

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	4
Pytania badawcze	4
Hipotezy	5
Uzasadnienie tytułu	5
Omówienie literatury naukowej i stopnia jej przydatności	5
Cel pracy	5
Zakres pracy	6
Analiza istniejącej literatury oraz dotychczasowych badań	6
1. Analiza istniejącej literatury oraz dotychczasowych badań	7
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	7
1.1.1. Metodyka	7
1.1.2. Wyniki	7
1.2. Examining Zero-Shot Vulnerability Repair with Large Language Models - Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, Brendan Dolan-Gavitt	7
1.3. Różnice między obecną pracą a istniejącą literaturą	8
2. Metodyka rozwiązania	10
3. Projekt oraz implementacja rozwiązania	12
3.1. Wstęp	12
3.2. Architektura systemu	12
3.2.1. Ogólny opis	12
3.2.2. Schemat blokowy	12
3.2.3. Wzbogacanie zapytań(promptów)	14
3.3. Implementacja oraz użycie	18
3.3.1. Środowisko programistyczne i wymagania	18
3.3.2. Uruchomienie programu	18
3.3.3. Funkcje programu	19
3.4. Integracja z CodeQL	21
3.5. Modele językowe	21

3.5.1.	Wykorzystanie Modeli Językowych	21
3.5.2.	Dostępne Modele	22
3.5.3.	Inżynieria Poleceń (Promptów)	22
3.5.4.	Integracja z API OpenAI	23
3.5.5.	Implementacja w projekcie	24
3.6.	Definicja Agenta AI	27
3.6.1.	Cechy Charakterystyczne	27
3.6.2.	Zastosowanie w gptester	27
3.7.	Konfiguracja Agentów AI	28
3.7.1.	Parametry Konfiguracyjne	28
3.7.2.	Narzędzia i Funkcje	28
3.8.	Rozwój i plany na przyszłość	29
3.8.1.	Obecne osiągnięcia	29
3.8.2.	Planowane rozszerzenia	29
3.9.	Podsumowanie	30
4.	Zbiory danych i ich przygotowanie	31
4.1.	Przegląd wykorzystanych zbiorów danych	31
4.2.	Proces przygotowania danych	31
4.2.1.	snoopysecurity/Vulnerable-Code-Snippets	31
4.2.2.	OWASP/NodeGoat	32
4.3.	Wyzwania i ograniczenia	32
4.4.	Podsumowanie	32
5.	Badania eksperymentalne	34
5.1.	Metodyka badań	34
5.1.1.	Procedura testowa	34
5.2.	Wyniki działania programu - przykład	35
5.2.1.	Opis przedstawionego wyniku	36
5.3.	Badania na zbiorze <i>snoopysecurity/Vulnerable-Code-Snippets</i>	37
5.4.	Studium przypadku: Analiza kodu podatnego na błędy typu "Out of Bounds" w zbiorze <i>snoopysecurity/Vulnerable-Code-Snippets</i>	38
5.4.1.	Dane wejściowe	38
5.4.2.	Wynik w formacie diff - może zmienić na wynik finalny, ponieważ nie ma kolorowych linii dla +/-	40
5.4.3.	Przygotowanie środowiska testowego	41
5.4.4.	Przeprowadzenie testów funkcjonalnych	41
5.4.5.	Interpretacja wyników	43
5.5.	Studium przypadku: Analiza kodu podatnego na błędy typu "File Inclusion" - skrawki kodu PHP, będące częścią aplikacji	44

6. Przeprowadzenie testów na aplikacji NodeGoat	45
6.1. Wstęp	45
6.2. Przygotowanie środowiska testowego	46
6.3. Testy funkcjonalne przed zmianami	46
6.4. Identyfikacja podatności za pomocą tradycyjnych skanerów	46
6.4.1. OWASP ZAP	47
6.4.2. Nessus	47
6.4.3. Statyczny analizator CodeQL	47
6.4.4. Wyniki otrzymane z analizy CodeQL	49
6.5. Analiza GPTester z wykorzystaniem RAG	49
6.6. Analiza GPTester bez RAG	50
6.7. Wprowadzanie zmian w kodzie	52
6.7.1. Wprowadzenie zmian za pomocą skryptu	52
6.7.2. Wprowadzenie zmian ręcznie	52
6.7.3. Testy funkcjonalne po wprowadzeniu zmian	55
6.7.4. Testy bezpieczeństwa po wprowadzeniu zmian	56
6.8. Identyfikacja podatności bez podpowiedzi w kodzie	58
6.9. Wnioski z analizy NodeGoat	60
Podsumowanie	60
Wnioski	61
Bibliografia	62
Spis listingów	63
Dodatki	64
A. Dodatek 1	65

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, jak skutecznie zaawansowane modele językowe, takie jak GPT-3.5, GPT-4 oraz w przyszłych iteracjach badania modele otwarto-źródłowe między innymi Mistral 7B i Falcon-7B-instruct, mogą być wykorzystane do automatycznego wykrywania i naprawy błędów bezpieczeństwa w kodzie zarówno aplikacji webowych jak i lokalnych.

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 tą pracą, a literaturą naukową polega na tym, że praca skupia się na praktycznym zastosowaniu modeli językowych w statycznej analizie kodu, 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?'.
— Projekt i implementację narzędzia do statycznej analizy kodu opartego na modelach OpenAI.
- Przygotowanie zbiorów danych i przykładów 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?"[1] <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"[2] <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 przesuwają 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. Wykorzystywane biblioteki: openai, asyncio, tiktoken, scipy, gitpython

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

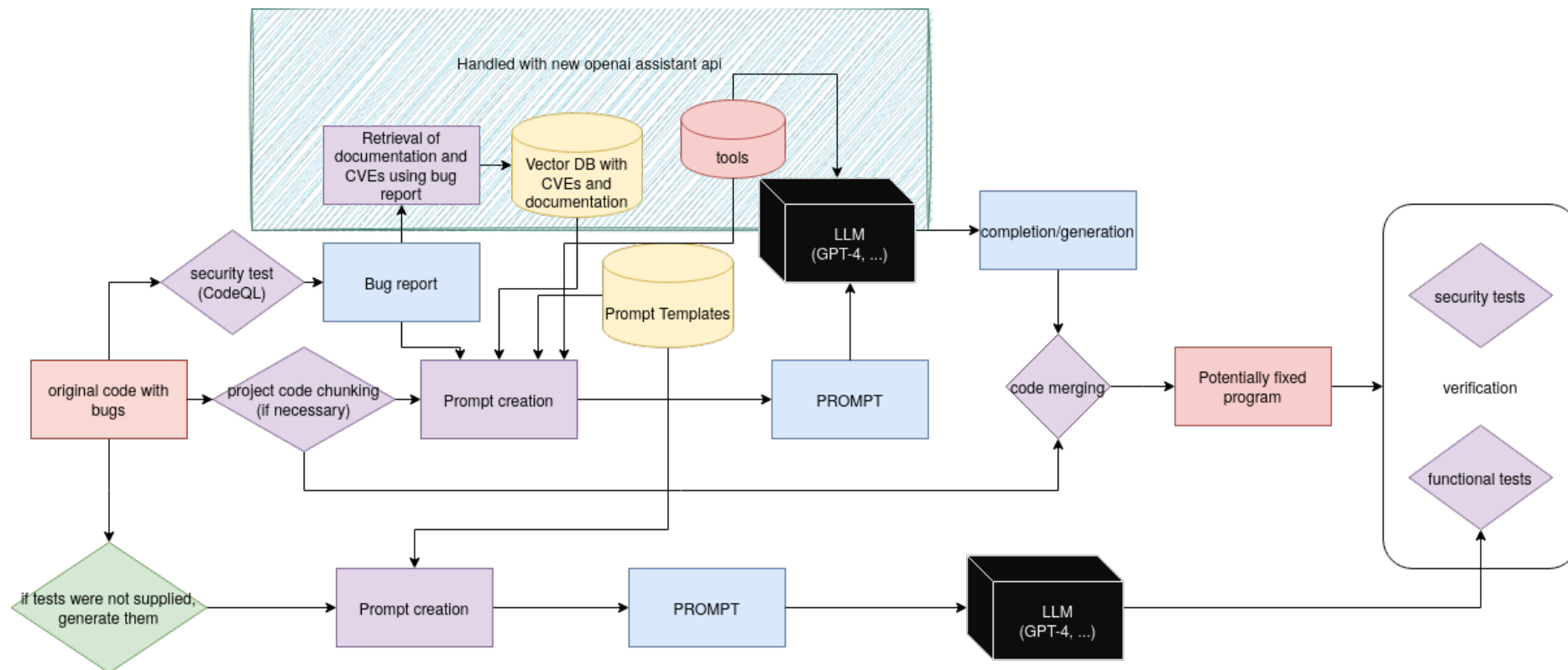
W ramach pracy inżynierskiej opracowano kompleksowe narzędzie **gptester**, które wykorzystuje zaawansowane modele językowe do analizy statycznej kodu. Narzędzie to wykorzystuje domyślnie model GPT-4 do generowania raportów na temat jakości kodu oraz proponowania napraw, ze szczególnym uwzględnieniem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

