawiać błędy bezpieczeństwa w kodzie źródłowym.

2. Mimo obiecującego potencjału, modele te mogą napotykać ograniczenia, szczególnie w bardziej złożonych i specyficznych scenariuszach związanych z cyberbezpieczeństwem.

UZASADNIENIE TYTUŁU

Tytuł pracy został dobrany tak, aby odzwierciedlał główny obszar zainteresowania badawczego, jakim jest wykorzystanie nowoczesnych technologii językowych w celu poprawy bezpieczeństwa aplikacji webowych. W kontekście rosnącej zależności od cyfrowych rozwiązań, temat ten zyskuje na znaczeniu, oferując nowe perspektywy i podejścia do zagadnień bezpieczeństwa. Tytuł można skrócić do **"Zastosowanie dużych modeli językowych w statycznej analizie kodu"**, ponieważ tak nazywa się problem odnajdywania i korekcji błędów w kodzie źródłowym. Korpus badawczy pracy został rozszerzony względem tytułu o projekty open-source aplikacji natywnych i desktopowych oraz wycinki błędnego kodu i poprawnego kodu.

OMÓWIENIE LITERATURY NAUKOWEJ I STOPNIA JEJ PRZYDATNOŚCI

Podstawę teoretyczną pracy stanowi literatura naukowa skupiająca się na dużych modelach językowych oraz ich zastosowaniu w cyberbezpieczeństwie. Szczególną uwagę poświęcono artykułowi "Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?", który posłużył jako punkt wyjścia dla badań. Praca ta ma na celu kontynuację i poszerzenie zakresu tych badań, wykorzystując literaturę naukową jako fundament do eksploracji nowych możliwości w zakresie analizy i naprawy błędów w kodzie. Różnica między pracą a literaturą naukową polega na tym, że praca skupia się na praktycznym zastosowaniu modeli językowych w celu wykrywania i naprawiania błędów bezpieczeństwa w kodzie, podczas gdy literatura naukowa skupia się na badaniu możliwości Sztucznej Inteligencji w tym zakresie.

CEL PRACY

Głównym celem pracy jest zbadanie skuteczności wykorzystania dużych modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa i podatności w kodzie źródłowym aplikacji webowych. W tym kontekście można wyróżnić następujące cele pośrednie:

— Opracowanie praktycznego rozwiązania do statycznej analizy kodu dla aplikacji webowych oraz lokalnych.
— Badanie skuteczności dużych modeli językowych w wykrywaniu podatności i luk bezpieczeństwa.

ZAKRES PRACY

Zakres pracy obejmuje:

— Analizę istniejącej literatury i badań, w szczególności artykułu 'Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?'.
— Projektowanie i implementacja narzędzia do statycznej analizy kodu opartego na modelach OpenAI.
— Przygotowanie zbioru danych z kodem zawierającym potencjalne podatności.
— Testowanie i porównanie skuteczności z innymi rozwiązaniami, np. oferowanymi przez firmę Snyk.
— Analiza wyników i formułowanie wniosków.

1. ANALIZA ISTNIEJĄCEJ LITERATURY ORAZ DOTYCHCZASOWYCH BADAŃ

1.1. CAN OPENAI CODEX AND OTHER LARGE LANGUAGE MODELS HELP US FIX SECURITY BUGS? - *HAMMOND PEARCE, BENJAMIN TAN, BALEEGH AHMAD, RAMESH KARRI, BRENDAN DOLAN-GAVITT*

1.1.1. Metodyka

W badaniu "Czy OpenAI Codex i inne duże modele językowe mogą pomóc nam naprawić błędy bezpieczeństwa?"[**?**] `https://arxiv.org/pdf/2112.02125v1.pdf` napisanym przez Hammond'a Pearce'a, Benjamina Tana, Baleegh'a Ahmad, Ramesh'a Karri oraz Brendan'a Dolan-Gavitt, autorzy skupili się na wykorzystaniu dużych modeli językowych (LLM) do naprawy podatności w kodzie w sposób zero-shot. Badanie koncentrowało się na projektowaniu monitów skłaniających LLM do generowania poprawionych wersji niebezpiecznego kodu. Przeprowadzono eksperymenty na szeroką skalę, obejmujące różne komercyjne modele LLM oraz lokalnie wytrenowany model.

1.1.2. Wyniki

Wyniki wykazały, że LLM mogą skutecznie naprawić 100% syntetycznie wygenerowanych scenariuszy oraz 58% podatności w historycznych błędach rzeczywistych projektów open-source. Odkryto, że różne sposoby formułowania informacji kluczowych w monitach wpływają na wyniki generowane przez modele. Zauważono, że wyższe temperatury generowania kodu przynoszą lepsze wyniki dla niektórych typów podatności, ale gorsze dla innych.

Tak dobrych wyników niestety nie należy interpretować dosłownie, ponieważ z racji, że badanie przeprowadzono na reprezentatywnej próbie, autorzy nie byli w stanie ręcznie sprawdzać poprawności każdej naprawy i wykorzystali w tym celu istniejące narzędzia statycznej analizy kodu, takie jak CodeQL. W związku z powyższym, aby ocenić rzeczywistą skuteczność LLM w naprawianiu podatności, potrzebne są dalsze badania.

1.2. EXAMINING ZERO-SHOT VULNERABILITY REPAIR WITH LARGE LANGUAGE MODELS - *HAMMOND PEARCE, BENJAMIN TAN, BALEEGH AHMAD, RAMESH KARRI, BRENDAN DOLAN-GAVITT*

W pracy naukowej pt. "Examining Zero-Shot Vulnerability Repair with Large Language Models"[**?**] `https://arxiv.org/pdf/2112.02125.pdf`, autorzy przedłużają swoje

badania nad potencjałem wykorzystania Large Language Models (LLM) w kontekście naprawy podatności w kodzie źródłowym. Niniejsze badanie koncentruje się na wyzwaniach związanych z generowaniem funkcjonalnie adekwatnego kodu w realistycznych warunkach aplikacyjnych. Rozszerzając zakres swoich wcześniejszych prac, autorzy skupiają się na bardziej skomplikowanych przypadkach użycia LLM, eksplorując ich zdolność do efektywnego i efektywnego adresowania złożonych problemów związanych z bezpieczeństwem oprogramowania.

Podstawowe pytania badawcze były następujące:

1. Czy LLM mogą generować bezpieczny i funkcjonalny kod do naprawy podatności?
2. Czy zmiana kontekstu w komentarzach wpływa na zdolność LLM do sugerowania poprawek?
3. Jakie są wyzwania przy używaniu LLM do naprawy podatności w rzeczywistym świecie?
4. Jak niezawodne są LLM w generowaniu napraw?

Eksperymenty potwierdziły, że choć LLM wykazują potencjał, ich zdolność do generowania funkcjonalnych napraw w rzeczywistych warunkach jest ograniczona. Wyzwania związane z inżynierią promptów i ograniczenia modeli wskazują na potrzebę dalszych badań i rozwoju w tej dziedzinie.

1.3. RÓŻNICE MIĘDZY OBECNĄ PRACĄ A ISTNIEJĄCĄ LITERATURĄ

W przeciwieństwie do dotychczasowych badań skoncentrowanych głównie na teoretycznym potencjale dużych modeli językowych (LLM) w kontekście zero-shot, niniejsza praca dyplomowa podejmuje kroki w kierunku praktycznego zastosowania tych technologii. Główną różnicą jest tutaj zastosowanie metod takich jak Retrieval Augmented Generation (RAG) oraz in-context learning, co przesuwa nasze podejście w stronę kontekstu few-shot.

— **Zastosowanie Metod RAG i In-context Learning:** W odróżnieniu od tradycyjnych podejść zero-shot, które polegają na generowaniu odpowiedzi bez uprzedniego dostosowania modelu do specyficznego zadania, moja praca wykorzystuje RAG i uczenie się w kontekście, aby lepiej dostosować modele do konkretnych scenariuszy związanych z bezpieczeństwem kodu. Te metody pozwalają na bardziej precyzyjną analizę i naprawę błędów w kodzie.

— **Praktyczne Zastosowanie Modeli Językowych:** Podczas gdy większość istniejących badań skupia się na badaniu możliwości SI w teorii, ta praca koncentruje się na praktycznym zastosowaniu modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa w kodzie. Przez to podejście, praca ta dostarcza bezpośrednich, aplikatywnych rozwiązań, które mogą być wykorzystane w rzeczywistych środowiskach programistycznych.

Takie podejście pozwala nie tylko na zrozumienie teoretycznego potencjału LLM, ale także na ocenę ich praktycznej przydatności w realnych scenariuszach związanych z cyberbezpieczeństwem. Znacząco poszerza to zakres badań w dziedzinie wykorzystania sztucznej inteligencji do poprawy bezpieczeństwa aplikacji, dostarczając nowych perspektyw i rozwiązań.

2. METODYKA ROZWIĄZANIA

W niniejszej pracy dyplomowej zastosowano szereg metod i środków, aby zaimplementować narzędzie do statycznej analizy kodu oraz zbadać i ocenić potencjał dużych modeli językowych w kontekście wykrywania i naprawiania błędów bezpieczeństwa w kodzie źródłowym aplikacji.

Metody i Środki			
Metoda	Opis	**Środek**	Opis
Zero-shot learning	Metoda uczenia maszynowego pozwalająca modelom wykonywać zadania bez wcześniejszego treningu, opierając się na zdolności do rozumienia i generalizacji.	**Modele językowe GPT-3.5, GPT-4**	Zaawansowane modele AI OpenAI do generowania tekstu i odpowiadania na zapytania.
Prompt engineering	Projektowanie promptów w celu uzyskania trafnych odpowiedzi od AI.	**OpenAI Assistant API**	API umożliwiające integrację modeli językowych w aplikacjach.
In-context learning	Uczenie się i dostosowywanie modeli AI na podstawie informacji zawartych w kontekście zapytań.	**Zbiory danych z kodem**	Zestawy danych z przykładami kodu zawierającymi błędy, używane do trenowania narzędzi do wykrywania podatności.
Retrieval Augmented Generation	Technika łącząca generowanie treści z wyszukiwaniem informacji, wspomagana przez OpenAI Assistant API.	**Projekty open-source zawierające podatności**	Publiczne projekty zawierające błędy bezpieczeństwa, używane w testowaniu aplikacji oraz ocenie skuteczności LLM.
Analiza porównawcza	Ocena różnych technik lub systemów poprzez porównanie.	**Statyczne testy podatności**	Narzędzia analizy statycznej kodu, np. CodeQL.
Programowanie obiektowe i funkcyjne	Dwa paradygmaty programowania, koncentrujące się odpowiednio na obiektach i funkcjach.	**Rozwiązania komercyjne, np. Snyk**	Narzędzia AI do zarządzania bezpieczeństwem oprogramowania.
		Python 3.12	Najnowsza wersja języka Python z zaawansowanymi funkcjami.
		Biblioteki: openai, asyncio	Biblioteki Pythona dla integracji z OpenAI i programowania asynchronicznego.
		Komputer osobisty	Urządzenie do tworzenia i testowania oprogramowania.

Tabela 2.1: Metody i środki wykorzystane w projekcie i badaniu.

