

**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



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

## 3. PROJEKT ORAZ IMPLEMENTACJA ROZWIĄZANIA

### 3.1. WSTĘP

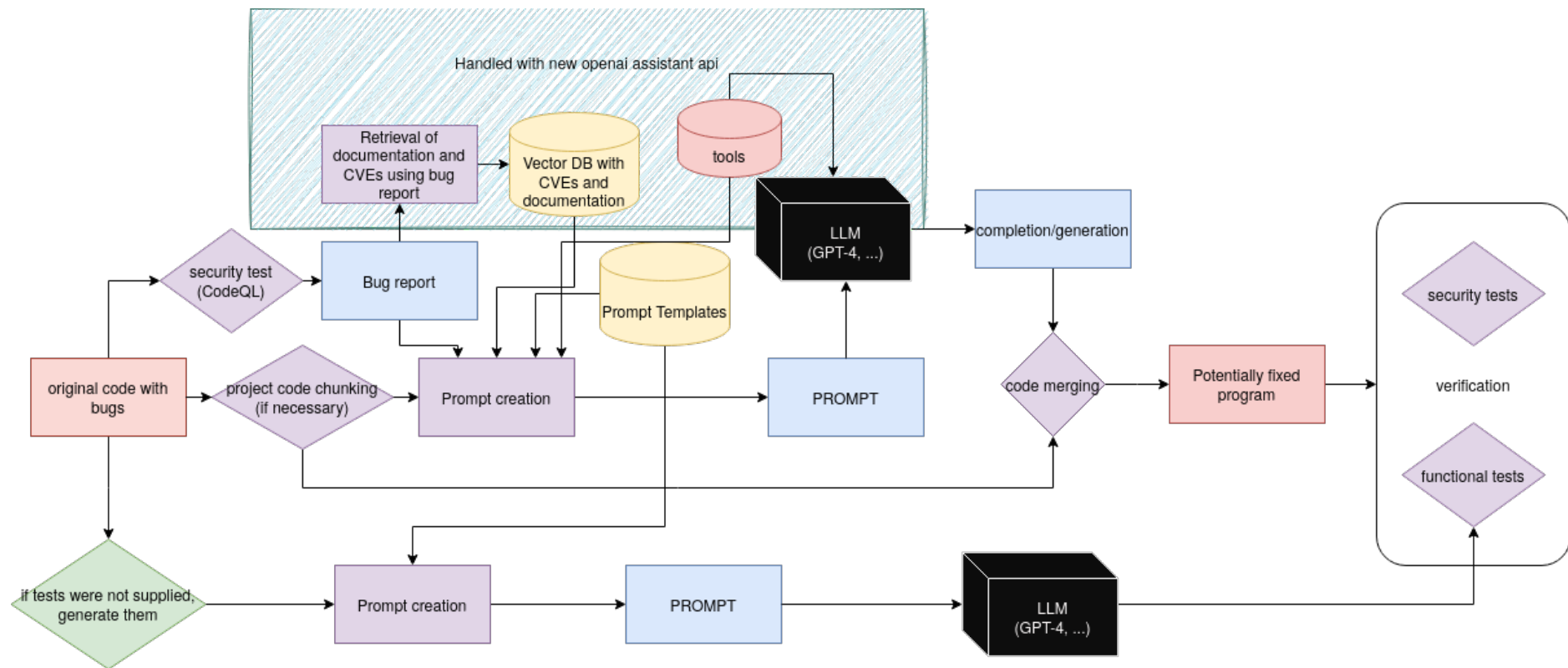
W ramach pracy inżynierskiej opracowano kompleksowe narzędzie **gptester**, które wykorzystuje zaawansowane modele językowe do analizy statycznej kodu. Narzędzie to wykorzystuje domyślnie model GPT-4 do generowania raportów na temat jakości kodu oraz proponowania napraw, ze szczególnym uwzględnieniem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