3.2. ARCHITEKTURA SYSTEMU

3.2.1. Ogólny opis

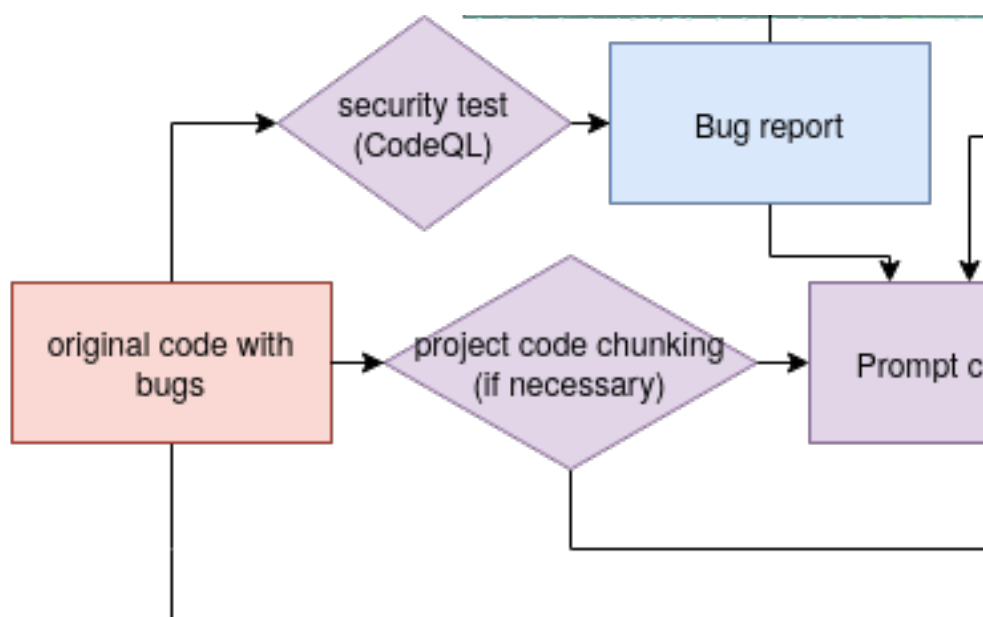
GPTester jest programem napisanym w języku Python, wykorzystującym model GPT-4 (lub GPT-3.5-turbo) dostarczony przez OpenAI. Jest zaprojektowany do uruchamiania z linii poleceń, a wyniki jego pracy są zapisywane w pliku formatu markdown oraz do osobnego katalogu z plikami wynikowymi zawierającymi poprawiony kod. Praktyczną funkcjonalność zapewnia wykorzystanie metod git, takich jak tworzenie i zapisywanie łatek (ang. patch), co umożliwia wygodne wprowadzanie poprawek do kodu oraz kontrolę wersji i łatwe scalanie kodu.

3.2.2. Schemat blokowy



Rys. 3.1: Schemat blokowy działania aplikacji *gptester*

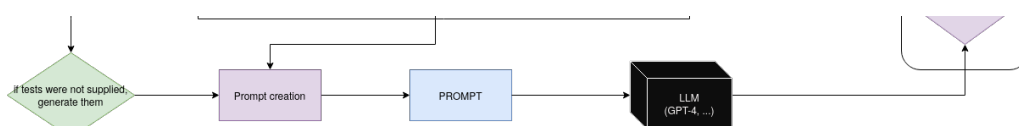
3.2.2.1. Dane wejściowe i wstępne przetwarzanie



Rys. 3.2: dane wejściowe w schemacie blokowym

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

3.2.2.2. Generacja testów funkcjonalnych

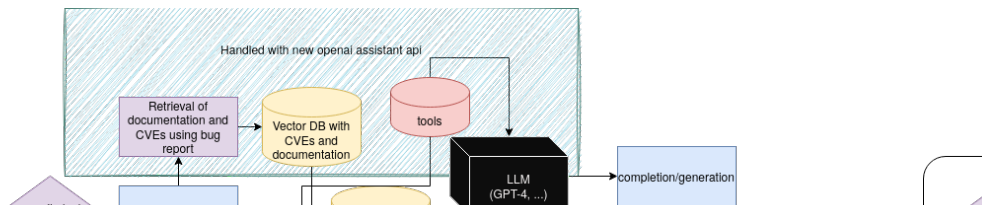


Rys. 3.3: Część schematu opisująca proces generatora testów funkcjonalnych

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 ich nie posiada. Problemem jest duża ilość kodu 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)

```

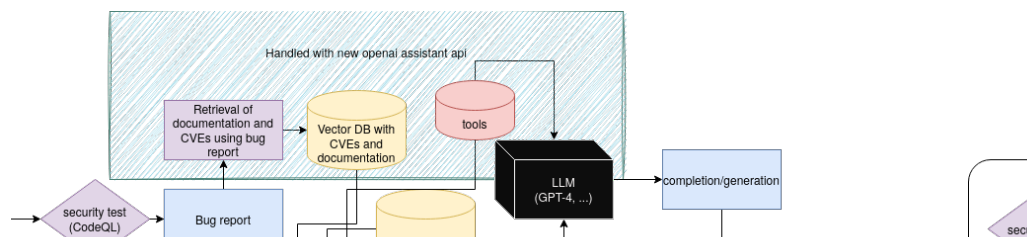
1 def relevance_for(self, query: str) -> float:
2     embedding = get_embedding(query)
3     task = get_embedding(self.name)
4     score = cosine(task, embedding)
5     return score