Metody i środki te zostały wybrane, aby zapewnić efektywne i wszechstronne podejście do analizy i naprawy kodu. Generacja wspomagana pobieraniem danych (RAG ang. Retrieval Augmented Generation) oraz uczenie się w kontekście(in-context learning) umożliwiają efektywną analizę i generowanie kodu. Z kolei analiza porównawcza pozwala na ocenę skuteczności różnych modeli i podejść. Wykorzystanie modeli językowych GPT-3.5 i GPT-4, statycznych testów podatności oraz innych narzędzi i zasobów, zapewnia solidną bazę do przeprowadzenia kompleksowych testów i analiz.

3. PROJEKT ORAZ IMPLEMENTACJA ROZWIĄZANIA

3.1. WSTĘP

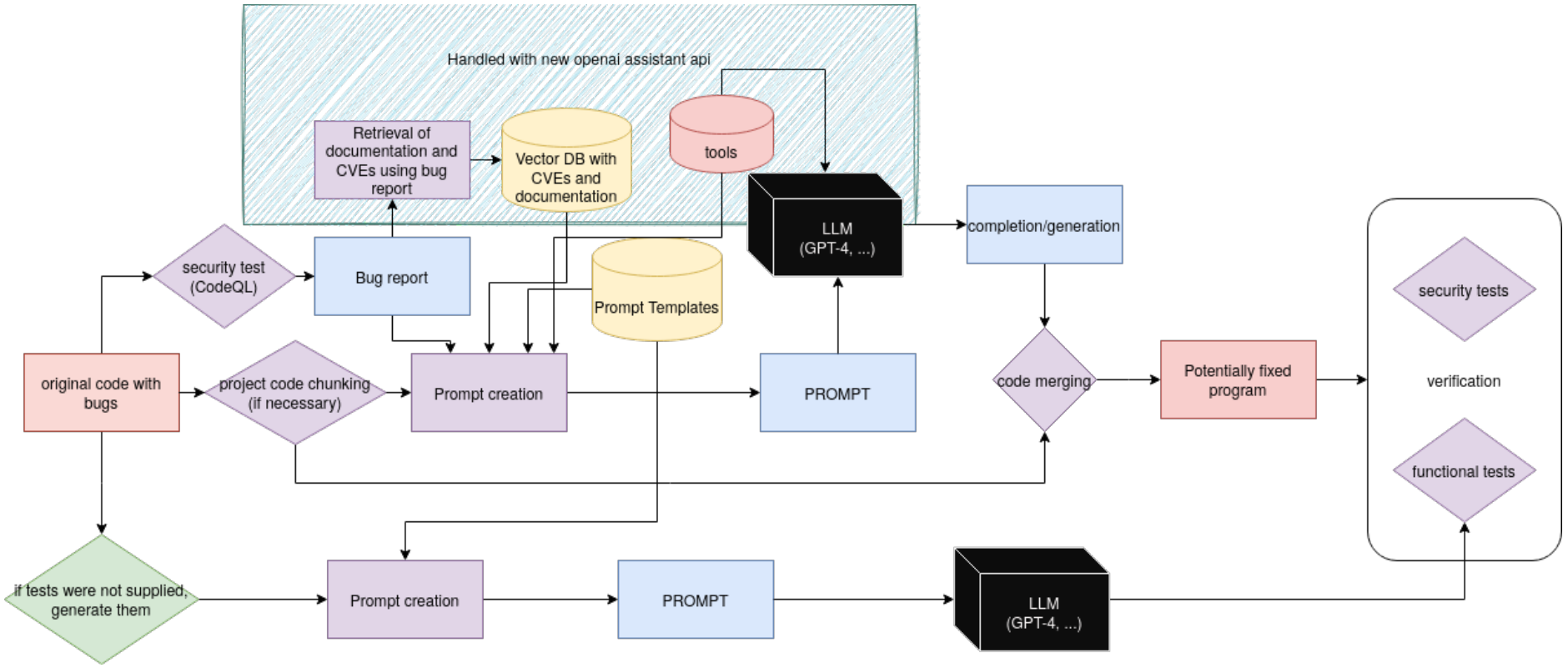
W ramach pracy inżynierskiej opracowano kompleksowe narzędzie **gptester**, które wykorzystuje zaawansowane modele językowe do analizy statycznej kodu. Narzędzie to wykorzystuje domyślnie model GPT-4 do generowania raportów na temat jakości kodu oraz proponowania napraw, ze szczególnym uwzględnieniem bezpieczeństwa kodu.

3.2. ARCHITEKTURA SYSTEMU

3.2.1. Ogólny opis

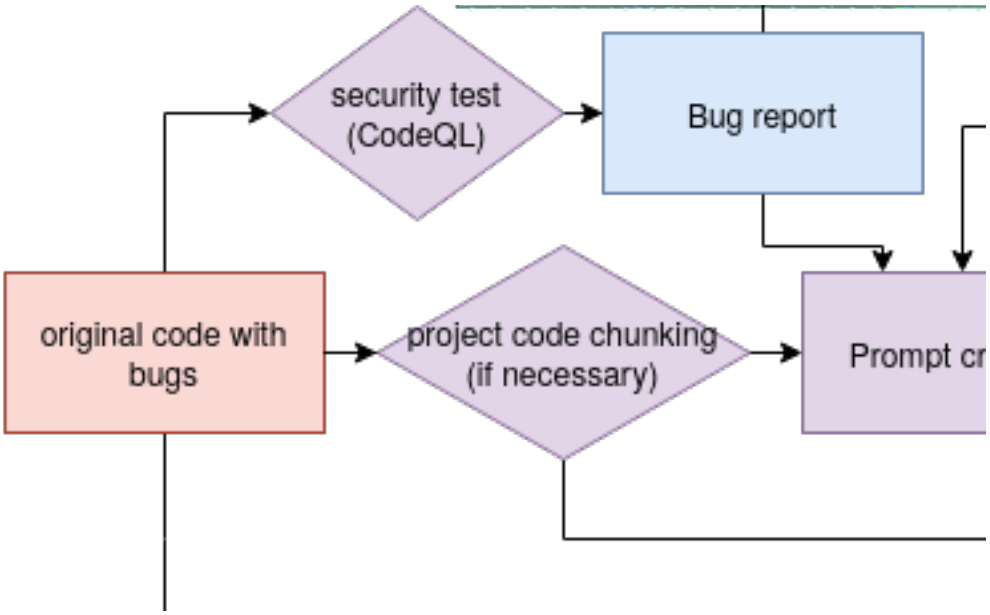
GPTester jest programem napisanym w języku Python, wykorzystującym model GPT-4 (lub GPT-3.5-turbo) dostarczony przez OpenAI. Jest zaprojektowany do uruchamiania z linii poleceń, a wyniki jego pracy są zapisywane w pliku formatu markdown oraz do osobnego katalogu z plikami wynikowymi zawierającymi poprawiony kod. W przyszłości planowane jest wprowadzanie poprawek do bazy kodu za pomocą funkcji git, aby znacznie ułatwić wprowadzanie i analizę proponowanych popraw.

3.2.2. Schemat blokowy



Rys. 3.1: Schemat blokowy działania aplikacji *gptester*

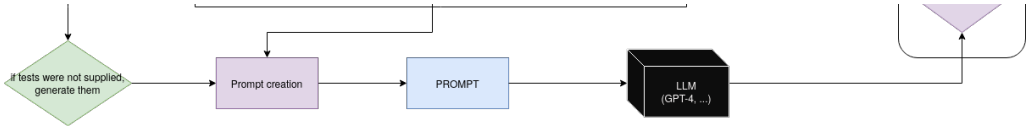
3.2.2.1. Dane wejściowe i wstępne przetwarzanie



Rys. 3.2: dane wejściowe w schemacie blokowym

Proces rozpoczyna się od oryginalnego kodu z błędami. Następnie jeśli użytkownik dostarczył wystarczająco danych dla CodeQL zostanie ono wykorzystane do wstępnej analizy, a dane wynikowe zostaną wykorzystane do wzbogacenia poleceń(promptów). W przeciwnym wypadku polecenie(prompt) zostanie skreowany na podstawie danych posiadanych.

3.2.2.2. Generacja testów funkcjonalnych

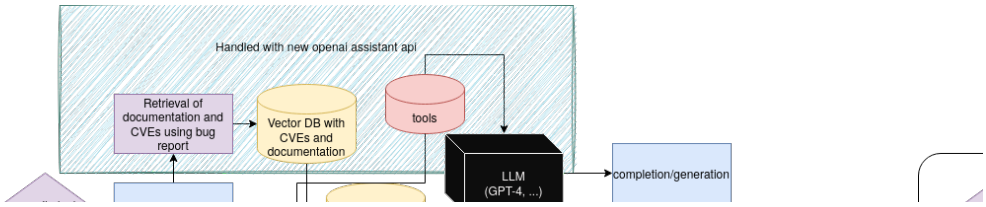


Rys. 3.3: Część schematu opisująca proces generatora testów funkcjonalnych

Jeżeli testy funkcjonalne dla naszego kodu nie zostały zapewnione, zostaną one wygenerowane za pomocą osobnego agenta. Jest to funkcja nadal testowana, która w przyszłości ma za zadanie generować testy funkcjonalne dla kodu, który nie posiada testów funkcjonalnych. Problemem jest ich duża ilość dla średnich i dużych projektów, dlatego zalecane jest dostarczenie własnego modułu testów funkcjonalnych.

3.2.3. Wzbogacanie zapytań(promptów)

Dla uzyskania jak najlepszych wyników oraz zapewnienia najnowszej dostępnej wiedzy na temat podatności wykorzystywane zostaje RAG, aby wzbogacić monit o dodatkowe



Rys. 3.4: Część schematu opisująca proces RAG

informacje. Są to informacje otrzymane z CodeQL, które zostają użyte do semantycznego wyszukania powiązanych wpisów w bazie danych CVE. Jeżeli CodeQL nie został użyty, monit zostaje wzbogacony o informacje z bazy CVE, które zostały wyszukane semantycznie w wektorowej bazie wiedzy. W tej chwili o użyciu dnaych z bazy wiedzy decyduje wybrany model OpenAI, dzięki nowym możliwością API. Nowe możliwości API jak własne narzędzia dla LLM, code interpreter (interpreter kodu) oraz semantic search (semantyczne wyszukiwanie) zostały wprowadzone 06.11.2023r. Sprawia to, że niniejszy własny kod do wzbogacania monitów jest niepotrzebny, ale w przyszłości może zostać użyty do wzbogacania monitów o dodatkowe informacje dla modeli otwartoźródłowych.

```
def get_embedding(text, model="text-embedding-ada-002"): # alternatively use code-embedding-ada
    text = str(text)
    text = text.replace("\n", " ")
    if len(text) != 0:
        return openai.Embedding.create(input=[text], model=model)["data"][0]["embedding"]
    return [0.0] * 1536

```

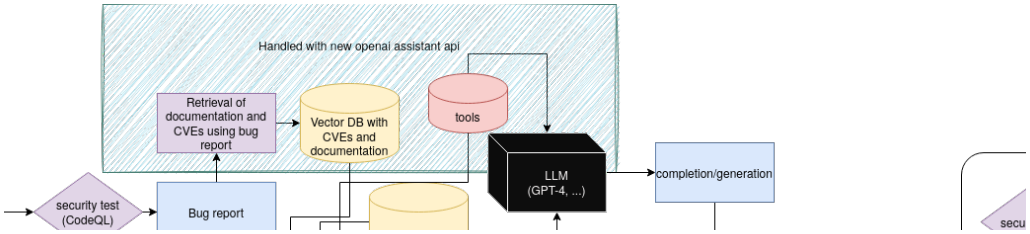
Listing 3.1: Kod tworzący reprezentację wektorową tekstu za pomocą API OpenAI, domyślnie ‘text-embedding-ada-002’, (models.py)

```
    def relevance_for(self, query: str) -> float:
        embedding = get_embedding(query)
        task = get_embedding(self.name)
        score = cosine(task, embedding)
        return score
```

Listing 3.2: Kod porównujący semantyczną odległość (models.py)

Przedstawione zostały rzeczywiste skrawki kodu znajdujące się w projekcie, natomiast w testach oraz podczas działania na modelach komeryjnych OpenAI używane są funkcje dostępne za pomocą API.