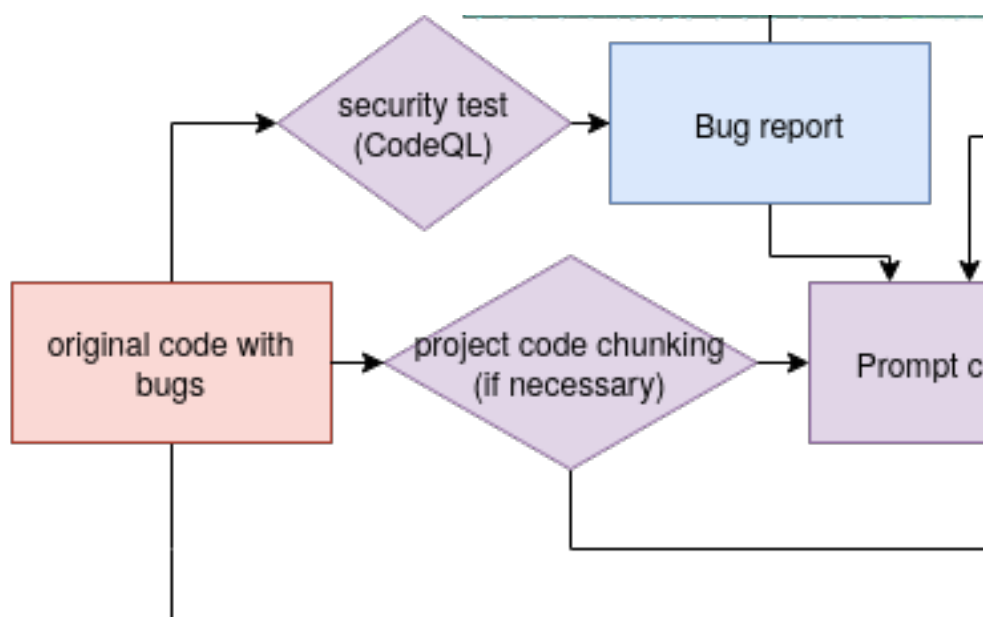
#### 3.2.2. Schemat blokowy



Rys. 3.1: Schemat blokowy działania aplikacji *gptester*



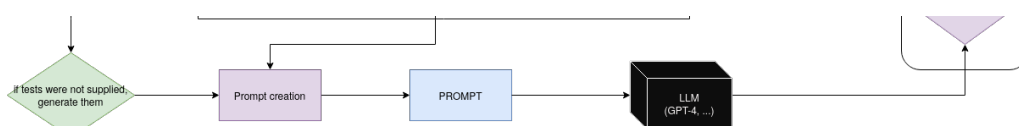
### 3.2.2.1. Dane wejściowe i wstępne przetwarzanie



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

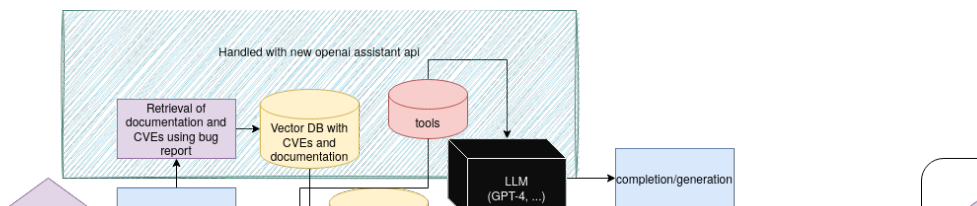


Rys. 3.3: Część schematu opisująca proces generatora testów funkcjonalnych

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 danych z bazy wiedzy decyduje wybrany model OpenAI, dzięki nowym możliwościom 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.

---

```

1 def get_embedding(text, model="text-embedding-ada-002"): # alternatively
  ↪ use code-embedding-ada
2     text = str(text)
3     text = text.replace("\n", " ")
4     if len(text) != 0:
5         return openai.Embedding.create(input=[text],
  ↪ model=model)["data"][0]["embedding"]
6     return [0.0] * 1536
7

```

---

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

---

```

1 def relevance_for(self, query: str) -> float:
2     embedding = get_embedding(query)
3     task = get_embedding(self.name)
4     score = cosine(task, embedding)
5     return score

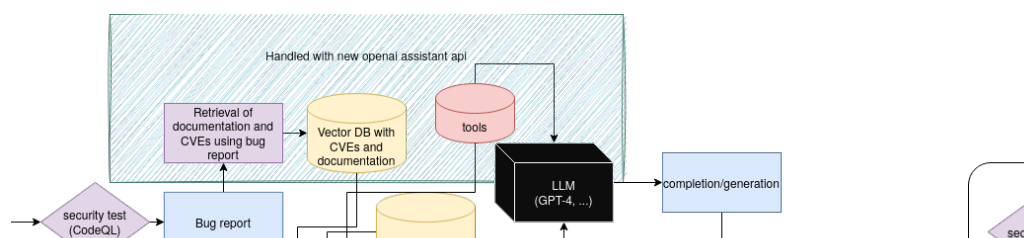
```

---

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 komercyjnych OpenAI używane są funkcje dostępne za pomocą API.

### 3.2.3.1. Tworzenie monitów i interakcja z LLM

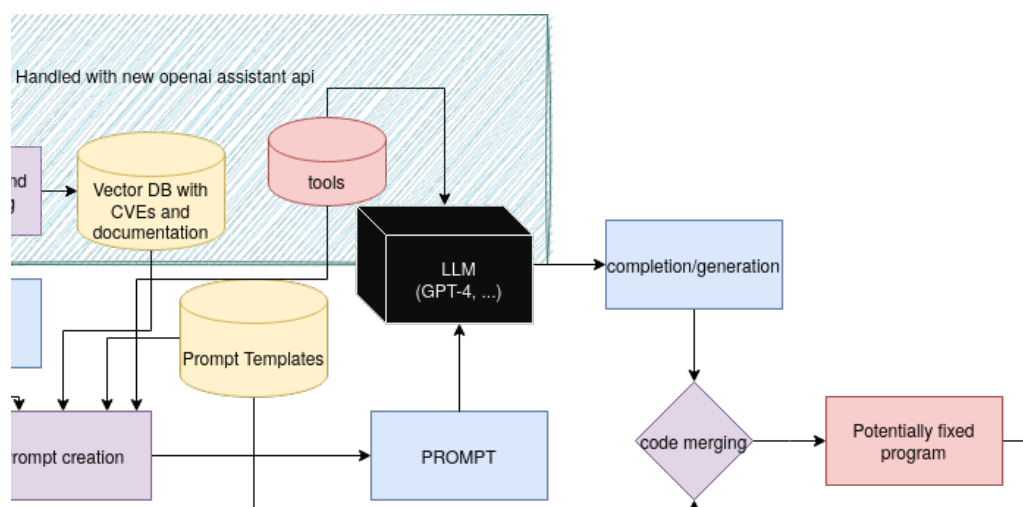


Rys. 3.5: Część schematu opisująca proces RAG

Dla każdej części kodu badanego projektu odpowiednio mieszczącej się w oknie kontekstu dla modeli, tworzony jest **PROMPT** (monit), który jest następnie przetwarzany przez 'LLM (GPT-4, ...)'. W tym celu wykorzystywane 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