```

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

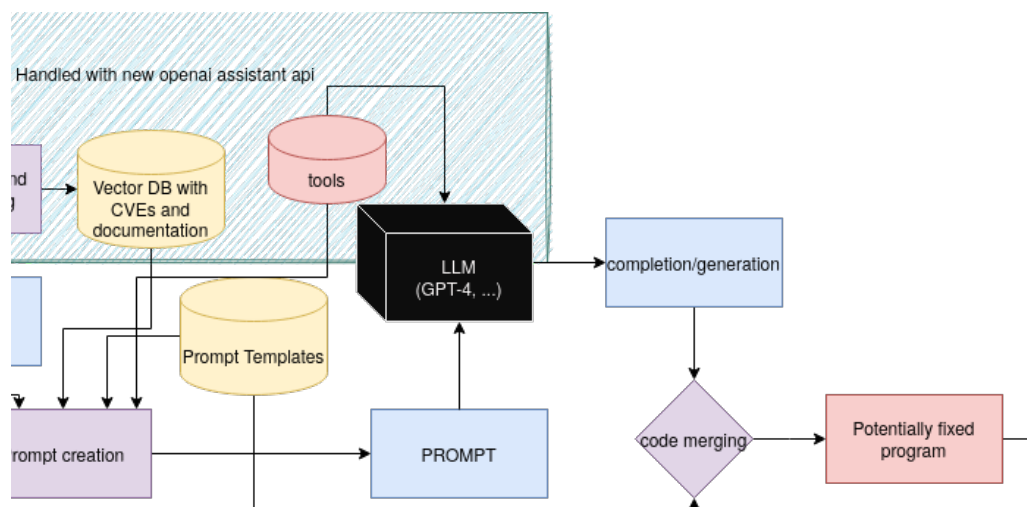


Rys. 3.5: Część schematu opisująca proces RAG

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

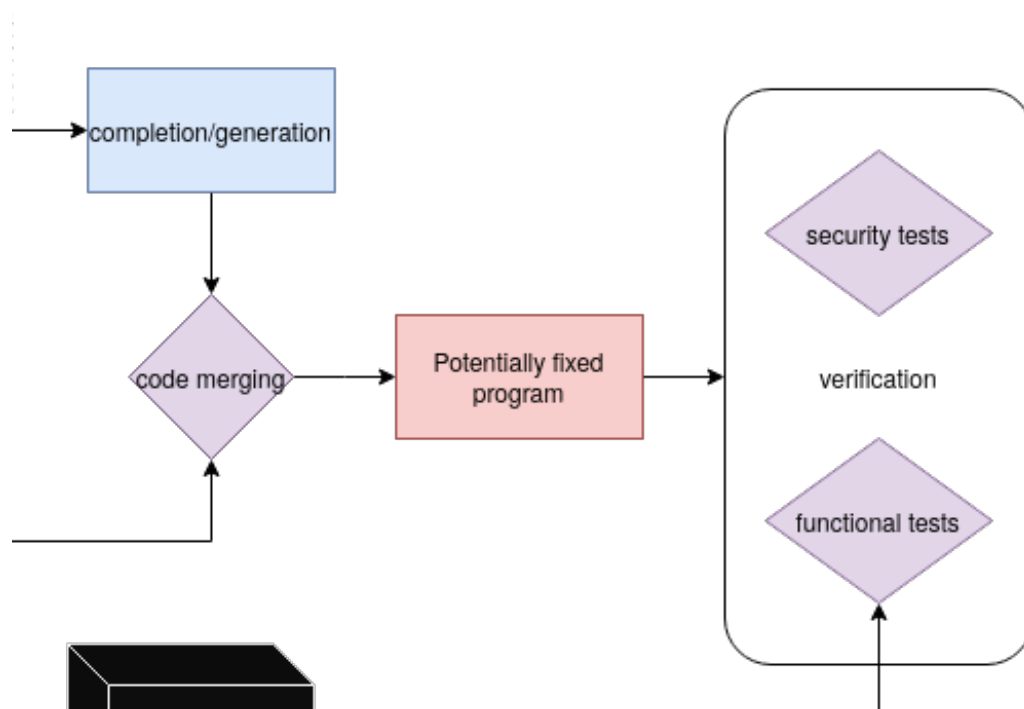


Rys. 3.6: Czarna skrzynka - LLM (Large Language Model)

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, running tests, ...), 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



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

poprawki 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
```

lub

```
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, korzystając z zaawansowanych modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa w kodzie. Wersja `assistant-0.5.2-beta` programu oferuje szereg funkcji, które są dostępne za pomocą argumentów linii poleceń, umożliwiając szeroką konfigurację i dostosowanie do specyficznych potrzeb analizy.

- **-h, -help**: Wyświetla pomoc programu, prezentując dostępne opcje i ich krótki opis, co ułatwia szybkie zrozumienie możliwości programu.
- **-v, -verbose**: Aktywuje tryb szczegółowych informacji, dzięki czemu gptester prezentuje dodatkowe informacje o każdym etapie przetwarzania, co jest przydatne do debugowania i głębszej analizy działania.
- **-m MODEL, -model MODEL**: Umożliwia wybór modelu językowego do analizy kodu, z domyślnym ustawieniem na "gpt-4-1106-preview". Pozwala na dostosowanie narzędzia do specyficznych wymagań projektu poprzez wybór innego dostępnego modelu.
- **-r, -retrieval**: Włącza Generację Wspomaganą Odnajdywaniem (Retrieval Augmented Generation - RAG) dla analizy kodu, zwiększając precyzję identyfikacji podatności przez odwołanie się do zewnętrznych baz wiedzy.
- **-f FILE_TO_KNOW, -file-to-know FILE_TO_KNOW**: Pozwala na wskazanie pliku, który ma być załączony do bazy wiedzy, co może pomóc w poprawie kontekstu analizy. Domyślnie ustawione na bazę danych CVE w formacie CSV.
- **-i IGNORE, -ignore IGNORE**: Umożliwia podanie ścieżki do pliku `.gitignore`, aby zignorować określone pliki podczas skanowania, co pomaga skupić się na kluczowych elementach kodu.
- **-p PATCH_FILE, -patch-file PATCH_FILE**: Określa ścieżkę dla wygenerowanego pliku z poprawkami, umożliwiając łatwe śledzenie i aplikowanie sugerowanych zmian w kodzie.
- **-o OUTPUT, -output OUTPUT**: Określa ścieżkę i nazwę pliku, do którego zostaną zapisane wyniki analizy. Pozwala na elastyczne zarządzanie dokumentacją wyników.
- **-t TESTS, -tests TESTS**: Wskazuje ścieżkę do testów funkcjonalnych, które mają być wykonane na projekcie. Ułatwia integrację z istniejącymi procedurami testowymi i pozwala na automatyczne generowanie testów, jeśli nie zostaną dostarczone.
- **-c, -codeql**: Włącza integrację z CodeQL, zaawansowanym narzędziem do analizy kodu. Wymaga zainstalowania narzędzia CodeQL-CLI i jest przeznaczone do wykrywania bardziej złożonych podatności.
- **-command COMMAND**: Umożliwia określenie polecenia budowania projektu, co jest niezbędne do prawidłowej integracji z CodeQL, szczególnie gdy w projekcie nie ma pliku `CMake` lub podobnego. Domyślnie ustawione na "make".

- **-language LANGUAGE**: Pozwala na określenie języka programowania projektu dla analizy CodeQL, z domyślnym ustawieniem na "cpp". Umożliwia dostosowanie analizy do różnych środowisk programistycznych.

```
[18:07:06] > ./main.py -h
usage: main.py [-h] [-v] [-m MODEL] [-r] [-f FILE_TO_KNOW] [-i IGNORE] [-p PATCH_FILE] [-o OUTPUT]
               [-t TESTS] [-c] [--command COMMAND] [--language LANGUAGE]
               directory

  (G)  (E)  (T)  (E)  (S)  (T)  (E)  (R)
  (E)  (R)  (E)  (S)  (T)  (E)  (R)

The static code analysis agent, version: assistant-0.5.2-beta

positional arguments:
  directory              Path to the directory to scan

options:
  -h, --help            show this help message and exit
  -v, --verbose         Print out all the outputs and steps taken
  -m MODEL, --model MODEL
                        Choose the LLM model for code analysis, default: "gpt-4-1106-preview"
  -r, --retrieval       Turn on Retrieval Augmented Generation for code analysis, default: False
  -f FILE_TO_KNOW, --file-to-know FILE_TO_KNOW
                        Point to a file to be uploaded to the knowledge base for retrieval
                        default: "699.csv" - CVE database in csv
                        !CURRENT VERSION SUPPORTS ONE FILE!
  -i IGNORE, --ignore IGNORE
                        Provide a path to a .gitignore file to ignore files from the scan,
                        a typical lists of files will be ignored by default
  -p PATCH_FILE, --patch-file PATCH_FILE
                        Provide a path for the generated patch file
                        default: "{your_project_dir}/GPTested/{timestamp}_patch.diff"
  -o OUTPUT, --output OUTPUT
                        Output the results to a specified directory
  -t TESTS, --tests TESTS
                        Provide a path to functional tests to run on the project
                        if you want to generate tests provide a prompt for the LLM like "generate"
                        !IN TESTING!NOT STABLE! Use codeql to enhance the scan
  -c, --codeql          REQUIRED to install CodeQL-CLI console tool
                        Provide a build command to run the project for codeql if no configuration file,
                        like cmake present in the project root directory, default: "make"
  --command COMMAND    Provide a build command to run the project for codeql if no configuration file,
                        like cmake present in the project root directory, default: "make"
  --language LANGUAGE  Provide a programming language of the project for codeql, default: "cpp"
```