3.2.3.1. Tworzenie monitów i interakcja z LLM

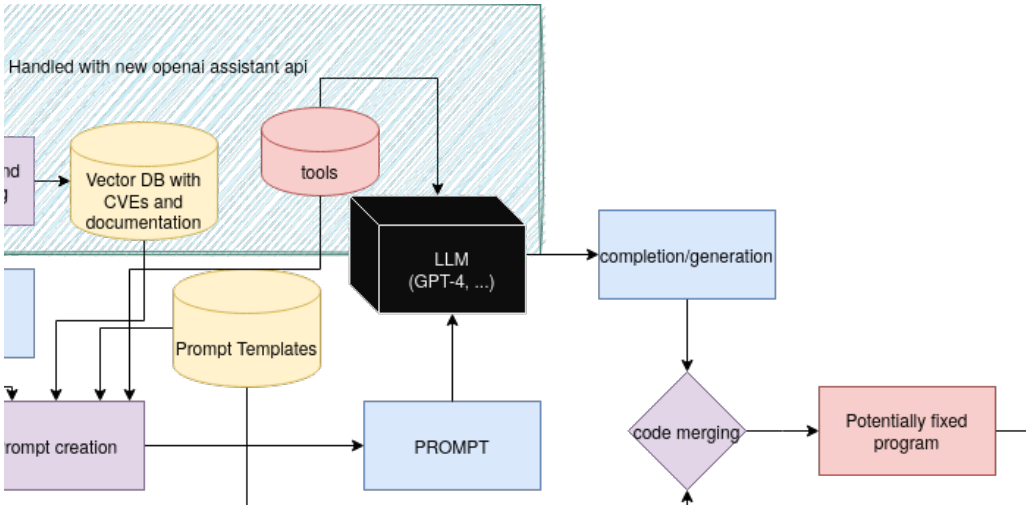


Rys. 3.5: Część schematu opisująca proces RAG

Dla każdej części kodu badanego projektu opdowiednio mieszczącej się w oknie kontekstu dla modeli, tworzony jest PROMPT (monit), który jest następnie przetwarzany przez 'LLM (GPT-4, ...)'. W tym celu wykorzywane są Prompt templates (szablony monitów) oraz semantycznie wyszukane skrawki z Vector DB with CVEs and documentation (wektorowa baza danych z CVE oraz dokumentacją).

Tak spreparowane zapytanie jest następnie zadane modelowi językowemu, który identyfikuje podatności oraz proponuje poprawki.

3.2.3.2. Generowanie kodu i scalanie

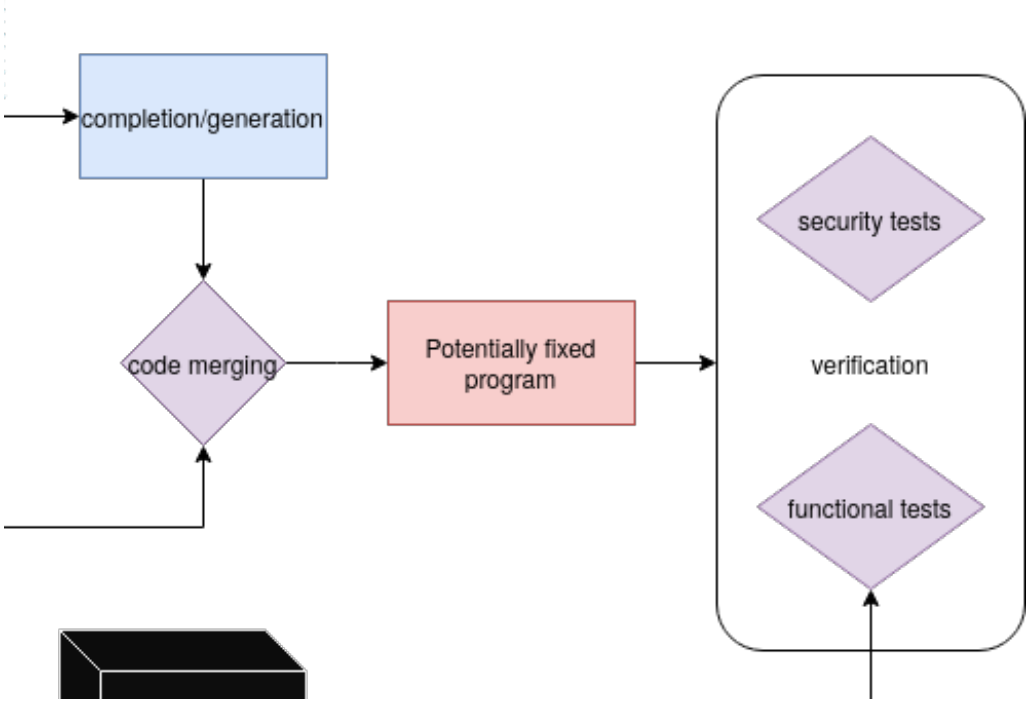


Rys. 3.6: Czarna skrzynka - LLM (Large Language Model)

LLM generuje uzupełnienie/generacje kodu oraz raport, które są następnie łączone (code merging) w potencjalnie naprawiony program. Proces ten wykorzystuje również narzędzia (code interpreter, knowledge retrieval, file writing, git patch, ...), aby ułatwić obróbkę wyników oraz wzbogacić generację.

3.2.3.3. Testy i weryfikacja

W ramach procesu weryfikacyjnego, naprawiony kod jest poddawany testom bezpieczeństwa oraz testom funkcjonalnym, mając na celu zapewnienie, że wprowadzone poprawki



Rys. 3.7: Czarna skrzynka - LLM (Large Language Model)

nie generują nowych defektów oraz że aplikacja funkcjonuje zgodnie z założeniami. W literaturze źródłowej, na której opiera się niniejsza praca inżynierska, do realizacji testów bezpieczeństwa stosowane są narzędzia takie jak CodeQL, czy ASAN/UBSAN, które umożliwiają wykrycie podatności w kodzie. Proces ten jest zautomatyzowany, co jest kluczowe dla przeprowadzenia badań w skali naukowej. Jednakże, z powodu specyfiki integracji CodeQL w ramach realizowanego projektu, jego wykorzystanie do testów nie jest rekomendowane, gdyż nie zapewnia wiarygodności wyników. W związku z tym, testy bezpieczeństwa są przeprowadzane manualnie w celu potwierdzenia prawidłowości działania systemu, podczas gdy testy funkcjonalne realizowane są w sposób automatyczny. Zaleca się wskazanie dedykowanego modułu testowego dla skanowanego projektu.

3.2.3.4. Dokumentacja i raportowanie

Wyniki pracy **gptester** są dokumentowane w raporcie o błędach, po czym raport jest zapisywany w plikach markdown, a poprawione pliki z kodem w katalogu ‘fixed‘ z odpowiednim znacznikiem czasowym. W przyszłości poprawki będą wprowadzane do bazy kodu za pomocą git patch.

3.2.3.5. Podsumowanie

Diagram blokowy przedstawia kompleksowy proces analizy i naprawy kodu, który jest silnie zależny od danych wejściowych (kod źródłowy i testy bezpieczeństwa), zaawansowanych algorytmów przetwarzania (duże modele językowe) oraz dokładności w generowaniu

poprawek kodu i ich weryfikacji. Cały proces jest automatyzowany, a weryfikacja jest możliwa na otrzymanych wynikach.

3.3. IMPLEMENTACJA ORAZ UŻYCIE

3.3.1. Środowisko programistyczne i wymagania

Projekt **gptester** został opracowany w środowisku programistycznym Python, z wykorzystaniem modelu GPT-4 dostarczonego przez OpenAI. Proces konfiguracji środowiska rozpoczyna się od przygotowania odpowiedniego środowiska Pythona i zainstalowania niezbędnych zależności.

Wymagania wstępne:

— Python w wersji >3.7 – Język programowania wykorzystany do napisania ‘gptester‘.
— Dostęp do internetu – Niezbędny do pobrania zależności i interakcji z modelem GPT-4 przez API OpenAI.

Instalacja zależności:

```
pip install -r requirements.txt
```

Plik ‘requirements.txt‘ zawiera wszystkie niezbędne biblioteki Pythona wymagane do działania **gptester**. Instalacja zależności jest prosta i może być wykonana w terminalu lub wirtualnym środowisku Pythona, co jest zalecane w celu uniknięcia konfliktów z istniejącymi pakietami.

3.3.2. Uruchomienie programu

```
        cd gptester
        python main.py -h
```

or

```
        cd gptester
        chmod +x main.py
        ./main.py --help
```

3.3.3. Funkcje programu

Program **gptester** został zaprojektowany jako wszechstronny narzędzie do analizy statycznej kodu, wykorzystując zaawansowane modele językowe do wykrywania i naprawiania błędów bezpieczeństwa w kodzie. Kluczowe funkcje programu są dostępne za pomocą różnorodnych argumentów linii poleceń, umożliwiając szeroką konfigurację i dostosowanie do specyficznych potrzeb analizy.

— **-h, –help**: Wyświetla pomoc programu, zawierającą informacje o dostępnych opcjach i ich krótki opis. Jest to przydatne dla użytkowników, którzy chcą szybko zrozumieć, jak korzystać z programu.
— **-v, –verbose**: Aktywuje tryb szczegółowych informacji. W tym trybie, ‘gptester‘ wyświetla dodatkowe informacje na temat każdego etapu przetwarzania, co jest przydatne do debugowania i analizy szczegółów wykonania.
— **-m MODEL, –model MODEL**: Umożliwia wybór modelu językowego używanego do analizy kodu. Domyślnie ustawiony na "gpt-4-1106-preview", ale użytkownik może wybrać inny model, jeśli jest dostępny i lepiej odpowiada wymaganiom projektu.
— **-o OUTPUT, –output OUTPUT**: Określa ścieżkę i nazwę pliku, do którego będą zapisane wyniki analizy. Domyślnie, raport jest zapisywany w pliku markdown w folderze "raports" z nazwą opartą na nazwie analizowanego folderu i znaczniku czasowym. Użytkownik może dostosować tę lokalizację według własnych preferencji.
— **-t TESTS, –tests TESTS**: Pozwala na podanie ścieżki do testów funkcjonalnych, które mają zostać wykonane na analizowanym projekcie. Ta funkcja jest szczególnie przydatna w środowiskach, gdzie istnieje potrzeba zintegrowanego podejścia do testowania i analizy kodu.
— **-c, –codeql**: Włącza integrację z CodeQL, zaawansowanym narzędziem do analizy kodu. Użytkownik musi mieć zainstalowane CodeQL-CLI, aby skorzystać z tej funkcji. Jest to szczególnie przydatne w wykrywaniu bardziej złożonych problemów w kodzie, które mogą umknąć prostym analizom.
— **–command COMMAND**: Umożliwia określenie polecenia budującego projekt, co jest niezbędne dla prawidłowej integracji z CodeQL w przypadku, gdy projekt nie zawiera pliku cmake lub podobnego w katalogu głównym. Domyślnie ustawione na "make".
— **–language LANGUAGE**: Pozwala na określenie języka programowania projektu do analizy w CodeQL. Domyślnie ustawione na "cpp", ale można dostosować do innych języków wspieranych przez CodeQL, co rozszerza możliwości analizy na różnorodne środowiska programistyczne.