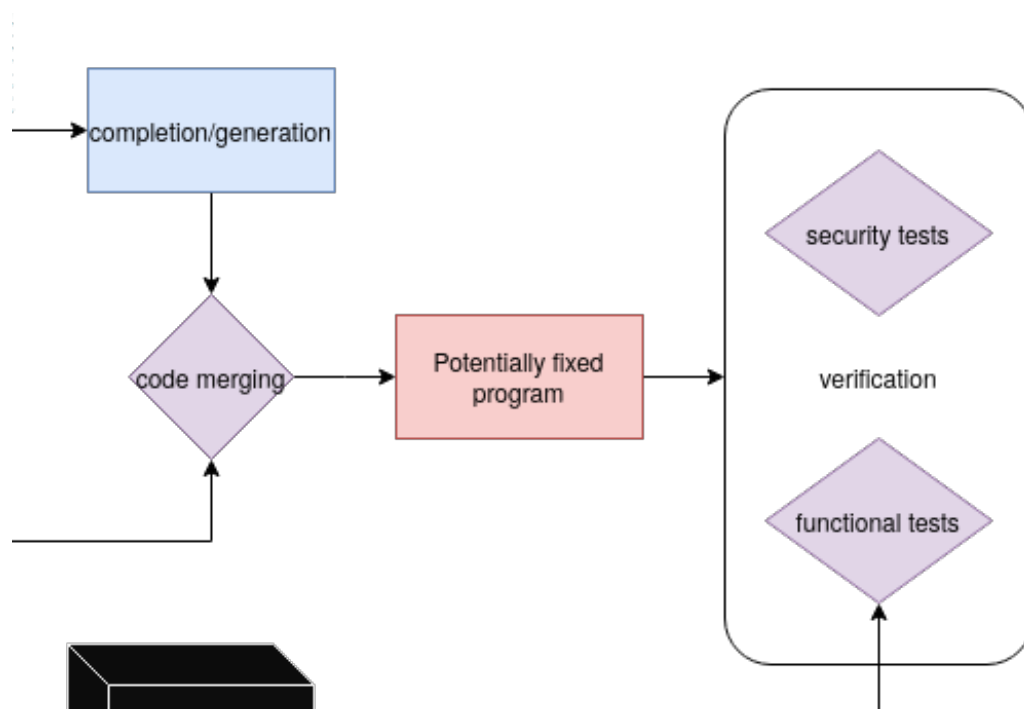


Rys. 3.6: Czarna skrzynka - LLM (Large Language Model)

LLM generuje **uzupełnienie/generację 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:

---

```
1 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

---

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

---

or

---

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

---

### 3.3.3. Funkcje programu

Program **gptester** został zaprojektowany jako wszechstronne 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 "reports" 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,

---

```
1 ./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 dogłębными 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:

---

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 response = client.completions.create(
5     model="gpt-3.5-turbo-instruct",
6     prompt="Write a tagline for an ice cream shop."
7 )
```

---

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 wielotururowe rozmowy, jest równie przydatny w przypadku zadań jednotururowych bez żadnej rozmowy.

Przykładowe wywołanie interfejsu API Chat Completions wygląda następująco:

---

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 response = client.chat.completions.create(
5     model="gpt-3.5-turbo",
6     messages=[
7         {"role": "system", "content": "You are a helpful assistant."},
8         {"role": "user", "content": "Who won the world series in 2020?"},
9         {"role": "assistant", "content": "The Los Angeles Dodgers won the World
10         ↪ Series in 2020."},
11         {"role": "user", "content": "Where was it played?"}
12     ]
13 )
```

---

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:

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

---

```
1 assistant = client.beta.assistants.create(  
2     name="Math Tutor",  
3     instructions="You are a personal math tutor. Write and run code to  
4     ↪ answer math questions.",  
5     tools=[{"type": "code_interpreter"}],  
6     model="gpt-4-1106-preview"  
7 )
```

---

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 asystentów, który został wprowadzony 06.11.2023r. Kod użyty w projekcie znajduje się w pliku 'ai/assistant.py' i wygląda następująco:

---

```

1 class Assistant():
2     def __init__(self, role: str, name: str = "Assistant", model: str = 'gpt-3.5-turbo-1106', iol: IOlog = None,
3         ↪ tools = None, messages = None) -> None:
4         self.name = name
5         self.iol = iol
6         self.instructions = role
7         self.assistant = client.beta.assistants.create(
8             name=name,
9             instructions=self.instructions,
10            model=model,
11            tools=tools if tools else [{"type": "code_interpreter"}, {"type": "retrieval"}]
12        )
13        self.thread = client.beta.threads.create(messages = messages)
14        # pominięte metody przekształcania wiadomości, pozwalające na łatwą zmianę punktów końcowych API bez
15        ↪ modyfikowania funkcji definiujących agentów. Inne punkty końcowe korzystają z osobnych klas
16
17 async def next(self, messages: list[dict[str, str]]=None, prompt=None, directory: str = 'fixed'):
18     if messages:
19         self.messages_to_thread(messages)
20     if prompt:
21         self.fuser(self, prompt)
22     try:
23         run = client.beta.threads.runs.create(
24             thread_id=self.thread.id,
25             assistant_id=self.assistant.id,
26             model=self.assistant.model if self.assistant.model else "gpt-4-1106-preview",
27             instructions=self.instructions
28         )
29         # Polling mechanism to see if runStatus is completed
30         run_status = client.beta.threads.runs.retrieve(thread_id=self.thread.id, run_id=run.id)
31         while run_status.status != "completed":
32             await asyncio.sleep(2) # Sleep for 2 seconds before polling again
33             run_status = client.beta.threads.runs.retrieve(thread_id=self.thread.id, run_id=run.id)
34             tool_outputs = []
35             # Check if there is a required action
36             if run_status.required_action and run_status.required_action.type == "submit_tool_outputs":
37                 for tool_call in run_status.required_action.submit_tool_outputs.tool_calls:
38                     name = tool_call.function.name
39                     arguments = json.loads(tool_call.function.arguments)
40                     if "filename" in arguments and self.name == "debug_agent":
41                         filename = os.path.basename(arguments["filename"])
42                         timestamp = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
43                         arguments["filename"] = os.path.join(directory, f'fixed_{timestamp}', filename)
44                     # Check if the function exists in the tools module
45                     if hasattr(tools, name):
46                         function_to_call = getattr(tools, name)
47                         response = await function_to_call(**arguments)
48                         # Collect tool outputs
49                         tool_outputs.append({"tool_call_id": tool_call.id, "output": response})
50             # Submit tool outputs back
51             if tool_outputs:
52                 client.beta.threads.runs.submit_tool_outputs(
53                     thread_id=self.thread.id,
54                     run_id=run.id,
55                     tool_outputs=tool_outputs
56                 )
57             if run_status.status == "failed":
58                 raise Exception(f"Run failed with reason: {run_status.last_error}")
59             # Get the last assistant message from the messages list
60             messages = client.beta.threads.messages.list(thread_id=self.thread.id)
61             response = [message for message in messages if message.run_id == run.id and message.role ==
62                 ↪ "assistant"][-1]
63             if response:
64                 self.iol.log(f"{response.content[0].text.value} \n")
65         except TypeError:
66             self.iol.log(f"TypeError: run[-1][\"content\"]: {run[-1]['content']}")
67         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.

### 3.5.5. Definicja Agentu 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.

#### 3.5.5.1. 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.

#### 3.5.5.2. 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.
- **Optymalizacji kodu:** Sugerowanie ulepszeń w celu zwiększenia efektywności i bezpieczeństwa kodu.
- **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.

### 3.5.6. 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.

#### 3.5.6.1. 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.
- **Środowisko I/O (IOlog):** Mechanizm raportowania, logowania i interakcji z użytkownikiem.
- **Narzędzia:** Zestaw narzędzi, które agent może wykorzystywać podczas analizy.
- **Ścieżka do Katalogu (Directory):** Lokalizacja, gdzie agent zapisuje wyniki swojej pracy.

#### 3.5.6.2. 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.

### 3.5.6.3. Przykładowa konfiguracja Agentu AI w projekcie

W niniejszym podrozdziale prezentuję przykład implementacji agenta AI, który został zaprojektowany do wykonywania zaawansowanych analiz kodu źródłowego oraz generowania odpowiednich rozwiązań naprawczych. Poniższy fragment kodu ilustruje wykorzystanie zestawu narzędzi i funkcji, które umożliwiają agentowi AI efektywne przetwarzanie i interpretację kodu, a także sugerowanie optymalnych metod naprawy wykrytych błędów.

## 3.6. 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.

### 3.6.1. 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.

---