Rys. 3.8: Wiadomość pomocnicza aplikacji *gptester*

Przykład użycia z pełną konfiguracją:

```
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"
```

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, 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 wdrożone zostaną także modele otwartoźródłowe, za pomocą narzędzia Ollama. Dostępne będą między innymi: Llama2, GPT-J, Mistral, Falcon, czy jakiegokolwiek 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:

ZAMIENŃ NA AKTUALNE POLECENIE

```
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` 3.3. 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. Choć 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:

```
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 )
```

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 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, czy parametry generowania komplecji, 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:

- **Instrukcje (Rola/Prompt Systemowy):** 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 konfiguracyjnych komplekcy, czy generacji, dlatego nie można przekazać parametrów takich jak ‘temperature’ czy ‘max_tokens’. W przyszłości, jeśli API Asystentów zostanie rozwinięte, będzie można przekazać te parametry i zbadać ich wpływ.

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.
- **Automatyzacja testów:** 'gptester' automatyzuje proces testowania kodu, w tym pisanie testów i ich wykonywanie, co pozwala na efektywne sprawdzanie poprawności kodu.
- **Aktualizacja kodu za pomocą funkcji git patch i plików diff:** Funkcjonalność, która pozwala na automatyczne wprowadzanie poprawek do kodu źródłowego na podstawie wygenerowanych plików diff. To ulepszenie ułatwia proces naprawy kodu, umożliwiając automatyczne aplikowanie poprawek oraz interaktywne wybieranie elementów z obu wersji za pomocą systemu kontroli wersji git.

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:

- **Wsparcie dla modeli otwartoźródłowych:** Wdrożenie modeli otwartoźródłowych, takich jak GPT-J, Falcon, Mistral, czy Llama2, co pozwoli na jeszcze większą elastyczność i dostosowanie do różnych zastosowań.

- **Rozbudowa i stabilizacja modułu testowania:** 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 większej ilości 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 sprawiają, 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/NodeGoat**: Repozytorium zawierające aplikację webową, która została stworzona w celu demonstracji podatności z TOP 10 OWASP. Aplikacja została napisana w NodeJS, a jej kod źródłowy jest dostępny w serwisie Github. <https://github.com/OWASP/NodeGoat>

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/NodeGoat

NodeGoat, jako aplikacja demonstrująca podatności OWASP Top 10, stanowiła cenne źródło do testowania 'gptester'. Dzięki dokumentacji i instrukcjom dostępnym w repozytorium, możliwe było efektywne wykorzystanie tej aplikacji do celów badawczych.

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 kontekście OWASP NodeGoat, aplikacja ta była w pełni funkcjonalna i umożliwiała realistyczne testowanie podatności. Jednakże, wyzwaniem okazało się wykorzystanie wcześniejszej wersji gptester, która nie posiadała jeszcze możliwości aktualizacji bazy kodu przez systemy kontroli wersji. To z kolei wymuszało ręczne łączenie zmian w kodzie z nowymi wersjami aplikacji, co było procesem czasochłonnym i komplikowało przeprowadzenie badań. Testy bezpieczeństwa przeprowadzone przy użyciu skanerów podatności, zwłaszcza OWASP ZAP - zoptymalizowanego do wykrywania podatności w aplikacjach takich jak NodeGoat - miały za zadanie ocenić skuteczność proponowanych przez gptester korekt.

4.4. PODSUMOWANIE

W rozdziale 4 przedstawiono i dokładnie przeanalizowano zbiory danych wykorzystane w badaniach nad skutecznością dużych modeli językowych (LLM) za pomocą programu **gptester**. Kluczowe znaczenie miało tutaj szczegółowe przygotowanie i przetwarzanie tych zbiorów, co miało bezpośredni wpływ na efektywność analizy statycznej kodu oraz kalibrację narzędzia.

Repozytorium *snoopysecurity/Vulnerable-Code-Snippets* dostarczyło przykłady kodu zawierające potencjalne luki bezpieczeństwa, które umożliwiły testowanie zdolności **gptester** do identyfikacji i sugerowania napraw błędów bezpieczeństwa. Z kolei aplikacja *OWASP/NodeGoat*, demonstrująca podatności z listy OWASP Top 10, posłużyła jako realistyczne środowisko do testowania skuteczności LLM w wykrywaniu podatności.

Wyzwaniem okazała się natura wykorzystanych próbek kodu, które często nie były reprezentatywne dla rzeczywistych aplikacji i wymagały dodatkowej pracy, aby mogły być użyte w testach. Wczesna wersja **gptester** również nałożyła ograniczenia na proces badawczy, szczególnie w kontekście aktualizacji bazy kodu i ręcznego scalania zmian.

Pomimo tych wyzwań, analiza zbiorów danych pozwoliła na zgłębienie możliwości i ograniczeń LLM w kontekście analizy bezpieczeństwa aplikacji webowych. Przeprowadzone badania podkreśliły znaczenie interdyscyplinarnego podejścia łączącego technologie LLM z wiedzą ekspercką w dziedzinie bezpieczeństwa, aby maksymalizować efektywność narzędzi typu **gptester** w wykrywaniu i naprawianiu podatności w oprogramowaniu.

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 umożliwiającą 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 analizy¹

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 OOBRead_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 popraw 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:

```

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

```

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 +/-