Przykład użycia z pełną konfiguracją:

W powyższym przykładzie, gptester analizuje kod znajdujący się w podanej ścieżce, z włączonym trybem szczegółowych informacji, korzystając z modelu GPT-4, zapisując wyniki do określonego pliku raportu, wykonując testy funkcjonalne, integrując z CodeQL,

```
./main.py /ścieżka/do/projektu --verbose --model "gpt-4-1106-preview"
↪ --output "moj_raport.md" --tests "/ścieżka/do/testów" --codeql
↪ --command "mvn -B -DskipTests -DskipAssembly" --language "java"
```

używając polecenia cmake do budowy projektu w języku C++. Budowa projektu jest wymagana przez CodeQL, dlatego ten argument jest wykorzystywany jedynie przy integracji z CodeQL.

3.4. INTEGRACJA Z CODEQL

Integracja gptester z CodeQL znacznie rozszerza jego funkcjonalność analizy statycznej kodu. CodeQL, opracowany przez Microsoft, a dokładnie GitHub, to zaawansowane narzędzie do semantycznej analizy kodu, które umożliwia wykrywanie złożonych podatności i błędów bezpieczeństwa.

Główne cechy integracji z CodeQL:

— **Zaawansowana Analiza Bezpieczeństwa**: CodeQL przekształca kod źródłowy w zapytywalną formę, co pozwala na przeprowadzenie głębokich analiz w poszukiwaniu subtelnych luk bezpieczeństwa.
— **Wsparcie Dla Wielu Języków**: Obsługa różnych języków programowania przez CodeQL, takich jak C++, Java, Python, co jest wykorzystywane przez ‘gptester‘ do analizy różnorodnych projektów.
— **Konfiguracja Procesu Budowania**: Możliwość dostosowania polecenia budowania projektu za pomocą opcji `-command`, niezbędna w przypadku braku pliku konfiguracyjnego jak cmake w katalogu głównym.
— **Elastyczność Analizy**: Użytkownik może wybrać między szybkimi analizami a bardziej dogłębnymi badaniami, co umożliwia dostosowanie procesu do konkretnych wymagań projektu.
— **Automatyzacja Wykrywania Podatności**: CodeQL automatyzuje proces wykrywania podatności, zwiększając skuteczność i efektywność analizy bezpieczeństwa kodu.

Integracja z CodeQL czyni **gptester** narzędziem nie tylko do wykrywania błędów syntaktycznych i strukturalnych, ale także do efektywnego identyfikowania subtelniejszych problemów bezpieczeństwa, które mogą umknąć podczas standardowych analiz. Do działania CodeQL niezbędne jest zainstalowanie CodeQL CLI, które można pobrać ze strony GitHub.

3.5. MODELE JĘZYKOWE

W ramach projektu **gptester** zaimplementowano zaawansowane modele językowe dostarczone przez OpenAI, w szczególności GPT-4, które odegrały kluczową rolę w procesie analizy statycznej kodu. Modele te wykorzystują techniki uczenia maszynowego i sztucznej inteligencji do generowania odpowiedzi na podstawie dostarczonych danych.

3.5.1. Wykorzystanie Modeli Językowych

Modele językowe w projekcie **gptester** są wykorzystywane do identyfikacji i sugerowania potencjalnych napraw błędów w kodzie źródłowym. Proces ten opiera się na zaawansowanej analizie kontekstu i semantyki kodu, co pozwala na precyzyjne wykrywanie nawet subtelnych podatności.

3.5.2. Dostępne Modele

Projekt integruje różne wersje modeli GPT, z dominującą rolą GPT-4, który charakteryzuje się wyższą zdolnością do zrozumienia złożonych zapytań i generowania bardziej precyzyjnych odpowiedzi. Dostępność innych modeli, takich jak GPT-3.5, zapewnia elastyczność w doborze narzędzia w zależności od specyficznych wymagań analizy.

3.5.2.1. Modele otwartoźródłowe

W przyszłości dostępne będą także modele otwartoźródłowe, za pomocą narzędzia Ollama. Dostępne będą między innymi: Llama2, GPT-J, Mistral, Falcon, czy jakikolwiek model dostępny w repozytoriach Ollama. Narzędzie to pozwala na łatwe pobieranie modeli, konteneryzowane uruchamianie, a także dostęp za pomocą API. Zapewni to jeszcze większą elastyczność i dostosowanie do potrzeb projektu.

3.5.3. Inżynieria Poleceń (Promptów)

Kluczowym aspektem wykorzystania modeli językowych jest inżynieria promptów, czyli proces projektowania i optymalizowania zapytań w celu uzyskania jak najbardziej trafnych odpowiedzi od modelu. W projekt **gptester** zaimplementowano zestaw specjalnie opracowanych promptów, które są dostosowane do identyfikacji różnych rodzajów błędów i podatności w kodzie.

Polecenie systemowe używane dla agenta odpowiedzialnego za identyfikację i naprawę błędów wygląda następująco:

```
You are a top-tier security specialist and developer. You have been
tasked with finding vulnerabilities and security bugs in a program.
You will be given either a code snippet, a whole codefile or multiple
files and optionally a description of errors found by CodeQL. You must
find all the errors, especially the ones that weren't found by CodeQL
```

```
and list them. You will output a potential fix using the git version
control format, stating which lines were deleted and which lines were
added. Then you will write the fixed code to a file with the same name
as the original file in a new folder called "fixed". Always write the
new file without asking for confirmation or more context, even when
you only have a snippet of code write it to a new file.

write_file = {
        "name": "write_file",
        "description": "Writes content to a specified file.",
        "parameters": {
            "type": "object",
            "properties": {
                "filename": {"type": "string"},
                "content": {"type": "string"}
            },
            "required": ["filename", "content"]
        }
    }
```

Można zauważyć, że prompty są złożone z dwóch części: pierwsza część to opis zadania, które ma wykonać model, a druga część to opis funkcji, która ma zostać użyta do zapisania plików. W ten sposób model jest w stanie zrozumieć kontekst zadania oraz wykonać odpowiednie operacje. Użycie funkcji oraz zwrócenie odpowiedzi przez tę funkcję jest zaimplementowane w ai/assistant.py **??**. Lokalizacja zapisywanych plików jest zmieniana przez kod i niezależna od modelu.

Prompt dla innych punktów końcowych API będzie wyglądał inaczej. Niezbędna jest wówczas implementacja parsera dla odpowiedzi od modelu językowego, aby wyodrębnić odpowiednie informacje i zapisać do plików.

3.5.4. Integracja z API OpenAI

Komunikacja z modelami językowymi odbywa się za pośrednictwem API OpenAI, co umożliwia wykorzystanie najnowszych osiągnięć w dziedzinie sztucznej inteligencji bez konieczności posiadania zasobów obliczeniowych do lokalnego trenowania modeli.

Można wyróżnić trzy główne sposoby komunikacji z API OpenAI:

1. **Completion(Komplecja/Uzupełnienie)**: Punkt końcowy API zakończenia otrzymał ostateczną aktualizację w lipcu 2023 r. i ma inny interfejs niż nowy punkt końcowy

zakończenia czatu. Zamiast danych wejściowych będących listą komunikatów, danymi wejściowymi jest dowolny ciąg tekstowy zwany podpowiedzią.
Przykładowe wywołanie starszego interfejsu API Completions wygląda następująco:

```
from openai import OpenAI
client = OpenAI()

response = client.completions.create(
  model="gpt-3.5-turbo-instruct",
  prompt="Write a tagline for an ice cream shop."
)
```

2. **ChatCompletion(Komplecja/Uzupełnienie dialogowe)**: Modele czatu przyjmują listę wiadomości jako dane wejściowe i zwracają wiadomość wygenerowaną przez model jako dane wyjściowe. Chociaż format czatu został zaprojektowany tak, aby ułatwić wieloturowe rozmowy, jest równie przydatny w przypadku zadań jednoturowych bez żadnej rozmowy.
Przykładowe wywołanie interfejsu API Chat Completions wygląda następująco:

```
from openai import OpenAI
client = OpenAI()

response = client.chat.completions.create(
  model="gpt-3.5-turbo",
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Who won the world series in 2020?"},
    {"role": "assistant", "content": "The Los Angeles Dodgers won the World
    ↪  Series in 2020."},
    {"role": "user", "content": "Where was it played?"}
  ]
)
```

3. **Assistant(Asystent)**: Punkt końcowy API Asystentów pozwala na tworzenie asystentów AI w ramach własnych aplikacji. Asystent posiada instrukcje i może wykorzystywać modele, narzędzia oraz wiedzę do odpowiadania na zapytania użytkowników. API Asystentów obecnie obsługuje trzy typy narzędzi: Interpreter Kodu, Pobieranie oraz Wywoływanie Funkcji. W przyszłości planujemy udostępnić więcej narzędzi stworzonych przez OpenAI oraz umożliwić dostarczanie własnych narzędzi na naszej platformie.
Przykładowe wywołanie interfejsu API Asystentów wygląda następująco:

```
assistant = client.beta.assistants.create(
    name="Math Tutor",
    instructions="You are a personal math tutor. Write and run code to
    ↪  answer math questions.",
    tools=[{"type": "code_interpreter"}],
    model="gpt-4-1106-preview"
)
```