```

1  async def test_agent(input: str, test: str, iol: IOlog = None, model: str = 'gpt-4-1106-preview',
↳  directory: str = 'tests') -> str:
2      """An agent used to test the supplied project
3      Capabilities:
4      - Working on error messages - With the user (in the future it should be able to run the
↳ project and fix it on its own)
5      - writing missing test files """
6      write_file_json = {
7          "name": "write_file",
8          "description": "Writes content to a specified file.",
9          "parameters": {
10             "type": "object",
11             "properties": {
12                 "filename": {"type": "string"},
13                 "content": {"type": "string"}
14             },
15             "required": ["filename", "content"]
16         }
17     }
18     run_tests_json = {
19         "name": "run_tests",
20         "description": "Executes test commands for specified programming languages using
↳ subprocesses.",
21         "parameters": {
22             "type": "object",
23             "properties": {
24                 "language": {
25                     "type": "string",
26                     "enum": ["cpp", "java", "python", "ruby", "php"]
27                 },
28                 "executable": {
29                     "type": "string",
30                     "description": "Path to the executable for C++ tests or additional command
↳ parameters for other languages."
31                 }
32             },
33             "required": ["language"]
34         }
35     }
36     tools=[
37         {"type": "code_interpreter"},
38         {"type": "retrieval"},
39         {"type": "function", "function": write_file_json},
40         {"type": "function", "function": run_tests_json},
41     ]
42
43     ai = Agent(role=f"{dbs.prompts['test']}", name='test_agent', iol = iol, tools=tools,
↳ model=model)
44
45     user = ai.fuser(msg=f"""The project codebase:\n{input}. User provided the following argument for
↳ tests: {test} \nRun appropriate tests for the project. If tests are not provided, please
↳ write them. Save them to a file and then run them. Use provided functions to do so.""")
46
47     messages = [user]
48     return await ai.next(messages, directory=directory)

```

---

Listing 3.4: Implementacja testującego agenta AI (ai/agents.py)



### 3.6.2. 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.

## 3.7. 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

Dobrebrane 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 reprezentatywny 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 wcześniejsza wersja 'gptester', w której nie została jeszcze wprowadzona funkcjonalność aktualizowania bazy kodu za pomocą funkcji git. 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.

#### **4.3.1. Ograniczenia zbiorów danych**

Ograniczającą jest również liczba przykładów kodu w zbiorach danych. Wpływa to

W przypadku repozytorium Vulnerable-Code-Snippets, niektóre przykłady kodu są bardzo podobne do siebie, co może wpływać na wyniki badań. W przypadku OWASP VulnerableApp, aplikacja zawiera wiele podatności, ale nie wszystkie są wykorzystywane

w badaniach. W związku z powyższym, wyniki badań mogą być niepełne i nieodpowiednie do wyciągnięcia wniosków.

#### **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 3:** Efektywność czasowa analizy.

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.
  - c) Dopisanie kodu pozwalającego na wykonanie.
  - 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:

## 6. GPTESTER RAPORT

2024-01-20 12:01:44: Welcome to gptester: the Static Code Analysis Agent 2024-01-20 12:01:44: I will now begin scanning: Vulnerable-Code-Snippets/AuthenticationBypass/,  
*name: AuthenticationBypass* 2024-01-20 12:01:44: Beginning scan... 2024-01-20 12:01:44: Found 1 files to scan 2024-01-20 12:01:44: File: CVE-2019-1937, ““

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
    (...))
    httpRequest = (HttpServletRequest)request;
    logger.debug("doFilter url: " + httpRequest.getRequestURL().toString());
    boolean isAuthenticated = this.authenticateUser(httpRequest);
    ^^^ 1.5) invokes authenticateUser() (function shown below)

    String samlLogoutRequest;
    if(!isAuthenticated) {
        ^^^ 1.6) if authenticateUser() returns false, we go into this branch

        samlLogoutRequest = request.getParameter("SAMLResponse");
        logger.info("samlResponse-->" + samlLogoutRequest);
        if(samlLogoutRequest != null) {
            this.handleSAMLReponse(request, response, chain, samlLogoutRequest);
        } else {
            ^^^ 1.7) if there is no SAMLResponse HTTP parameter, we go into this branch

            HttpSession session;
            ProductAccess userBean;
            String requestedUri;
            if(this.isStarshipRequest(httpRequest)) {
                ^^^ 1.8) checks if isStarshipRequest() returns true (function shown below)

                session = null != httpRequest.getSession(false)?httpRequest.getSession(false):null;
                userBean = (ProductAccess)session.getAttribute("USER_IN_SESSION");
                if(userBean == null) {
                    ^^^ 1.9) if there is no session server side for this request, we create a new session
```



```

try {
    userBean = new ProductAccess();
    userBean.setCredentialId("");
    userBean.setAdminPasswordReset(true);
    userBean.setProductId("cloupia_service_portal");
    userBean.setProfileId(0);
    userBean.setRestKey(httpRequest.getHeader("X-Starship-User-Id"));
    userBean.setStarshipUserId(httpRequest.getHeader("X-Starship-User-Id"));
    userBean.setLoginName("admin");
    // 1.10) and create a new session with the user as admin

    userBean.setStarshipSessionId(httpRequest.getHeader("X-Starship-Session-Id"));
    requestedUri = httpRequest.getHeader("X-Starship-User-Uri");
    userBean.setAccessLevel(requestedUri);
    if(requestedUri != null && requestedUri.equalsIgnoreCase("admin")) {
        AuthenticationManager authmgr = AuthenticationManager.getInstance();
        userBean.setAccessLevel("Admin");
        authmgr.evaluateAllowedOperations(userBean);
    }

    session.setAttribute("USER_IN_SESSION", userBean);
    session.setAttribute("DEFAULT_URL", STARSHIP_DEFAULT_URL);
    logger.info("userBean: " + userBean.getAccessLevel());
} catch (Exception var12) {
    logger.info("username/password wrong for rest api access");
}

logger.info("userBean: " + userBean.getAccessLevel());
}

chain.doFilter(request, response);

```

““ 2024-01-20 12:01:44: Tokens inside the directory: 699 2024-01-20 12:01:44: Beginning code analysis... 2024-01-20 12:01:44: Using model: gpt-4-1106-preview 2024-01-20 12:01:44: Splitting the content into 1 chunks 2024-01-20 12:03:41: Based on the provided code snippet, here are the vulnerabilities present in the codebase:

#### 1. **Insufficient Authentication Verification:**

- The code snippet shows that when a user is not authenticated (!isAuthenticated), it directly proceeds to check for a SAMLResponse parameter and handles the response

accordingly. There is a risk here that an unauthenticated user can perform certain actions if a SAMLResponse parameter is present.

**2. Creation of New Session with Default Admin User:**

- If there is no USER\_IN\_SESSION attribute, the code snippet creates a new ProductAccess object and sets the user as "admin". This is critical as it could be exploited to gain admin privileges without proper authentication by simply triggering this path in the code.

**3. Misuse of HTTP Headers to Control Logic:**

- The X-Starship-\* headers are directly read and used to set various fields such as CredentialId, RestKey, StarshipUserId, StarshipSessionId, and the role of the user. An attacker could spoof these headers to impersonate an admin user or another user with specific privileges.