```
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     if(fp == NULL){
8         printf("\nCan't open file or file doesn't exist.");
9         exit(0);
10    }
11    -    fprintf("\n\tHeader\twidth\theight\tdata\t\r\n");
12    +    printf("\n\tHeader\twidth\theight\tdata\n");
13
14    +    if (sizeof(struct Image) == 0) return 0; // Prevent division by zero in fread
15    +
16    while(fread(&img,sizeof(img),1,fp)>0){
17        printf("\n\t%s\t%d\t%d\t%s\r\n",img.header,img.width,img.height,img.data);
18        int size1 = img.width + img.height; //Vulnerability: integer overflow
19    +    if (img.width > INT_MAX - img.height) continue; // Prevent integer overflow
20        char* buff1=(char*)malloc(size1);
21        memcpy(buff1,img.data,sizeof(img.data)); //Vulnerability: no data buffer size/malloc success check?
22    @@ -31,16 +33,15 @@ int ProcessImage(char* filename){
23        free(buff1); //Vulnerability: double free
24    }
25    else{
26        if(size1 == 123456){
27            buff1[0]='a'; //Vulnerability: use after free
28        }
29    }
30    -    int size2 = img.width - img.height+100; //Vulnerability: integer underflow
31    -    //printf("Size1:%d",size1);
32    +    // Removed potential integer underflow by ensuring img.width is always greater than img.height before
33    ↪ subtraction
34    +    int size2 = img.height > img.width ? 100 : img.width - img.height + 100;
35    char* buff2=(char*)malloc(size2);
36    memcpy(buff2,img.data,sizeof(img.data));
37    int size3= img.width/img.height;
38    -    //printf("Size2:%d",size3);
39    +    // Check for division by zero
40    +    if (img.height == 0) continue;
41    char buff3[100];
42    -    char* buff4 =(char*)malloc(size3);
43    -    memcpy(buff4,img.data,sizeof(img.data));
44    +    // Check if size3 is within buffer size
45    +    if (size3 < 0 || size3 >= sizeof(buff3)) continue;
46    char OOB_stack = buff3[size3+100]; //Vulnerability: out of bound read (stack)
47    char OOB_heap = buff4[100];
48    +    char* buff4 = NULL;
49    +    if (size3 < sizeof(buff3)) {
50    +        buff4 = (char*)malloc(size3);
51    +        if (!buff4) continue; // Check malloc success
52    +        memcpy(buff4,img.data,sizeof(img.data));
53    -    buff3[size3+100]='c'; //Vulnerability: out of bound write (Stack)
54    +    // Prevent out of bound write to buff3
55    +    if (size3 + 100 < sizeof(buff3)) buff3[size3+100] = 'c';
56    -    buff4[100]='c'; //Vulnerability: out of bound write (Heap)
57    +    // Prevent out of bound write to buff4 and potential memory leak
58    +    if (buff4 && size3 >= 0 && size3 < 100) buff4[size3] = 'c';
59    if(size3>10){
60    -        buff4=0; //memory leak?
61    +        if (buff4) { free(buff4); buff4 = NULL; } // Properly free memory
62    } else{
63        free(buff4);
64    }
65    free(buff2);
66    @@ -50,8 +51,8 @@ int ProcessImage(char* filename){
67    if(size3>10){
68        buff4=0; //memory leak?
69    -
70    }
71    fclose(fp);
72    }
```

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 5.2 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 skrypcie `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 skryptu `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 eksploatacji 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:

Tabela 5.1: Vulnerable Code Snippets – oryginał – możliwe odpowiedzi w komentarzach

lp.	nazwa podatności	wykryto	max(subiektywne)	funkcjonalność	efektywność	bezpieczeństwo
17	Out of Bounds	11	9	1	0.5	1

5.5. STUDIUM PRZYPADKU: ANALIZA KODU PODATNEGO NA BŁĘDY TYPU "FILE INCLUSION" - SKRAWKI KODU PHP, BĘDĄCE CZĘŚCIĄ APLIKACJI

Większość przykładów z tego zbioru to skrawki aplikacji webowych, dlatego przygotowanie środowiska testowego dla tych przykładów było bardziej skomplikowane.

6. PRZEPROWADZENIE TESTÓW NA APLIKACJI NODEGOAT

6.1. WSTĘP

W ramach niniejszego rozdziału zostanie omówiona aplikacja NodeGoat, służąca do edukacji w zakresie zabezpieczeń aplikacji internetowych opartych na platformie Node.js. NodeGoat, będąc projektowaną z myślą o nauce i praktyce, oferuje środowisko, w którym użytkownicy mogą eksplorować i naprawiać najczęściej występujące luki bezpieczeństwa określone w ramach OWASP Top 10 dla aplikacji webowych.

Node.js, jako lekka, szybka i skalowalna platforma, zyskuje coraz większą popularność wśród deweloperów aplikacji internetowych. Z tego względu narzędzia takie jak NodeGoat odgrywają kluczową rolę w edukacji z zakresu bezpieczeństwa, umożliwiając użytkownikom praktyczne zrozumienie zagrożeń i metod ich przeciwdziałania.

Zrozumienie OWASP Top 10

NodeGoat zapewnia kompleksowy tutorial, który opisuje poszczególne zagrożenia z listy OWASP Top 10, demonstrując je na przykładzie działającej aplikacji. Użytkownicy mają możliwość nie tylko zapoznania się z teorią, ale również praktycznego przetestowania każdego z zagrożeń, co stanowi cenny element edukacyjny.

Praktyczne testowanie zabezpieczeń

Aplikacja NodeGoat została zaprojektowana z celowymi słabościami bezpieczeństwa, dzięki czemu użytkownicy mogą w bezpiecznym środowisku przeprowadzać ataki i testować różne techniki obronne. Dostęp do aplikacji jest możliwy lokalnie, co umożliwia dogłębne badanie kodu oraz modyfikowanie go w celu naprawy istniejących luk.

Konta użytkowników

Domyślnie, baza danych aplikacji zawiera predefiniowane konta użytkowników, w tym konto administratora oraz kilka kont użytkowników zwykłych. Umożliwia to szybkie rozpoczęcie testowania aplikacji bez konieczności ręcznego konfigurowania środowiska.

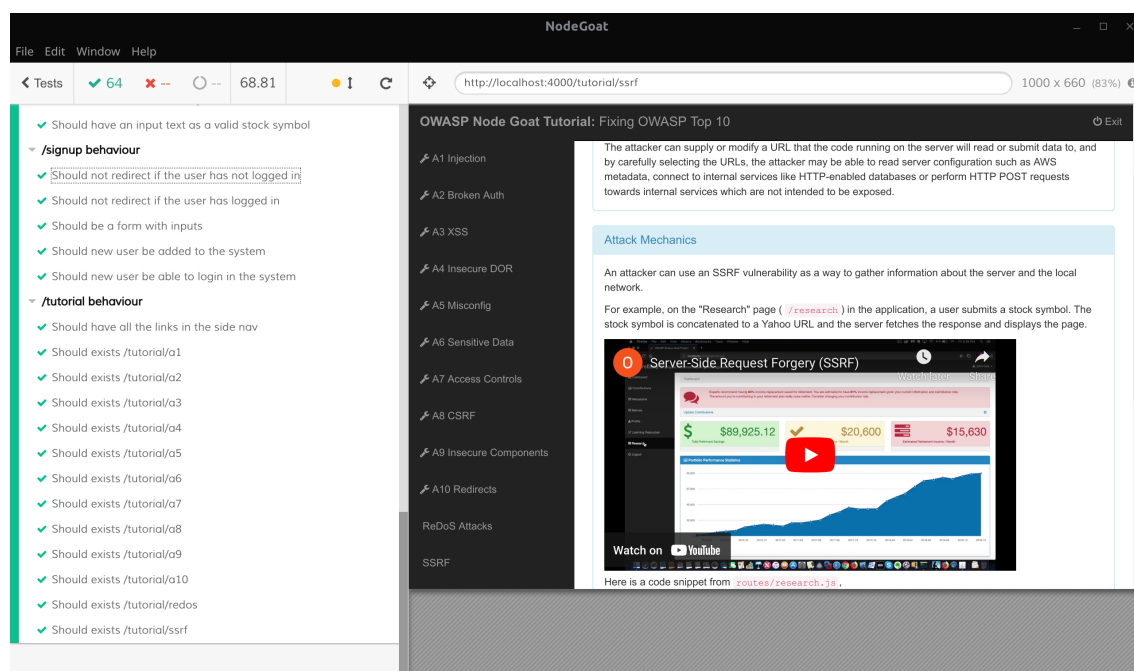
W niniejszym rozdziale szczegółowo przedstawione zostaną metody testowania aplikacji NodeGoat, wykorzystując do analizy duże modele językowe. Przeprowadzone badania

mają na celu nie tylko identyfikację istniejących słabości, ale również ocenę skuteczności zastosowanych metod naprawczych.

6.2. PRZYGOTOWANIE ŚRODOWISKA TESTOWEGO

Aby przygotować aplikację do lokalnego uruchomienia bez korzystania z konteneryzacji, niezbędne było poprawienie wersji paczek w pliku package.json oraz obniżenie używanej wersji npm i node do node v16.0.0 (npm v7.10.0). Następnie należało zainstalować wszystkie zależności za pomocą komendy npm install. Po wykonaniu tych kroków aplikacja była gotowa do uruchomienia komendą npm run.

6.3. TESTY FUNKCJONALNE PRZED ZMIANAMI



Rys. 6.1: Wyniki testów funkcjonalnych aplikacji NodeGoat przed wprowadzeniem zmian

6.4. IDENTYFIKACJA PODATNOŚCI ZA POMOCĄ TRADYCYJNYCH SKANERÓW

Przed przystąpieniem do naprawy podatności, aplikacja DVNA została poddana skanowaniu przy użyciu skanerów OWASP ZAP i Nessus. Wyniki tych skanowań zostały wykorzystane jako punkt odniesienia do oceny skuteczności narzędzia gptester w kontekście naprawy podatności.

6.4.1. OWASP ZAP

Skanowanie aplikacji NodeGoat przy użyciu OWASP ZAP wykazało obecność 20 podatności, w tym 3 krytyczne, 5 wysokiego ryzyka, 5 średniego ryzyka oraz 7 niskiego ryzyka. Poniżej przedstawiono listę zidentyfikowanych podatności wraz z ich ryzykiem i opisem.

Tabela 6.1: Wyniki skanowania aplikacji NodeGoat przy użyciu skanera OWASP-zap

		Risk			
		High (= High)	Medium (>= Medium)	Low (>= Low)	Informational (>= Informational)
site	http://localhost:4000	3 (3)	5 (8)	5 (13)	7 (20)

Tabela 6.2: Podsumowanie typów alertów i powiązanych zagrożeń

Alert Type	Risk	Count (%)
Cross Site Scripting (Reflected)	High	3 (15.0%)
External Redirect	High	1 (5.0%)
Open Redirect	High	1 (5.0%)
CSP: Wildcard Directive	Medium	16 (80.0%)
Content Security Policy (CSP) Header Not Set	Medium	24 (120.0%)
Directory Browsing	Medium	2 (10.0%)
Missing Anti-clickjacking Header	Medium	24 (120.0%)
Vulnerable JS Library	Medium	2 (10.0%)
Cookie without SameSite Attribute	Low	5 (25.0%)
Cross-Domain JavaScript Source File Inclusion	Low	24 (120.0%)
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Low	63 (315.0%)
Timestamp Disclosure - Unix	Low	1 (5.0%)
X-Content-Type-Options Header Missing	Low	37 (185.0%)
Authentication Request Identified	Informational	1 (5.0%)
Information Disclosure - Suspicious Comments	Informational	26 (130.0%)
Loosely Scoped Cookie	Informational	5 (25.0%)
Modern Web Application	Informational	23 (115.0%)
Session Management Response Identified	Informational	8 (40.0%)
User Agent Fuzzer	Informational	290 (1,450.0%)
User Controllable HTML Element Attribute (Potential XSS)	Informational	2 (10.0%)
Total		Various

6.4.2. Nessus

Skanowanie aplikacji DVNA przy użyciu Nessus wykazało obecność 21 podatności, w tym 2 średniego ryzyka, 2 niskiego ryzyka oraz 17 informacyjnych. Poniżej przedstawiono listę zidentyfikowanych podatności wraz z ich ryzykiem i opisem.

6.4.3. Statyczny analizator CodeQL

6.4.3.1. Proces przeprowadzenia testu

1. Przygotowanie środowiska testowego dla aplikacji NodeGoat.

2. Uruchomienie skanowania z wykorzystaniem programu. Przed przystąpieniem do skanowania należy utworzyć bazę danych CodeQL. W tym celu należy wykonać następujące kroki:

- a) Zainstalować CodeQL CLI.
- b) Zainicjalizować i zbudować bazę danych CodeQL.