3.5.5. Implementacja w projekcie

Kod użyty w projekcie został dostosowany do nowego interfejsu Assistant API, który jest zgodny z najnowszymi wersjami modeli językowych. W projekcie znajduje się także kod wykorzystujący starsze interfejsy API, który może być użyty w przypadku starszych wersji modeli językowych. Ten kod oraz własna implementacja bazy wektorowej wynika z daty wprowadzenia punktu końcowego API asystantów, który został wprowadzony 06.11.2023r. Kod użyty w projekcie znajduje się w pliku ‘ai/assistant.py‘ i wygląda następująco:

```python
class Assistant():
    def __init__(self, role: str, name: str = "Assistant", model: str = 'gpt-3.5-turbo-1106', iol: IOlog = None,
    ↪  tools = None, messages = None) -> None:
        self.name = name
        self.iol = iol
        self.instructions = role
        self.assistant = client.beta.assistants.create(
            name=name,
            instructions=self.instructions,
            model=model,
            tools=tools if tools else [{"type": "code_interpreter"}, {"type": "retrieval"}]
        )
        self.thread = client.beta.threads.create(messages = messages)
    # pominięte metody przekształcania wiadomości, pozwalające na łatwą zmianę punktów końcowych API bez
    ↪  modyfikowania funkcji definiujących agentów. Inne punkty końcowe korzystają z osobnych klas

    async def next(self, messages: list[dict[str, str]]=None, prompt=None, directory: str = 'fixed'):
        if messages:
            self.messages_to_thread(messages)
        if prompt:
            self.fuser(self, prompt)
        try:
            run = client.beta.threads.runs.create(
                thread_id=self.thread.id,
                assistant_id=self.assistant.id,
                model=self.assistant.model if self.assistant.model else "gpt-4-1106-preview",
                instructions=self.instructions
            )
            # Polling mechanism to see if runStatus is completed
            run_status = client.beta.threads.runs.retrieve(thread_id=self.thread.id, run_id=run.id)
            while run_status.status != "completed":
                await asyncio.sleep(2)  # Sleep for 2 seconds before polling again
                run_status = client.beta.threads.runs.retrieve(thread_id=self.thread.id, run_id=run.id)
                tool_outputs = []
                # Check if there is a required action
                if run_status.required_action and run_status.required_action.type == "submit_tool_outputs":
                    for tool_call in run_status.required_action.submit_tool_outputs.tool_calls:
                        name = tool_call.function.name
                        arguments = json.loads(tool_call.function.arguments)
                        if "filename" in arguments and self.name == "debug_agent":
                            filename = os.path.basename(arguments["filename"])
                            timestamp = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
                            arguments["filename"] = os.path.join(directory, f'fixed_{timestamp}', filename)
                        # Check if the function exists in the tools module
                        if hasattr(tools, name):
                            function_to_call = getattr(tools, name)
                            response = await function_to_call(**arguments)
                            # Collect tool outputs
                            tool_outputs.append({"tool_call_id": tool_call.id, "output": response})
                # Submit tool outputs back
                if tool_outputs:
                    client.beta.threads.runs.submit_tool_outputs(
                        thread_id=self.thread.id,
                        run_id=run.id,
                        tool_outputs=tool_outputs
                    )
                if run_status.status == "failed":
                    raise Exception(f"Run failed with reason: {run_status.last_error}")
            # Get the last assistant message from the messages list
            messages = client.beta.threads.messages.list(thread_id=self.thread.id)
            response = [message for message in messages if message.run_id == run.id and message.role ==
            ↪  "assistant"][-1]
            if response:
                self.iol.log(f"{response.content[0].text.value} \n")
        except TypeError:
            self.iol.log(f"TypeError: run[-1][\"content\"]: {run[-1]['content']}")
        return messages
```

Listing 3.3: Kod używany do komunikacji z API OpenAI (ai/assistant.py)

Przy używaniu punktu końcowego ChatCompletion niezbędna jest implementacja parsera, aby możliwe było wyodrębnienie wiadomości oraz zapisywanie do plików. Taki parser znajduje się w pliku *utils/chat_to_files.py*. Przewaga ChatCompletion nad Assistant jest taka, że są to stabilniejsze metody z takimi funkcjami jak przesyłanie strumieniowe, które nie są dostępne w Assistant.

3.6. DEFINICJA AGENTA AI

Agent AI w kontekście projektu **gptester** definiuje się jako zaawansowany system komputerowy, który wykorzystuje techniki sztucznej inteligencji i uczenia maszynowego do automatyzacji zadań związanych w tym przypadku z analizą statyczną kodu źródłowego. Agent ten jest zaprogramowany do samodzielnego podejmowania decyzji na podstawie dostarczonych mu danych, mając na celu identyfikację i naprawianie błędów oraz podatności bezpieczeństwa w kodzie.

3.6.1. Cechy Charakterystyczne

Agent AI charakteryzuje się następującymi cechami:

— **Autonomia**: Możliwość samodzielnego działania bez bezpośredniej interwencji człowieka, opierając się na zasadach i algorytmach AI.
— **Interaktywność**: Umiejętność komunikacji z użytkownikami lub innymi systemami w celu wymiany informacji i wykonania zadań.
— **Narzędzia**: Wykorzystanie różnorodnych narzędzi i funkcji do analizy i generowania rozwiązań.
— **Pamięć długotrwała**: Możliwość zapamiętywania informacji i wykorzystywania ich w przyszłych zadaniach, Zaimplementowana dzięki technice generowania wspomaganego pobieraniem (RAG), w tym celu możliwe są także inne rozwiązania. Nie wykorzystywana w projekcie.

3.6.2. Zastosowanie w gptester

W projekcie **gptester**, agent AI odgrywa kluczową rolę w:

— **Wykrywaniu błędów**: Automatyczne identyfikowanie błędów w kodzie źródłowym.
— **Generowaniu napraw**: Proponowanie rozwiązań naprawczych dla wykrytych problemów.
— **Testowaniu**: Automatyzowanie procesu testowania kodu, w tym pisania testów i ich wykonywania.

Dzięki swojej zaawansowanej konfiguracji i integracji z modelami językowymi GPT-4, agenci AI w **gptester** pozwalają na nowoczesne podejście do analizy statycznej kodu, za-

pewniając łatwość w implementacji oraz wysoką skuteczność i efektywność w wykrywaniu oraz naprawianiu błędów programistycznych.

3.7. KONFIGURACJA AGENTÓW AI

Projekt **gptester** wykorzystuje zaawansowanych agentów AI, które są kluczowymi elementami w procesie analizy statycznej kodu. Konfiguracja tych agentów obejmuje szereg parametrów i narzędzi, które są niezbędne do ich efektywnego działania.

3.7.1. Parametry Konfiguracyjne

Każdy agent AI jest inicjalizowany z określonymi parametrami konfiguracyjnymi, które definiują jego zachowanie i funkcjonalność:

— **Rola**: Określa podstawowy zakres działania agenta, np. debugowanie lub testowanie. Jest to odpowiednik polecenia systemowego (system prompt) w interakcji z modelami za pomocą punktu końcowego 'ChatCompletion' w OpenAI API.
— **Nazwa**: Unikalna nazwa agenta, używana do identyfikacji i logowania.
— **Model Językowy**: Wskazuje na model AI używany przez agenta, domyślnie ustawiony na GPT-4 w najnowszej wersji.
— **Narzędzia**: Zestaw narzędzi, które agent może wykorzystywać podczas analizy.

Niestety, w punkcie końcowym API Asystentów nie ma możliwości przekazania parametrów konfigurujących komplecje i generacje, dlatego nie można przekazać parametrów takich jak 'temperature' czy 'max_tokens'. W przyszłości, jeśli API Asystentów zostanie rozwinięte, będzie można przekazać te parametry.

3.7.2. Narzędzia i Funkcje

Agent AI korzysta z różnorodnych narzędzi i funkcji, które wspierają jego działanie w różnych scenariuszach:

— **Code Interpreter**: Narzędzie do interpretacji, analizy oraz wykonywania kodu źródłowego udostępniony przez OpenAI.
— **Retrieval**: Mechanizm wyszukiwania i odzyskiwania informacji z zewnętrznych źródeł udostępniony przez OpenAI.
— **Funkcje Specyficzne**: Takie jak 'write_file' i 'run_tests', które umożliwiają zapisywanie treści do plików i wykonanie testów.
— **Integracja z CodeQL**: W przypadku agenta "debug_agent", integracja z CodeQL pozwala na głębszą analizę bezpieczeństwa kodu.

Każdy agent AI w **gptester** jest zaprojektowany w taki sposób, aby był elastyczny i mógł być łatwo dostosowany do zmieniających się wymagań projektowych, co umożliwia szerokie zastosowanie w różnych scenariuszach analizy kodu.

3.8. ROZWÓJ I PLANY NA PRZYSZŁOŚĆ

Sekcja ta skupia się na omówieniu obecnego stanu projektu ‘gptester‘ oraz planowanych rozszerzeń i ulepszeń, które mają zostać wprowadzone w przyszłości. Planowane działania są zgodne z informacjami zawartymi w sekcji "In development" pliku README.md.

3.8.1. Obecne osiągnięcia

Projekt ‘gptester‘ osiągnął już kilka kluczowych kamieni milowych w swoim rozwoju:

— **Podstawowa funkcjonalność**: Program już teraz oferuje podstawowe funkcje analizy statycznej kodu, umożliwiając identyfikację typowych błędów i podatności.
— **Wykorzystanie technik RAG oraz in-context learning**: ‘gptester‘ wykorzystuje zaawansowane techniki generacji wspomaganej odzyskiwaniem danych oraz uczenia się w kontekście, co pozwala na lepsze dostosowanie modeli językowych do specyficznych zadań.
— **Integracja z CodeQL**: Znaczącym osiągnięciem jest wdrożenie integracji z CodeQL, co znacznie rozszerza możliwości analizy kodu, szczególnie w zakresie wykrywania złożonych błędów bezpieczeństwa. Niestety funkcja jest nadal testowana i stabilizowana.

Te osiągnięcia stanowią solidną podstawę dla dalszego rozwoju i rozbudowy ‘gptester‘, kładąc nacisk na wydajność, dokładność i wszechstronność narzędzia.

3.8.2. Planowane rozszerzenia

W ramach dalszego rozwoju, projekt ‘gptester‘ ma w planach kilka istotnych rozszerzeń i ulepszeń:

— **Aktualizacja kodu za pomocą funkcji git i plików patch**: Rozwój funkcjonalności, która pozwoli na automatyczne wprowadzanie poprawek do kodu źródłowego na podstawie wygenerowanych plików git i patch. To ulepszenie ułatwi proces naprawy kodu, umożliwiając automatyczne aplikowanie poprawek oraz interaktywne wybieranie elementów z obu wersji.
— **Dodanie więcej testów oraz automatyzacja testów bezpieczeństwa**: Rozbudowa zestawu testów funkcjonalnych, jednostkowych i bezpieczeństwa, co pozwoli na lepsze sprawdzanie niezawodności i efektywności ‘gptester‘. Automatyzacja testów pozwoli na skuteczniejsze wykrywanie błędów oraz ułatwi badania.
— **Obsługa więcej języków programowania dla CodeQL**: Rozszerzenie integracji z CodeQL o więcej języków programowania, co zwiększy użyteczność ‘gptester‘ w

różnorodnych projektach programistycznych. Planowane jest dodanie wsparcia dla popularnych języków takich jak JavaScript, Python czy Ruby.

Te planowane rozszerzenia mają na celu nie tylko ulepszenie obecnych funkcjonalności gptester, ale również wprowadzenie nowych możliwości, które uczynią narzędzie jeszcze bardziej wszechstronnym i przydatnym w różnych kontekstach analizy kodu.

3.9. PODSUMOWANIE

W niniejszym rozdziale przedstawiono szczegółowy opis projektu ‘gptester‘, jego obecne możliwości oraz plany rozwojowe. ‘gptester‘, jako zaawansowane narzędzie do analizy statycznej kodu, wykorzystujące model GPT-4 od OpenAI, stanowi znaczący krok naprzód w dziedzinie automatyzacji i poprawy jakości kodu źródłowego.

Podstawowe osiągnięcia:

— Rozwój podstawowych funkcjonalności analizy statycznej, umożliwiających efektywne wykrywanie i naprawianie błędów w kodzie.
— Integracja z CodeQL, dzięki której ‘gptester‘ zyskuje zdolność do przeprowadzania bardziej zaawansowanych analiz bezpieczeństwa.
— Elastyczność w obsłudze różnorodnych scenariuszy użytkowania poprzez konfigurowalne argumenty linii poleceń.