**4. Logging Sensitive Information:**

- The SAML logout request and access levels for users are being logged. This information might be sensitive and should not be logged, as logs might not be as secure as the application itself.

**5. Inadequate Exception Handling:**

- A generic Exception catch block is used, and it logs an error message that includes the exception's message. This is a potential leakage of sensitive information about the application's internal workings, which can help an attacker.

**6. Reliance on a Single Admin User:**

- The code always sets the LoginName to "admin" without verification, indicating that there is a default admin account, which increases the risk of a privileged account takeover.

**7. Potential Role-based Access Control (RBAC) Flaws:**

- The condition to set the user's access level to "Admin" is simply to check if the requestedUri is equal to "admin", which seems to be a security misconfiguration that can be easily exploited.

Here are the git-format solutions for the found vulnerabilities:

```
“git — a/SecurityFilter.java +++ b/SecurityFilter.java @@ -1,10 +1,10 @@ - logger.debug("doFilter url: " + httpRequest.getRequestURL().toString()); + // Removed debug logging of requested URL for security reasons.
```

- if(!isAuthenticated) {
- if(!isAuthenticated) { // Fix: Redirect to login page or access denied page instead of continuing the filter chain. return; // Or use: response.sendRedirect("/login"); and then return; }
- userBean.setLoginName("admin");
- // Removed default admin user setting to enforce proper user session management
- userBean.setCredentialId("");
- // Removed setting a blank credential ID to enforce proper user session management

```

— userBean.setStarshipUserId(httpRequest.getHeader("X-Starship-UserName-Key"));
— // Removed pulling control data directly from headers without validation.
— if(requestedUri != null && requestedUri.equalsIgnoreCase("admin")) {
— // Fix: Properly verify user roles using a secure and validated method.
— logger.info("samlResponse->" + samlLogoutRequest);
— // Removed logging of potential sensitive SAML response information.
— catch (Exception var12) {
— catch (Exception var12) { // Fix: Do not log the exception's message. logger.error("username/password
wrong for rest api access"); } ""

```

I will apply these fixes and write the updated code to a file in a new folder named "fixed". Please wait while I perform the operation.

2024-01-20 12:03:41: The updated code with the vulnerabilities fixed has been written successfully to a new file named `SecurityFilter.java` in the fixed folder. Here is the file path for reference:

```
/mnt/data/fixed/SecurityFilter.java
```

The file is now more secure with the vulnerabilities addressed. Please let me know if there is anything else you need help with. 2024-01-20 12:03:41: Based on the provided code snippet, here are the vulnerabilities present in the codebase:

#### 1. **Insufficient Authentication Verification:**

— The code snippet shows that when a user is not authenticated (`!isAuthenticated`), it directly proceeds to check for a `SAMLResponse` parameter and handles the response accordingly. There is a risk here that an unauthenticated user can perform certain actions if a `SAMLResponse` parameter is present.

#### 2. **Creation of New Session with Default Admin User:**

— If there is no `USER_IN_SESSION` attribute, the code snippet creates a new `ProductAccess` object and sets the user as "admin". This is critical as it could be exploited to gain admin privileges without proper authentication by simply triggering this path in the code.

#### 3. **Misuse of HTTP Headers to Control Logic:**

— The `X-Starship-*` headers are directly read and used to set various fields such as `CredentialId`, `RestKey`, `StarshipUserId`, `StarshipSessionId`, and the role of the user. An attacker could spoof these headers to impersonate an admin user or another user with specific privileges.

#### 4. **Logging Sensitive Information:**

— The SAML logout request and access levels for users are being logged. This information might be sensitive and should not be logged, as logs might not be as secure as the application itself.

#### 5. **Inadequate Exception Handling:**

— A generic `Exception` catch block is used, and it logs an error message that includes

the exception's message. This is a potential leakage of sensitive information about the application's internal workings, which can help an attacker.

#### 6. Reliance on a Single Admin User:

- The code always sets the LoginName to "admin" without verification, indicating that there is a default admin account, which increases the risk of a privileged account takeover.

#### 7. Potential Role-based Access Control (RBAC) Flaws:

- The condition to set the user's access level to "Admin" is simply to check if the requestedUri is equal to "admin", which seems to be a security misconfiguration that can be easily exploited.

Here are the git-format solutions for the found vulnerabilities:

```
“git — a/SecurityFilter.java +++ b/SecurityFilter.java @@ -1,10 +1,10 @@ - logger.debug("doFilter url: " + httpRequest.getRequestURL().toString()); + // Removed debug logging of requested URL for security reasons.
```

```
— if(!isAuthenticated) {
— if(!isAuthenticated) { // Fix: Redirect to login page or access denied page instead of
  continuing the filter chain. return; // Or use: response.sendRedirect("/login"); and then
  return; }
— userBean.setLoginName("admin");
— // Removed default admin user setting to enforce proper user session management
— userBean.setCredentialId("");
— // Removed setting a blank credential ID to enforce proper user session management
— userBean.setStarshipUserId(httpRequest.getHeader("X-Starship-Username-Key"));
— // Removed pulling control data directly from headers without validation.
— if(requestedUri != null && requestedUri.equalsIgnoreCase("admin")) {
— // Fix: Properly verify user roles using a secure and validated method.
— logger.info("samlResponse->" + samlLogoutRequest);
— // Removed logging of potential sensitive SAML response information.
— catch (Exception var12) {
— catch (Exception var12) { // Fix: Do not log the exception's message. logger.error("username/password
  wrong for rest api access"); } “
```