```
> codeql database create GoatQL-db --language=javascript
Initializing database at /home/paris/projekty/INZYNIERKA/testing-envs/NodeGoat/GoatQL-db.
Running build command: []
[2024-01-25 19:33:30] [build-stdout] Single-threaded extraction.
[2024-01-25 19:33:34] [build-stdout] package.json: Main file set to server.js
[2024-01-25 19:33:34] [build-stdout] Extracting .../NodeGoat/app/views/tutorial/a1.html
[2024-01-25 19:33:34] [build-stdout] Done extracting .../NodeGoat/app/views/tutorial/a1.html (91 ms)
[2024-01-25 19:33:34] [build-stdout] Extracting .../NodeGoat/app/views/tutorial/a10.html
[2024-01-25 19:33:34] [build-stdout] Done extracting .../NodeGoat/app/views/tutorial/a10.html (5 ms)
[2024-01-25 19:33:34] [build-stdout] Extracting .../NodeGoat/app/views/tutorial/a2.html
.
.
.
[2024-01-30 19:33:38] [build-stdout] Extracting .../codeql/javascript/tools/data/externs/nodejs/vm.js
[2024-01-30 19:33:38] [build-stdout] Done extracting .../codeql/javascript/tools/data/externs/nodejs/vm.js (8 ms)
Finalizing database at /home/paris/projekty/INZYNIERKA/testing-envs/NodeGoat/GoatQL-db.
Successfully created database at /home/paris/projekty/INZYNIERKA/testing-envs/NodeGoat/GoatQL-db.
```

- c) Uruchomić wybrane zapytania na bazie danych wyszukujące podatności. W tym celu uruchomiłem domyślne pakiety zapytań dostępne w CodeQL CLI. W celu uruchomienia zapytania należy wykonać następującą komendę:

```
> codeql database analyze GoatQL-db javascript-code-scanning.qls --format=csv --output=default-goat-QLresults.csv
Running queries.
[1/88] Loaded .../codeql/javascript-queries/0.8.2/AngularJS/InsecureUrlWhitelist.qlx.
```

Komenda zakończyła się sukcesem, a w katalogu pojawił się plik z wynikami analizy.