Plany rozwojowe:

— Rozbudowa funkcjonalności aktualizacji kodu źródłowego za pomocą plików git i patch, co uprości proces wprowadzania poprawek.
— Dodanie wsparcia dla dodatkowych języków programowania w integracji z CodeQL, co rozszerzy zakres zastosowania ‘gptester‘.
— Automatyzacja testów.

Podsumowując, ‘gptester‘ już teraz stanowi potężne narzędzie do analizy i poprawy kodu źródłowego, a planowane rozbudowy i ulepszenia sprawią, że będzie ono jeszcze bardziej wszechstronne i skuteczne. Projekt ten pokazuje, jak technologie AI i narzędzia do automatycznej analizy kodu mogą przyczynić się do poprawy jakości oprogramowania oraz efektywności procesów programistycznych.

4. ZBIORY DANYCH I ICH PRZYGOTOWANIE

W kontekście niniejszego rozdziału dokonano prezentacji oraz analizy zbiorów danych, które zostały wykorzystane w procesie testowania programu **gptester** oraz badania skuteczności dużych modeli językowych (LLM). Szczegółowo opisany został proces przygotowania i przetwarzania tych zbiorów danych, co ma kluczowe znaczenie dla efektywności analizy statycznej kodu i kalibracji narzędzia.

4.1. PRZEGLĄD WYKORZYSTANYCH ZBIORÓW DANYCH

Następująca sekcja zawiera omówienie źródeł danych, ich specyfikacji oraz roli, jaką odgrywają w kontekście projektu. Analiza ta obejmuje zarówno otwarte zbiory danych, repozytoria kodu, jak i bazy danych podatności.

— **snoopysecurity/Vulnerable-Code-Snippets**: Repozytorium w serwisie Github zawierające zbiór fragmentów kodu zawierających luki bezpieczeństwa. Fragmenty pobrane z różnych wpisów na blogach, książek, zasobów itp. Zbiór w głównej mierze używany do testowania implementacji. Niektóre fragmenty kodu zawierają wskazówki w nazwach/komentarzach. Ewentualne naruszenie praw autorskich niezamierzone.
`https://github.com/snoopysecurity/Vulnerable-Code-Snippets`

— **OWASP VulnerableApp**: Aplikacja webowa zawierająca wiele podatności, używana do testowania narzędzi do testowania bezpieczeństwa aplikacji webowych.
`https://github.com/SasanLabs/VulnerableApp`

4.2. PROCES PRZYGOTOWANIA DANYCH

Dobrane przeze mnie zbiory danych zostały tak, by nie trzeba było dostosowywać programu do konkretnego formatu. Oznacza to, że wskazane repozytoria zawierają przykłady kodu zapisane w plikach.

4.2.1. snoopysecurity/Vulnerable-Code-Snippets

Repozytorium zawiera wiele plików z przykładami kodu, które mogą zawierać błędy bezpieczeństwa. Pliki te zostały pobrane z różnych źródeł, takich jak blogi, książki, zasoby itp. Pliki te zawierają często komentarze lub nazwy zmiennych, które wskazują na potencjalne błędy bezpieczeństwa. Pozwala to nam na izolację problemu identyfikacji podatności

od generowania kodu. W pierwszej kolejności badania zostały przeprowadzone bez wprowadzania zmian w kodzie, aby ocenić skuteczność modeli językowych w korekcji błędów bezpieczeństwa. W kolejnym kroku, w celu przebadania zdolności do identyfikowania błędów, zostały wprowadzone zmiany w kodzie, takie jak usunięcie komentarzy, zmienienie nazw zmiennych, itp. W ten sposób można było sprawdzić, czy modele językowe są w stanie wykryć błędy bezpieczeństwa, gdy mają więcej informacji na temat kodu.

4.2.2. OWASP VulnerableApp

Aplikacja webowa zawierająca wiele podatności, używana w testach narzędzi do testowania bezpieczeństwa aplikacji webowych. Zawiera wiele przykładów kodu, które mogą zawierać błędy bezpieczeństwa. Dzięki użyciu w badaniu przykładu z prawdziwej aplikacji, można było sprawdzić, czy modele językowe są w stanie wykryć błędy bezpieczeństwa w prawdziwym kodzie.

4.3. WYZWANIA I OGRANICZENIA

Głównym wyzwaniem prezentowanym przez użyte przeze mnie próbki badawcze wynikają z ich charakteru. Zbiór danych Vulnerable-Code-Snippets nie jest reprezentatywne dla rzeczywistych aplikacji, a jedynie zawiera przykłady kodu, które mogą zawierać błędy bezpieczeństwa. W przypadku większości przykładów kodu, nie jest możliwe uruchomienie go bez posiadania kodu całego projektu, co utrudnia ewaluację. W związku z powyższym niektóre skrawki kodu zostały obudowane w aplikacje webową, natomiast inne pominięte. Nie każdy skrawek kodu w repozytorium jest wycięty z aplikacji webowej, te przykłady zostały uwzględnione w badaniach i sprawiały najmniej problemów.

W przypadku OWASP VulnerableApp, aplikacja jest w pełni uruchamialna i możliwe jest badanie funkcjonalności programu na przykładzie rzeczywistym. Trudnością jest natomiast wczesna wersja 'gptester', w której nie została jeszcze wprowadzona funkcjonalność aktualizowania bazy kodu za pomocą funkcji systemów kontroli wersji. Powoduje to konieczność ręcznego scalania zmian w kodzie z nowymi wersjami aplikacji, co jest czasochłonne i utrudnia badania. Testy bezpieczeństwa zostaną przeprowadzone za pomocą skanerów podatności, przede wszystkim OWASP ZAP, który jest zoptymalizowany do wykrywania podatności w aplikacji OWASP VulnerableApp, co pomoże w faktycznej ocenie skuteczności wprowadzonych korekt.

4.4. PODSUMOWANIE

Podsumowujemy, jak przygotowanie i analiza zbiorów danych wpłynęła na projekt ‘gptester‘ i jakie wnioski można z tego wyciągnąć.

5. BADANIA EKSPERYMENTALNE

Niniejszy rozdział jest poświęcony prezentacji wyników badań eksperymentalnych przeprowadzonych w ramach projektu. Wyniki te przedstawione w sposób klarowny, z wykorzystaniem wykresów i tabel dla lepszej interpretacji. Wnioski wynikające z badań są bezpośrednio związane z założonymi celami projektu i opierają się na analizie uzyskanych danych.

Zaprezentowana metodyka badań obejmuje szczegółowy opis zastosowanych procedur testowych, co pozwala na ocenę wiarygodności i trafności uzyskanych wyników.

5.1. METODYKA BADAŃ

W tej sekcji szczegółowo omawiam metody oraz podejście zastosowane podczas eksperymentów. Zostaną przedstawione narzędzia, parametry konfiguracyjne oraz procedura testowa, które razem tworzą ramy metodyczne naszego badania.

5.1.1. Procedura testowa

Zaprojektowana procedura testowa miała na celu dokładną weryfikację funkcjonalności programu oraz ocenę jego skuteczności w wykrywaniu i naprawie podatności. Kryteria testowe zostały dobrane w sposób umożliwiający kompleksową analizę:

- **Kryterium 1**: Dokładność identyfikacji podatności.
- **Kryterium 2**: Skuteczność proponowanych napraw.
  - **Kryterium 2.1**: Funkcjonalność naprawy.
  - **Kryterium 2.2**: Bezpieczeństwo naprawy.
  - **Kryterium 2.3**: Efektywność naprawy.
- **Kryterium 3**: Efektywność czasowa analizy[1]

Procedura testowa przebiegała według następujących etapów:

1. Selekcja i przygotowanie danych testowych.
2. Analiza statyczna kodu z wykorzystaniem narzędzi AI.
3. Przygotowanie środowiska testowego.
   a) Instalacja zależności.
   b) Konfiguracja środowiska.

[1] *W tej wersji pracy pominięto.*

c) Dopisanie kodu pozwalającego na wykonanie, w razie konieczności.
d) Uruchomienie aplikacji.

4. Przeprowadzenie testów funkcjonalnych oraz testów bezpieczeństwa poprzez przeprowadzenia eksploatacji.
5. Analiza i interpretacja wyników.

5.2. WYNIKI DZIAŁANIA PROGRAMU - PRZYKŁAD

Wyniki działania programu, prezentowane na konsoli oraz dokumentowane w plikach raportów, zapewniają bezpośredni wgląd w proces analizy kodu:

GPTESTER RAPORT

2024-01-25 21:00:51: Beginning scan for Vulnerable-Code-Snippets/Out of Bounds/...
2024-01-25 21:00:51: Found 1 files to scan 2024-01-25 21:00:51: Tokens inside the directory: 701
2024-01-25 21:00:51: Beginning code analysis...
2024-01-25 21:00:51: Using model: gpt-4-1106-preview
2024-01-25 21:00:51: Splitting the content into 1 chunks
2024-01-25 21:03:52: Analyzing the provided code snippet from `damnvuln.c`, I have identified several vulnerabilities, which I have listed below along with a brief description of each:

1. **Integer Overflow**: `int size1 = img.width + img.height; //Vulnerability: integer overflow` If `img.width` and `img.height` are large enough, the sum could overflow, resulting in incorrect behavior or potentially exploitable conditions.
2. **No Check for** `malloc` **Success**: `char* buff1=(char*)malloc(size1);` There is no check to see if `malloc` has successfully allocated memory, which could lead to a null pointer dereference if the allocation fails.
3. **Double Free**: `if(size1/2==0){ free(buff1); //Vulnerability: double free }` If `size1` is even, `buff1` is freed twice, which is undefined behavior and a critical security vulnerability.
4. **Use After Free**: `if(size1 == 123456){ buff1[0]='a'; //Vulnerability: use after free }` Modifying memory after it has been freed could lead to arbitrary code execution or program crashes.
5. **Integer Underflow**: `int size2 = img.width - img.height+100; //Vulnerability: integer underflow` If `img.height` is greater than `img.width + 100`, `size2` could become negative, causing unexpected behavior in `malloc`.

6. **Out of Bounds Read (Stack)**: `char OOBR_stack = buff3[size3+100]; //Vulnerability: out of bound read (stack)` Reading values outside the bounds of an array could lead to information disclosure or crashes.
7. **Out of Bounds Write (Stack)**: `buff3[size3+100]='c'; //Vulnerability: out of bound write (Stack)` Writing beyond the bounds of the stack buffer can overwrite important data and lead to control-flow hijacking.
8. **Out of Bounds Write (Heap)**: `buff4[100]='c'; //Vulnerability: out of bound write (Heap)` Similar to the stack buffer overflow but on the heap, leading to data corruption or arbitrary code execution vulnerabilities.
9. **Memory Leak**: `if(size3>10){ buff4=0; //memory leak? }` Assigning `0` to `buff4` without freeing it first, can lead to a memory leak if `size3` is greater than 10.
10. **No Check for Division by Zero or Negative Results**: `int size3 = img.width / img.height;` Division by zero can occur if `img.height` is zero, and if `img.height` is greater than `img.width`, the result is a negative integer for `size3`.
11. **Possible Buffer Overflow** (No CodeQL error mentioned, but it's an issue): `memcpy(buff1, img.data, sizeof(img.data));` There is no check that the size of the allocated buffer (`buff1`, `buff2`, `buff4`) is at least as large as the amount of data being copied, potentially leading to a buffer overflow.

Now, I will suggest potential fixes and provide the corrected code using the git version control format.