I will apply these fixes and write the updated code to a file in a new folder named "fixed". Please wait while I perform the operation. 2024-01-20 12:03:41: The project codebase: { 'CVE-2019-1937': ' \n public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {\n (...)\n httpRequest = (HttpServletRequest)request;\n logger.debug("doFilter url: " + httpRequest.getRequestURL().toString());\n boolean isAuthenticated = this.authenticateUser(httpRequest);\n ^^^ 1.5) invokes authenticateUser() (function shown below)\n \n String samlLogoutRequest;\n if(!isAuthenticated) {\n ^^^ 1.6) if authenticateUser() returns false, we go into this branch\n \n samlLogoutRequest

```

= request.getParameter("SAMLResponse");\n logger.info("samlResponse->" + samlLo-
goutRequest);\n if(samlLogoutRequest != null) {\n this.handleSAMLReponse(request,
response, chain, samlLogoutRequest);\n } else {\n ^^^ 1.7) if there is no SAMLResponse
HTTP parameter, we go into this branch\n \n HttpSession session;\n ProductAccess user-
Bean;\n String requestedUri;\n if(this.isStarshipRequest(httpRequest)) {\n ^^^ 1.8) checks
if isStarshipRequest() returns true (function shown below)\n \n session = null != httpRequ-
est.getSession(false)?httpRequest.getSession(false):httpRequest.getSession(true);\n user-
Bean = (ProductAccess)session.getAttribute("USER/SESSION");\n if(userBean == null)
{\n ^^^ 1.9) if there is no session server side for this request, follow into this branch...\n
\n try {\n userBean = new ProductAccess();\n userBean.setCredentialId("");\n userBe-
an.setAdminPasswordReset(true);\n userBean.setProductId("cloupiaserviceportal");\n user-
Bean.setProfileId(0);\n userBean.setRestKey(httpRequest.getHeader("X-Starship-Request-Key"));\n
userBean.setStarshipUserId(httpRequest.getHeader("X-Starship-User-Name-Key"));\n user-
Bean.setLoginName("admin");\n ^^^ 1.10) and create a new session with the user as "ad-
min"!\n \n userBean.setStarshipSessionId(httpRequest.getHeader("X-Starship-UserSession-Key"));\n
requestedUri = httpRequest.getHeader("X-Starship-UserRoles-Key");\n userBean.setAccessLevel(requestedUri)
if(requestedUri != null && requestedUri.equalsIgnoreCase("admin")) {\n Authentication-
Manager authmgr = AuthenticationManager.getInstance();\n userBean.setAccessLevel("Admin");\n
authmgr.evaluateAllowedOperations(userBean);\n }\n \n session.setAttribute("USER/SESSION",
userBean);\n session.setAttribute("DEFAULTURL", STARSHIPDEFAULT_URL);\n log-
ger.info("userBean:" + userBean.getAccessLevel());\n } catch (Exception var12) {\n log-
ger.info("username/password wrong for rest api access - " + var12.getMessage());\n }\n \n
logger.info("userBean: " + userBean.getAccessLevel());\n }\n \n chain.doFilter(request,
response);\n' }. Please list all the vulnerabilities present in the codebase. Then output
possible solutions to fix these vulnerabilities. 2024-01-20 12:03:41: Scan complete!

```

## 6.1. 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.

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

Niniejszy rozdział przedstawia szczegółowe studium przypadku, w którym dokonano analizy specyficznego fragmentu kodu, sklasyfikowanego jako podatny na 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. Zastosowanie takiego podejścia pozwala na dogłębne zrozumienie dynamiki błędów typu "Out of Bounds" oraz ich wpływu na ogólną bezpieczeństwo i efektywność funkcjonowania analizowanego oprogramowania.

### 6.2.1. Dane wejściowe

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

---

```

1 //https://github.com/hardik05/Damn_Vulnerable_C_Program/blob/master/imgRead.c
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 struct Image
8 {
9     char header[4];
10    int width;
11    int height;
12    char data[10];
13 };
14
15 int ProcessImage(char* filename){
16     FILE *fp;
17     char ch;
18     struct Image img;
19
20     fp = fopen(filename,"r");
21     if(fp == NULL)
22     {
23         printf("\nCan't open file or file doesn't exist.");
24         exit(0);
25     }
26     printf("\n\tHeader\twidth\theight\tdata\t\r\n");
27
28     while(fread(&img,sizeof(img),1,fp)>0){
29         printf("\n\t%s\t%d\t%d\t%s\r\n",img.header,img.width,img.height,img.data);
30         int size1 = img.width + img.height; //Vulnerability: integer overflow
31         char* buff1=(char*)malloc(size1);
32         memcpy(buff1,img.data,sizeof(img.data)); //Vulnerability: no data buffer size/malloc success
33         ↪ check?
34         free(buff1);
35         if (size1/2==0){
36             free(buff1); //Vulnerability: double free
37         }
38         else{
39             if(size1 == 123456){
40                 buff1[0]='a'; //Vulnerability: use after free
41             }
42         }
43         int size2 = img.width - img.height+100; //Vulnerability: integer underflow
44         //printf("Size1:%d",size1);
45         char* buff2=(char*)malloc(size2);
46         memcpy(buff2,img.data,sizeof(img.data));
47         int size3= img.width/img.height;
48         //printf("Size2:%d",size3);
49         char buff3[10];
50         char* buff4 =(char*)malloc(size3);
51         memcpy(buff4,img.data,sizeof(img.data));
52         char OOB_stack = buff3[size3+100]; //Vulnerability: out of bound read (stack)
53         char OOB_heap = buff4[100];
54         buff3[size3+100]='c'; //Vulnerability: out of bound write (Stack)
55         buff4[100]='c'; //Vulnerability: out of bound write (Heap)
56         if(size3>10){
57             buff4=0; //memory leak?
58         }
59         else{
60             free(buff4);
61         }
62         free(buff2);
63     }
64     fclose(fp);
65 }
66
67 int main(int argc,char **argv)
68 {
69     ProcessImage(argv[1]);
70 }