```
PrototypePollutingFunction.ql : [83/88 eval 25ms] Results written to codeql/javascript-queri
PrototypePollutingMergeCall.ql : [84/88 eval 260ms] Results written to codeql/javascript-quer
InsufficientPasswordHash.ql : [85/88 eval 5ms] Results written to codeql/javascript-querie
RequestForgery.ql : [86/88 eval 143ms] Results written to codeql/javascript-quer
LinesOfCode.ql : [87/88 eval 3ms] Results written to codeql/javascript-querie
LinesOfUserCode.ql : [88/88 eval 644ms] Results written to codeql/javascript-quer
Shutting down query evaluator.
Interpreting results.
Analysis produced the following diagnostic data:
```

Diagnostic	Summary
+-----+	
Successfully extracted files	77 results

Analysis produced the following metric data:

Metric	Value
+-----+	
Total lines of user written JavaScript and TypeScript code in the database	2926

Na aplikacji został również uruchomiony pakiet kwerend `javascript-extended-security.qls`

3. Dokumentacja wykrytych podatności oraz sugerowanych przez program napraw.
4. Analiza efektywności napraw i ich wpływ na bezpieczeństwo aplikacji za pomocą innych skanerów podatności.

6.4.4. Wyniki otrzymane z analizy CodeQL

Tabela 6.3: Podsumowanie wyników analizy CodeQL

Issue	Description	File	Occurrences
Inefficient regular expression	Can cause exponential backtracking	profile.js	1
Polynomial regex on uncontrolled data	May be vulnerable to DoS attacks	profile.js, session.js	2
Database query from user input	Risk of malicious code insertion	user-dao.js	2
Clear text sensitive cookie transmission	Sensitive cookies sent without SSL	server.js	1
Missing CSRF middleware	Lack of CSRF protection risks malicious requests	server.js	Multiple
Server-side URL redirect	May redirect to malicious websites	index.js	1
Code injection	Allows arbitrary code execution by a user	contributions.js	Multiple
Missing rate limiting	Vulnerable to DoS attacks	index.js	1
Server-side request forgery	User-controlled data in URL risks forgery attacks	research.js	1
DOM text as HTML	Leads to XSS vulnerability	bootstrap.js	Multiple
Unsafe jQuery plugin	Constructs HTML unsafely, risking XSS	bootstrap.js	Multiple

Rozszerzony pakiet kwerend rozszerzył wyniki o następujące wpisy:

Tabela 6.4: Extended Scanning Results with CodeQL

Issue	Description	File	Details
Failure to abandon session	Reuse of session can lead to account access	index.js	Session not invalidated post-login
Log injection	Forged log entries risk from user inputs	session.js	Log entry built from user input
Indirect command line in shell	May introduce command-line injection vulnerabilities	Gruntfile.js	Command depends on unsanitized env variable

6.5. ANALIZA GPTESTER Z WYKORZYSTANIEM RAG

Z uwagi na wielkość projektu i limit dla API asystentów wynoszący 32768 znaków (nie tokenów), kod był przetwarzany przez aplikację w aż 50 częściach asynchronicznie. Wykorzystana została dodatkowo funkcja generacji wspomaganej pobieraniem danych.

Niestety firmowy klucz API OpenAI, z którego korzystałem by wykonać skan został pod sam koniec wycofany, prawdopodobnie przez dużą liczbę zapytań. W związku z tym, udało się przeprowadzić skanowanie tylko dla 45 części kodu, co stanowiło około 9/10 kodu aplikacji. Skanowanych było o 10 plików więcej niż uwzględniał CodeQL, co wynikało z faktu, że CodeQL nie uwzględniał plików konfiguracyjnych i tych które zawierały kod HTML. Pozwoliło to agentowi na aktualizację wersji oprogramowania w plikach konfiguracyjnych.

Ze względu na zwiększone koszty związane z wysyłaniem tylu zapytań do OpenAI i ograniczenia czasowe zoptymalizowałem kod tak, aby skanowanie odbywało się ignorując pliki nie zawierające kodu. Tę konfigurację można edytować z pomocą pliku `config.py`.

6.6. ANALIZA GPTESTER BEZ RAG

Następnie uruchomiłem proces analizy kodu za pomocą GPTester bez użycia RAG. W tym celu wykorzystałem prywatny klucz API OpenAI, dzięki czemu mogłem dowiedzieć się jakie koszty są generowane podczas skanowania. Zoptymalizowałem także wybieranie plików do skanowania, aby ograniczyć koszty. W sumie GPTester znalazł 21 podatności, bez określania ich ryzyka. Wśród nich były podatności takie jak:

1. **Hardcoded Sensitive Information:** The `db-reset.js` file contains hardcoded sensitive information such as usernames and passwords that should not be stored in plain text. This is a security risk as it can be exploited if the code is exposed.
 2. **Insecure Randomness:** The `db-reset.js` file generates random allocation percentages using `Math.random()`, which is not cryptographically secure and thus not suitable for security-sensitive contexts.
 3. **Unvalidated Redirects and Forwards:** The `index.js` file uses query parameter `url` to redirect the user without validation in the `/learn` route. This can lead to phishing attacks or redirection to malicious sites.
 4. **Insecure Direct Object References (IDOR):** The `allocations.js` file uses `req.params.userId` directly from the URL, which allows users to access other users' allocation data without proper authorization.
 5. **Server-Side JavaScript Injection (SSJI):** The `contributions.js` file uses an insecure `eval()` function to parse user inputs, which could allow an attacker to run arbitrary code on the server.
 6. **Cross-Site Scripting (XSS):** The `profile.js` file attempts to encode user-supplied input with `ESAPI.encoder().encodeForHTML(doc.website)` which could lead to XSS attacks if not properly encoded for the context in which it is rendered.
 7. **Regular Expression Denial of Service (ReDoS):** The `profile.js` file uses a vulnerable regular expression that may cause the application to freeze with specially crafted input.
1. Log Injection Vulnerability - In the `handleLoginRequest` method in `session.js`, user input (`userName`) is being logged without proper sanitization, which can be exploited by an attacker to forge logs or execute code.
 2. Insecure Password Storage - In `user-dao.js`, the `addUser` method stores passwords as plain text, which is insecure as it can be easily read if the database is compromised.

3. Session Fixation - In `session.js`, the method `handleLoginRequest` doesn't regenerate the session ID after a successful login, making the application vulnerable to session fixation attacks.
4. NoSQL Injection - The method `getByIdAndThreshold` in `allocations-dao.js` constructs a query using string concatenation without properly sanitizing the `threshold` parameter, making it susceptible to NoSQL injection attacks.
5. Insufficient Password Strength Validation - The `validateSignup` method in `session.js` uses a regular expression for password validation, but the commented out regex suggests that a stronger password policy was considered and not implemented.
6. Sensitive Data Exposure - In `profile-dao.js`, sensitive information like `ssn` and `dob` are stored without encryption, which makes them vulnerable to exposure if there is a data breach.
1. Insecure Database Connection (`../testing-envs/NodeGoat/server.js`) - The `MongoClient.connect` call uses a connection string that might not be secure. The code for setting up the database connection does not include SSL/TLS configuration options which are important for secure transmission of data over the network.
2. Server-Side Template Injection (`../testing-envs/NodeGoat/server.js`) - The template engine `Swig` is used without `autoescaping` enabled, this could lead to Server-Side Template Injection (SSTI) if the template rendering includes user input.
3. HTTP Connection (`../testing-envs/NodeGoat/server.js`) - The server is started with an insecure HTTP connection instead of HTTPS, which exposes the transmitted data to potential interception and tampering.
4. Insufficient Session Management (`../testing-envs/NodeGoat/server.js`) - The session is created with `saveUninitialized: true`, which can lead to the generation of session identifiers even when the session is not actually started for the user.
5. Lack of Cookie Security (`../testing-envs/NodeGoat/server.js`) - No secure flags for cookies are set like `httpOnly` and `secure`. This can make the session cookies vulnerable to XSS attacks and interception over insecure connections.
6. Lack of Security Headers (`../testing-envs/NodeGoat/server.js`) - Security headers such as Content Security Policy (CSP), X-Frame-Options, X-XSS-Protection, and others are commented out. This could lead to various attacks such as clickjacking, XSS, etc.
7. Insecure Static File Permissions (`../testing-envs/NodeGoat/docker-compose.yml`)
 - The Dockerfile setup for the NodeGoat container runs commands as the root user, which can be a security risk if the container is compromised.
8. Incomplete Logging Configuration (`../testing-envs/NodeGoat/Gruntfile.js`)
 - The Grunt configuration does not include tasks for security logging or monitoring, which can leave the application without proper tracking of security-related events.

Wyniki zostały przedstawione w częściach, tak jak były przetwarzane. Dla klarowności

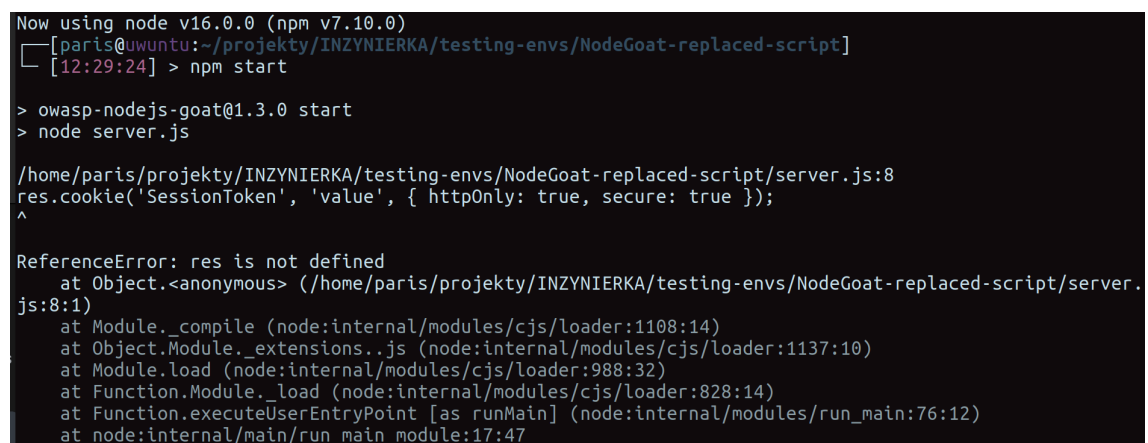
przedstawionych wyników funkcjonalność wyboru formatu wyjścia będzie rozwijana w przyszłych wersjach aplikacji.

6.7. WPROWADZANIE ZMIAN W KODZIE

Po przeprowadzeniu analiz kodu, należało wprowadzić zmiany w kodzie, które miały naprawić wykryte podatności. W tej wersji program dostarczył 38 nowych plików. W nowych wersjach ilość analizowanych plików została zoptymalizowana ze względu na koszty.

6.7.1. Wprowadzenie zmian za pomocą skryptu

Z uwagi na ciągle dopracowywanie funkcji aktualizacji kodu za pomocą funkcji git, napisano prosty skrypt bash (`gptester/Utils/replace_files.sh`), który zamieniał pliki o tych samych nazwach. Niestety plików z tymi samymi nazwami jest kilka w oryginalnym projekcie co doprowadziło do wielokrotnej zamiany i błędów uruchomienia aplikacji. Wiele z otrzymanych plików miało jedynie część kodu, co uniemożliwiło uruchomienie aplikacji.