2024-01-25 21:03:52: Tests completed! 2024-01-25 21:04:53: Scan complete!

5.2.1. Opis przedstawionego wyniku

Przedstawiony raport zawiera informacje o wykrytych podatnościach, wraz z ich opisem oraz sugestiami napraw. W tym przebiegu aplikacji agent dokonał wyboru zapisania sugestii napraw do pliku o rozszerzeniu diff. Dodatkowo zapisał poprawioną wersję badanego pliku, która jest możliwa do skompilowania. Wszystkie te informacje i pliki zostały wygenerowane przez program, na podstawie analizy kodu źródłowego. Formatowanie markdown zostało zinterpretowane przed umieszczeniem w niniejszej pracy inżynierskiej.

Lokalizacja napraw reprezentowana w formacie diff może zostać przez agenta zapisana zarówno w raporcie jak i w pliku. W tej chwili decyduje o tym model językowy, ale podczas dalszych prac nad projektem, planowane jest dodanie mechanizmów, które pozwolą na wybór preferowanego sposobu prezentacji napraw, co pozwoli na łatwe aplikowanie poprawa za pomocą systemów kontroli wersji.

5.3. BADANIA NA ZBIORZE *SNOOPYSECURITY/VULNERABLE-CODE-SNIPPETS*

Analiza zbioru *snoopysecurity/Vulnerable-Code-Snippets* dostarczyła istotnych informacji na temat specyfiki podatności i skuteczności ich wykrywania przez system. Zbiór ten, zawierający 184 pliki źródłowe o łącznej liczbie 41831 tokenów, stanowił reprezentatywną próbkę dla naszych eksperymentów.

Eksperymenty przeprowadzono z wykorzystaniem poniższych parametrów:

```
> ./main.py -m 'gpt-4-1106-preview' Vulnerable-Code-Snippets/
```

```
  ___  ___  _____           _
 / __|| _ \|_   _| ___  ___| |_  ___  _ _
| (_ ||  _/  | |  / -_)(_-/|  _|/ -_)| '_|
 \___||_|    |_|  \___|/__/ \__|\___||_|


          The static code analysis agent, version: assistant-0.3

2024-01-25 21:10:51: Beginning scan for Vulnerable-Code-Snippets/
2024-01-25 21:10:51: Found 131 files to scan
2024-01-25 21:10:52: Tokens inside the directory: 30712
2024-01-25 21:10:52: Using model: gpt-4-1106-preview
2024-01-25 21:10:52: Beginning code analysis...
```

Program pokazał nam, że w katalogu zawierającym skrawki podatnego kodu znajduje się 131 plików, a łączna liczba tokenów w tych plikach wynosi 30712.

5.4. STUDIUM PRZYPADKU: ANALIZA KODU PODATNEGO NA BŁĘDY TYPU "OUT OF BOUNDS" W ZBIORZE SNOOPYSECURITY/VULNERABLE-CODE-SNIPPETS

Niniejszy podrozdział przedstawia szczegółowe studium przypadku, w którym dokonano analizy specyficznego fragmentu kodu, sklasyfikowanego jako zawierający błędy typu "Out of Bounds". Analiza ta została przeprowadzona na przykładzie wybranym ze zbioru *snoopysecurity/Vulnerable-Code-Snippets*. Omawiany plik źródłowy zawierał kod, który został opatrzony komentarzami zaznaczającymi potencjalne miejsca podatności. Te adnotacje umożliwiają dokonanie porównawczej oceny zachowania się programu w kontekście występowania bądź braku zidentyfikowanych wskazówek dotyczących podatności.

5.4.1. Dane wejściowe

W katalogu znajdował się jeden plik o nazwie *damnvuln.c*, zawierający następujący kod źródłowy:

```c
//https://github.com/hardik05/Damn_Vulnerable_C_Program/blob/master/imgRead.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct Image
{
        char header[4];
        int width;
        int height;
        char data[10];
};

int ProcessImage(char* filename){
        FILE *fp;
        char ch;
        struct Image img;

        fp = fopen(filename,"r");
        if(fp == NULL){
                printf("\nCan't open file or file doesn't exist.");
                exit(0);
        }
        printf("\n\tHeader\twidth\theight\tdata\t\r\n");
        while(fread(&img,sizeof(img),1,fp)>0){
                printf("\n\t%s\t%d\t%d\t%s\r\n",img.header,img.width,img.height,img.data);
                int size1 = img.width + img.height; //Vulnerability: integer overflow
                char* buff1=(char*)malloc(size1);
                memcpy(buff1,img.data,sizeof(img.data)); //Vulnerability: no data buffer size/malloc success
    ↪  check?
                free(buff1);
                if (size1/2==0){
                        free(buff1); //Vulnerability: double free
                }
                else{
                        if(size1 == 123456){
                                buff1[0]='a'; //Vulnerability: use after free
                        }
                }
                int size2 = img.width - img.height+100; //Vulnerability: integer underflow
                //printf("Size1:%d",size1);
                char* buff2=(char*)malloc(size2);
                memcpy(buff2,img.data,sizeof(img.data));
                int size3= img.width/img.height;
                //printf("Size2:%d",size3);
                char buff3[10];
                char* buff4 =(char*)malloc(size3);
                memcpy(buff4,img.data,sizeof(img.data));
                char OOBR_stack = buff3[size3+100]; //Vulnerability: out of bound read (stack)
                char OOBR_heap = buff4[100];
                buff3[size3+100]='c'; //Vulnerability: out of bound write (Stack)
                buff4[100]='c'; //Vulnerability: out of bound write (Heap)
                if(size3>10){
                                buff4=0; //memory leak?
                }
                else{
                        free(buff4);
                }
                free(buff2);
        }
        fclose(fp);
}
int main(int argc,char **argv)
{
        ProcessImage(argv[1]);
}
```

Listing 5.1: Kod źródłowy błędnego skrawka kodu *damnvuln.c*

5.4.2. Wynik w formacie diff - może zmienić na wynik finalny, ponieważ nie ma kolorowych linii dla +/-

```
diff --git a/damnvuln.c b/damnvuln.c
index e69de29..bb2e3ae 100644
--- a/damnvuln.c
+++ b/damnvuln.c
@@ -20,10 +20,12 @@ int ProcessImage(char* filename){
   fp = fopen(filename,"r");
   if(fp == NULL){
     printf("\nCan't open file or file doesn't exist.");
     exit(0);
   }
-        fprintf("\n\tHeader\twidth\theight\tdata\t\r\n");
+        printf("\n\tHeader\twidth\theight\tdata\n");

+        if (sizeof(struct Image) == 0) return 0; // Prevent division by zero in fread
+
   while(fread(&img,sizeof(img),1,fp)>0){
     printf("\n\t%s\t%d\t%d\t%s\r\n",img.header,img.width,img.height,img.data);
     int size1 = img.width + img.height; //Vulnerability: integer overflow
+               if (img.width > INT_MAX - img.height) continue; // Prevent integer overflow
     char* buff1=(char*)malloc(size1);
     memcpy(buff1,img.data,sizeof(img.data)); //Vulnerability: no data buffer size/malloc success check?
@@ -31,16 +33,15 @@ int ProcessImage(char* filename){
       free(buff1); //Vulnerability: double free
     }
     else{
       if(size1 == 123456){
         buff1[0]='a'; //Vulnerability: use after free
       }
     }
-               int size2 = img.width - img.height+100; //Vulnerability: integer underflow
-               //printf("Size1:%d",size1);
+               // Removed potential integer underflow by ensuring img.width is always greater than img.height before
↪  subtraction
+               int size2 = img.height > img.width ? 100 : img.width - img.height + 100;
     char* buff2=(char*)malloc(size2);
     memcpy(buff2,img.data,sizeof(img.data));
     int size3= img.width/img.height;
-               //printf("Size2:%d",size3);
+               // Check for division by zero
+               if (img.height == 0) continue;
     char buff3[10];
-               char* buff4 =(char*)malloc(size3);
-               memcpy(buff4,img.data,sizeof(img.data));
+               // Check if size3 is within buffer size
+               if (size3 < 0 || size3 >= sizeof(buff3)) continue;
     char OOBR_stack = buff3[size3+100]; //Vulnerability: out of bound read (stack)
-               char OOBR_heap = buff4[100];
+               char* buff4 = NULL;
+               if (size3 < sizeof(buff3)) {
+                       buff4 = (char*)malloc(size3);
+                       if (!buff4) continue; // Check malloc success
+                       memcpy(buff4,img.data,sizeof(img.data));
+               }
-               buff3[size3+100]='c'; //Vulnerability: out of bound write (Stack)
+               // Prevent out of bound write to buff3
+               if (size3 + 100 < sizeof(buff3)) buff3[size3+100] = 'c';
-               buff4[100]='c'; //Vulnerability: out of bound write (Heap)
+               // Prevent out of bound write to buff4 and potential memory leak
+               if (buff4 && size3 >= 0 && size3 < 100) buff4[size3] = 'c';
     if(size3>10){
-                               buff4=0; //memory leak?
+                               if (buff4) { free(buff4); buff4 = NULL; } // Properly free memory
     } else{
       free(buff4);
     }
     free(buff2);
@@ -50,8 +51,8 @@ int ProcessImage(char* filename){
   if(size3>10){
        buff4=0; //memory leak?
-
   }}
   fclose(fp);
 }
```

Listing 5.2: Wynik działania programu w formacie diff na kodzie źródłowym *damnvuln.c*

Oprócz przedstawionego powyżej wyniku, program wygenerował również raport w

formacie Markdown, który został przedstawiony w sekcji **??** oraz plik *damnvuln_fixed.c*, który zawiera poprawiony kod źródłowy.

5.4.3. Przygotowanie środowiska testowego

Dla podanego przykładu przygotowanie środowiska testowego polegało na skompilowaniu i uruchomieniu programu. W tym celu należało wykonać następujące kroki:

1. Skompilowanie programu za pomocą kompilatora *gcc*:

```
> gcc damnvuln.c -o damnvuln
```

2. Uruchomienie programu z wykorzystaniem przykładowego pliku wejściowego:

```
> ./damnvuln input.jpg
```

Jest to aplikacja lokalna dlatego przygotowanie środowiska testowego dla tego przykładu nie wymagało instalacji dodatkowych zależności, ani dopisywania tego kodu do istniejącej aplikacji webowej. Niestety wiele przykładów z tego zbioru do działania wymaga kodu źródłowego całej aplikacji, dlatego przygotowanie środowiska testowego dla tych przykładów było bardziej skomplikowane. W takich przypadkach należało wykonać następujące kroki:

1. Instalacja zależności.
2. Konfiguracja środowiska.
3. Dopisanie kodu pozwalającego na wykonanie.
4. Uruchomienie aplikacji.