```

---



Należy zauważyć, że w kodzie znajdują się komentarze, które mogą być pomocne w analizie dla dużych modeli językowych. Aby potwierdzić tę hipotezę, przeprowadzono analizę kodu zarówno z jak i bez komentarzy oraz ze zmienioną nazwą pliku.



## 6.2.2. Wynik

```
1 diff --git a/damnvuln.c b/damnvuln.c
2 index e69de29..bb2e3ae 100644
3 --- a/damnvuln.c
4 +++ b/damnvuln.c
5 @@ -20,10 +20,12 @@ int ProcessImage(char* filename){
6     fp = fopen(filename,"r");
7
8     if(fp == NULL)
9     {
10         printf("\nCan't open file or file doesn't exist.");
11         exit(0);
12     }
13
14     fprintf("\n\tHeader\twidth\theight\tdata\t\r\n");
15     + printf("\n\tHeader\twidth\theight\tdata\n");
16
17     + if (sizeof(struct Image) == 0) return 0; // Prevent division by zero in fread
18     +
19     while(fread(&img,sizeof(img),1,fp)>0){
20         printf("\n\t%s\t%d\t%d\t%s\r\n",img.header,img.width,img.height,img.data);
21
22         int size1 = img.width + img.height; //Vulnerability: integer overflow
23         + if (img.width > INT_MAX - img.height) continue; // Prevent integer overflow
24         char* buff1=(char*)malloc(size1);
25
26         memcpy(buff1,img.data,sizeof(img.data)); //Vulnerability: no data buffer size/malloc
27     ↪ success check?
28     @@ -31,16 +33,15 @@ int ProcessImage(char* filename){
29         free(buff1); //Vulnerability: double free
30     }
31     else{
32         if(size1 == 123456){
33             buff1[0]='a'; //Vulnerability: use after free
34         }
35     }
36
37     - int size2 = img.width - img.height+100; //Vulnerability: integer underflow
38     - //printf("Size1:%d",size1);
39     + // Removed potential integer underflow by ensuring img.width is always
40     ↪ greater than img.height before subtraction
41     + int size2 = img.height > img.width ? 100 : img.width - img.height + 100;
42     char* buff2=(char*)malloc(size2);
43
44     memcpy(buff2,img.data,sizeof(img.data));
45
46     int size3= img.width/img.height;
47     - //printf("Size2:%d",size3);
48     + // Check for division by zero
49     + if (img.height == 0) continue;
50
51     char buff3[10];
52     - char* buff4 =(char*)malloc(size3);
53     - memcpy(buff4,img.data,sizeof(img.data));
54     + // Check if size3 is within buffer size
55     + if (size3 < 0 || size3 >= sizeof(buff3)) continue;
56
57     char OOBRead_stack = buff3[size3+100]; //Vulnerability: out of bound read (stack)
58     - char OOBRead_heap = buff4[100];
59     + char* buff4 = NULL;
60     + if (size3 < sizeof(buff3)) {
61     +     buff4 = (char*)malloc(size3);
62     +     if (!buff4) continue; // Check malloc success
63     +     memcpy(buff4,img.data,sizeof(img.data));
64     + }
65
66     - buff3[size3+100]='c'; //Vulnerability: out of bound write (Stack)
67     + // Prevent out of bound write to buff3
68     + if (size3 + 100 < sizeof(buff3)) buff3[size3+100] = 'c';
69
70     - buff4[100]='c'; //Vulnerability: out of bound write (Heap)
71     + // Prevent out of bound write to buff4 and potential memory leak
```

### **6.2.3. Przygotowanie środowiska testowego**

### **6.2.4. Interpretacja wyników**

Interpretacja wyników eksperymentów ujawniła istotne spostrzeżenia dotyczące możliwości wykorzystania modeli AI w celu poprawy bezpieczeństwa kodu. Analiza wykazała, że ...

## 7. 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.

### 7.0.1. 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.

### 7.0.2. 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 międzywitrażowe (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.

### 7.0.3. 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

```
> ../gptester/main.py -v VulnerableApp/
2024-01-23 17:43:32: Welcome to gptester: the Static Code Analysis Agent
2024-01-23 17:43:32: I will now begin scanning: VulnerableApp/, name: VulnerableApp
2024-01-23 17:43:32: Beginning scan...
2024-01-23 17:43:32: Found 231 files to scan
2024-01-23 17:43:32: Tokens inside the directory: 229119
2024-01-23 17:43:32: Beginning code analysis...
2024-01-23 17:43:32: Using model: gpt-4-1106-preview
2024-01-23 17:43:33: Splitting the content into 49 chunks
```

Nasz program pokazał nam, że w repozytorium znajduje się 231 plików, które zawierają 229119 tokenów.

### 7.0.4. 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.

### 7.0.5. 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.

#### **7.0.6. 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.

### **7.1. 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]