5.4.4. Przeprowadzenie testów funkcjonalnych

Przeprowadzono test funkcjonalny, który polegał na wykonaniu programu z wykorzystaniem przykładowego pliku wejściowego. Program zwrócił błąd Segmentation Fault, co oznacza, że wystąpił błąd podczas wykonywania programu. W tym przypadku błąd ten został spowodowany przez błędy typu "Out of Bounds", które zostały wykryte przez program.

```
> ./damnvuln ~/Pictures/egzamin_praktyka.png

Header width height data

PNG

169478669 218103808 IHDR
Segmentation fault (core dumped)
```

Natomiast naprawiony program zwrócił:

```
> ./damnvuln_fixed ~/Pictures/egzamin_praktyka.png

Header width height data

PNG

169478669 218103808 IHDR
Integer underflow detected
```

Oznacza to że program wykrył błąd typu "Out of Bounds" i zwrócił informację o tym błędzie. Niestety sugerowana poprawa tego błędu wprowadziła jedynie kontrolę tych błędów. Część odpowiadająca za wyświetloną informację to:

```
1     if (img.height > img.width + 100) {
2       fprintf(stderr, "Integer underflow detected\n");
3       free(buff1);
4       fclose(fp);
5       exit(EXIT_FAILURE);
6   }
```

Listing 5.3: Fragment kodu odpowiadający za wyświetlenie informacji o błędzie

Aby zbadać jak różnorodne są wyniki programu gptester dla tego samego kodu bez zmiany parametrów wykonywania przeprowadzono analizę ponownie. Otrzymano wtedy znacznie inny wynik, który nadal wykrył błędy, ale zaimplementował inne rozwiązanie podatności.

```
 > gcc -c damnvuln.c -o damnvuln-fixed2
damnvuln.c: In function ‘ProcessImage’:
damnvuln.c:44:17: warning: implicit declaration of function ‘memcpy’ [-Wimplicit-
   44 |                 memcpy(buff1,img.data,sizeof(img.data));
      |                 ^~~~~~
damnvuln.c:6:1: note: include ‘<string.h>’ or provide a declaration of ‘memcpy’
    5 | #include<limits.h>
  +++ |+#include <string.h>
    6 |
damnvuln.c:44:17: warning: incompatible implicit declaration of built-in function
   44 |                 memcpy(buff1,img.data,sizeof(img.data));
      |                 ^~~~~~
damnvuln.c:44:17: note: include ‘<string.h>’ or provide a declaration of ‘memcpy’
```

Tym razem agent zwrócił kod, który się nie kompilował, ponieważ nie był dołączony plik nagłówkowy `string.h`. Po dopisaniu odpowiedniej biblioteki, program wykonał się poprawnie i zwrócił następujący wynik:

```
> ./damnvuln3 ~/Pictures/egzamin_praktyka.png

Header width height data

PNG

169478669 218103808 IHDR
Integer underflow detected in size2 calculation.
```

```
1     unsigned int size2;
2         if(__builtin_sub_overflow(img.width, img.height, &size2))
3         {
4             printf("Integer underflow detected in size2 calculation.");
5             fclose(fp);
6             exit(EXIT_FAILURE);
7         }
8         size2 += 100;
9         char* buff2 = (char*)malloc(size2);
```

Listing 5.4: Fragment kodu odpowiadający za wyświetlenie informacji o błędzie

Dla każdego z podanych przeze mnie danych wejściowych został wyświetlony komunikat o wykryciu błędu Integer Underflow.

5.4.5. Interpretacja wyników

Analiza wyników testów funkcjonalnych przeprowadzonych na programie `damnvuln` i jego zmodyfikowanych wersjach pozwala na dokonanie istotnych obserwacji dotyczących skuteczności działania narzędzia `gptester` oraz zdolności modeli językowych do wykrywania i naprawy błędów typu "Out of Bounds".

Pierwszy test funkcjonalny, w którym oryginalna wersja programu `damnvuln` zwróciła błąd Segmentation Fault, wskazuje na obecność poważnego błędu, który uniemożliwia poprawne wykonanie programu. Taki wynik podkreśla znaczenie analizy statycznej kodu w celu identyfikacji potencjalnych zagrożeń i wad, które mogą prowadzić do krytycznych awarii aplikacji. Przede wszystkim wskazuje to na dużą podatność i wysoki potencjał do eksploitacji programu. Złośliwy podmiot może wykorzystać ten błąd do nadpisania miejsca

w pamięci inaczej nie dostępnego, prowadząc do wykonania kodu arbitralnego, co może prowadzić do kradzieży danych, utraty poufności, a nawet całkowitego przejęcia kontroli nad systemem, zwłaszcza przy ustawionym bicie lepkim (sticky bit) podatnego programu.

W przypadku zmodyfikowanej wersji programu, gdzie błąd Segmentation Fault został zastąpiony komunikatem o błędzie Integer Underflow, obserwujemy, że narzędzie `gptester` było w stanie wykryć i częściowo naprawić błąd. Zmodyfikowany kod, chociaż poprawnie identyfikuje rodzaj błędu, wprowadza jedynie kontrolę tego błędu, nie adresując w pełni jego przyczyny. W tym przypadku natomiast nie da się wykorzystać błędu do wykonania kodu arbitralnego, ponieważ nie jest on już krytyczny. Jednakże, w przypadku gdyby błąd ten występował w innym miejscu programu, mógłby on prowadzić do nieprzewidywalnych konsekwencji, takich jak utrata danych, bądź nieprawidłowe działanie programu.

Kolejna iteracja testu, z wykorzystaniem innego wyniku generowanego przez `gptester`, przyniosła kod, który początkowo nie kompilował się z powodu brakującego pliku nagłówkowego. Po jego dołączeniu, program został uruchomiony i ponownie zwrócił informację o wykryciu błędu Integer Underflow, lecz w inny sposób niż poprzednio. Tym razem zastosowano funkcję sprawdzającą przepełnienie dla obliczeń, co stanowi bardziej zaawansowane i technicznie poprawne podejście do problemu.

Wnioski płynące z tych eksperymentów wskazują, że narzędzie `gptester` i wykorzystane w nim modele językowe posiadają zdolność do identyfikacji i proponowania poprawek dla wybranych rodzajów błędów bezpieczeństwa w kodzie. Jednakże, jakość i kompletność tych poprawek może być zmienna, co wymaga dalszej analizy i możliwej interwencji ze strony użytkownika. W kontekście błędów typu "Out of Bounds", `gptester` wykazał zdolność do wykrywania potencjalnych problemów, ale rozwiązania oferowane przez narzędzie wymagają dodatkowej weryfikacji i dostosowania, aby w pełni adresować przyczyny tych błędów.

Przedstawiony przypadek otrzymuje przeze mnie następujące oceny:

6. BADANIE FUNKCJONALNOŚCI NA APLIKACJI WEBOWEJ OWASP VULNERABLEAPP

Analiza funkcjonalności programu do analizy bezpieczeństwa została przeprowadzona z wykorzystaniem aplikacji webowej OWASP VulnerableApp. Jest to narzędzie celowo zawierające liczne podatności, które mają na celu symulację realnych luk bezpieczeństwa, co pozwala na dogłębne testowanie i ocenę narzędzi do skanowania podatności.

6.1. CHARAKTERYSTYKA APLIKACJI OWASP VULNERABLEAPP

Aplikacja OWASP VulnerableApp została zaprojektowana z myślą o dostarczeniu platformy edukacyjnej dla deweloperów oraz specjalistów od bezpieczeństwa, którzy pragną zgłębić wiedzę na temat bezpieczeństwa aplikacji webowych. Narzędzie to charakteryzuje się skalowalnością, elastycznością oraz łatwością integracji, czyniąc je idealnym środowiskiem do nauki oraz testowania.

6.2. TESTOWANE RODZAJE PODATNOŚCI

OWASP VulnerableApp umożliwia testowanie szerokiego zakresu podatności, w tym, ale nie ograniczając się do:

— Podatności JWT
— Wstrzykiwanie poleceń (Command Injection)
— Podatności związane z przesyłaniem plików (File Upload Vulnerability)
— Przejście ścieżki (Path Traversal)
— Iniekcje SQL (SQL Injection)
— Skrypty międzywitrynowe (XSS)
— Ataki oparte na External XML Entities (XXE)
— Open Redirect
— Server-Side Request Forgery (SSRF)

Zawarte podatności są reprezentatywne dla typowych zagrożeń w aplikacjach internetowych, co pozwala na wszechstronne i realistyczne testowanie narzędzi do ich wykrywania i naprawy.

6.3. ZAWARTOŚĆ ZNAJDUJĄCA SIĘ W REPOZYTORIUM

Repozytorium aplikacji VulnerableApp zawiera projekt aplikacji webowej napisany w następującym stosie technologicznym:

— Java 8
— Spring Boot
— Maven 3.6.1
— ReactJS
— Javascript/TypeScript

```
> ./main.py ../testing-envs/VulnerableApp/
```

```
  ___  ___  _____           _
 / __|| _ \|_   _| ___  ___| |_  ___  _ _
| (_ ||  _/  | |  / -_)(_-/|  _|/ -_)| '_|
 \___||_|    |_|  \___|/__/ \__|\___||_|

The static code analysis agent, version: assistant-0.3
```

```
2024-01-25 22:05:23: Beginning scan for ../testing-envs/VulnerableApp/
2024-01-25 22:05:23: Found 97 files to scan
2024-01-25 22:05:23: Tokens inside the directory: 77513
2024-01-25 22:05:23: Using model: gpt-4-1106-preview
2024-01-25 22:05:23: Beginning code analysis...
```

W repozytorium znajduje się 231 plików, które zawierają 229119 tokenów z uwzględnieniem wszystkich plików. Nasz program pokazał wartości dla plików zawierających kod, a dokładnie tych które nie są wyspecjalizowane w liście nazw do ignorowania.

6.4. PROCEDURA PRZEPROWADZENIA TESTÓW

Testy zostały przeprowadzone przy użyciu najnowszej wersji programu, zgodnie z następującymi krokami:

1. Przygotowanie środowiska testowego z wykorzystaniem aplikacji OWASP VulnerableApp.
2. Uruchomienie skanowania z wykorzystaniem programu.
3. Dokumentacja wykrytych podatności oraz sugerowanych przez program napraw.
4. Analiza efektywności napraw i ich wpływ na bezpieczeństwo aplikacji za pomocą innych skanerów podatności.

6.4.1. Oczekiwane rezultaty

W wyniku przeprowadzonych testów oczekujemy uzyskania szczegółowych danych na temat liczby wykrytych podatności, rodzajów podatności, a także czasu potrzebnego na ich wykrycie i naprawę. Dane te zostaną następnie wykorzystane do stworzenia szczegółowych wykresów i tabel ilustrujących skuteczność programu.

6.5. WNIOSKI I DALSZE KIERUNKI BADAŃ

Na podstawie zebranych danych zostaną wyciągnięte wnioski dotyczące skuteczności narzędzia w kontekście poszczególnych typów podatności oraz ogólnej wydajności. Dalsze badania mogą również koncentrować się na porównaniu wyników z innymi narzędziami dostępnymi na rynku oraz na rozwoju nowych funkcji i usprawnień w badanym programie.

6.6. WNIOSKI

Na podstawie przeprowadzonych badań eksperymentalnych udało się zweryfikować założenia dotyczące efektywności wykorzystania modeli AI w procesie identyfikacji i naprawy podatności w kodzie źródłowym. Główne wnioski to:

PODSUMOWANIE

[17]

BIBLIOGRAFIA

[1] Hammond Pearce, Benjamin Tan, B.A.R.K.B.D.G., *Can openai codex and other large language models help us fix security bugs?* 2022.

[2] Hammond Pearce, Benjamin Tan, B.A.R.K.B.D.G., *Examining zero-shot vulnerability repair with large language models*. 2022.

Spis rysunków

SPIS LISTINGÓW

Spis tabel

Dodatki

A. DODATEK 1

[20]