```
Now using node v16.0.0 (npm v7.10.0)
[paris@uwuntu:~/projekty/INZYNIERKA/testing-envs/NodeGoat-replaced-script]
[12:29:24] > npm start

> owasp-nodejs-goat@1.3.0 start
> node server.js

/home/paris/projekty/INZYNIERKA/testing-envs/NodeGoat-replaced-script/server.js:8
res.cookie('SessionToken', 'value', { httpOnly: true, secure: true });
^
ReferenceError: res is not defined
    at Object.<anonymous> (/home/paris/projekty/INZYNIERKA/testing-envs/NodeGoat-replaced-script/server.js:8:1)
    at Module._compile (node:internal/modules/cjs/loader:1108:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1137:10)
    at Module.load (node:internal/modules/cjs/loader:988:32)
    at Function.Module._load (node:internal/modules/cjs/loader:828:14)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:76:12)
    at node:internal/main/run_main_module:17:47
```

Rys. 6.2: Błąd uruchomienia aplikacji NodeGoat po wprowadzeniu zmian

Przykład pokazuje jak ważne jest, aby zawsze sprawdzić wyjście wygenerowane przez Sztuczną Inteligencję, ponieważ są to algorytmy niedeterministyczne, co oznacza, że nie można przewidzieć dokładnie jakie będzie wyjście programu dla danego wejścia. W tym przypadku, program wygenerował kod, który nie był w pełni funkcjonalny, co uniemożliwiło uruchomienie aplikacji. Sprytne metody wprowadzania zmian or

6.7.2. Wprowadzenie zmian ręcznie

Aby uniknąć konfliktów i zamieniania plików nie posiadających pełnego kodu, zdecydowałem się na ręczne wprowadzenie zmian. W tym celu skopiowałem pliki z nowej wersji do katalogu z aplikacją NodeGoat, uważnie sprawdzając ich treść i ręcznie łącząc

kod. Niestety wiele zmian było gruntownych wymagających generowania certyfikatów, implementacji systemów hashujących i tym podobnych co utrudnia wprowadzenie zmian w krótkim czasie.

Następnie uruchomiłem aplikację i sprawdziłem czy działa poprawnie. Po uruchomieniu aplikacji, przeprowadziłem testy funkcjonalne, które potwierdziły poprawność działania aplikacji.

```
Now using node v16.0.0 (npm v7.10.0)
[paris@uwuntu:~/projekty/INZYNIERKA/testing-envs/NodeGoat-replaced-script]
[12:29:24] > npm start

> owasp-nodejs-goat@1.3.0 start
> node server.js

/home/paris/projekty/INZYNIERKA/testing-envs/NodeGoat-replaced-script/server.js:8
res.cookie('SessionToken', 'value', { httpOnly: true, secure: true });
^
ReferenceError: res is not defined
    at Object.<anonymous> (/home/paris/projekty/INZYNIERKA/testing-envs/NodeGoat-replaced-script/server.js:8:1)
    at Module._compile (node:internal/modules/cjs/loader:1108:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1137:10)
    at Module.load (node:internal/modules/cjs/loader:988:32)
    at Function.Module._load (node:internal/modules/cjs/loader:828:14)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:76:12)
    at node:internal/main/run_main_module:17:47
```

Rys. 6.3: Błąd uruchomienia aplikacji NodeGoat po wprowadzeniu zmian ręcznie

Błąd wystąpił w pliku, w którym błędnie złączyłem kod. Do naprawy użyłem wersji przeglądarkowej ChatGPT, która znalazła błąd i odpowiedziała jak go naprawić. Po naprawie błędu, należało skonfigurować bazę danych MongoDB, tak by używała certyfikatów.

```

> owasp-nodejs-goat@1.3.0 start
> node server.js

Current Config:
{
  port: 4000,
  db: 'mongodb://localhost:27017/nodegoat',
  cookieSecret: 'session_cookie_secret_key_here',
  cryptoKey: 'a_secure_key_for_crypto_here',
  cryptoAlgo: 'aes256',
  hostname: 'localhost',
  environmentalScripts: [
    '<script>document.write("<script src='http://' + (location.host || "localhost").split(":")[0] + ":35729/livereload.js'></" + "script>");</script>'
  ],
  zapHostName: '192.168.56.20',
  zapPort: '8080',
  zapApiKey: 'v9dn0balpqas1pcc281tn5ood1',
  zapApiFeedbackSpeed: 5000
}
the options [useNewUrlParser] is not supported
the options [useUnifiedTopology] is not supported
Error: DB: connect
Error [MongoError]: failed to connect to server [localhost:27017] on first connect [Error: Client network
k socket disconnected before secure TLS connection was established
  at connResetException (node:internal/errors:683:14)
  at TLSSocket.onConnectEnd (node:_tls_wrap:1583:19)
  at TLSSocket.emit (node:events:377:35)
  at endReadableNT (node:internal/streams/readable:1312:12)
  at processTicksAndRejections (node:internal/process/task_queues:83:21) {
  name: 'MongoError'
}]
  at Pool.<anonymous> (/home/paris/projekty/INZYNIERKA/testing-envs/NodeGoat/node_modules/mongodb-core
/lib/topologies/server.js:336:35)
  at Pool.emit (node:events:365:28)
  at Connection.<anonymous> (/home/paris/projekty/INZYNIERKA/testing-envs/NodeGoat/node_modules/mongod
b-core/lib/connection/pool.js:280:12)
  at Object.onceWrapper (node:events:472:26)
  at Connection.emit (node:events:365:28)
  at TLSSocket.<anonymous> (/home/paris/projekty/INZYNIERKA/testing-envs/NodeGoat/node_modules/mongod
b-core/lib/connection/connection.js:189:49)
  at Object.onceWrapper (node:events:472:26)
  at TLSSocket.emit (node:events:365:28)
  at emitErrorNT (node:internal/streams/destroy:193:8)
  at emitErrorCloseNT (node:internal/streams/destroy:158:3)
  at processTicksAndRejections (node:internal/process/task_queues:83:21)

```

Rys. 6.4: Błąd konfiguracji MongoDB po wprowadzeniu zmian ręcznie

Po aktualizacji konfiguracji, aplikacja uruchomiła się poprawnie. Następnie przeprowadziłem testy funkcjonalne, które potwierdziły poprawność działania aplikacji.

```

[paris@uwuntu:~/projekty/INZYNIERKA/testing-envs/NodeGoat]
[15:33:21] > npm start

owasp-nodejs-goat@1.3.0 start
> node server.js

Current Config:
{
  port: 4000,
  db: 'mongodb://localhost:27017/nodegoat',
  cookieSecret: 'session_cookie_secret_key_here',
  cryptoKey: 'a_secure_key_for_crypto_here',
  cryptoAlgo: 'aes256',
  hostname: 'localhost',
  environmentalScripts: [
    '<script>document.write("<script src='http://' + (location.host || "localhost").split(":")[0] + ":35729/livereload.js'></" + "script>");</script>'
  ],
  zapHostName: '192.168.56.20',
  zapPort: '8080',
  zapApiKey: 'v9dn0balpqas1pcc281tn5ood1',
  zapApiFeedbackSpeed: 5000
}
Connected to the database
Express http server listening on port 4000

```

Rys. 6.5: Poprawne uruchomienie aplikacji NodeGoat po wprowadzeniu zmian ręcznie

6.7.3. Testy funkcjonalne po wprowadzeniu zmian

Tabela 6.5: Wyniki testów funkcjonalnych aplikacji NodeGoat po wprowadzeniu zmian ręcznie

Spec	Time	Tests	Passing	Failing	Skipped
allocations_spec.js	00:10	4	1	1	2
benefits_spec.js	00:02	5	-	1	4
contributions_spec.js	00:02	4	-	1	3
dashboard_spec.js	00:01	4	-	1	3
general_spec.js	00:01	3	-	1	2
learn_spec.js	00:01	2	-	1	1
login_spec.js	00:02	8	-	1	7
logout_spec.js	00:03	3	-	3	-
memos_spec.js	00:02	4	-	1	3
profile_spec.js	00:02	5	-	1	4
research_spec.js	00:01	4	-	1	3
signup_spec.js	00:02	5	-	1	4
tutorial_spec.js	00:08	13	-	13	-
Total (13 of 13 failed)		64	1	27	36

Niestety konfiguracje nadal nie były poprawne, a w plikach znajdowały się błędy typu not defined. W związku z tym, należało ręcznie wprowadzić zmiany w plikach konfiguracyjnych i zdebugować aplikację (przede wszystkim poprawne definicje zewnętrznych bibliotek oraz użycie mongodb zgodnie z dokumentacją nowej wersji sterownika). Problemem była również Po wprowadzeniu zmian, aplikacja uruchomiła się. Następnie przeprowadziłem testy funkcjonalne, które wykazały wiele błędów aplikacji.

Tabela 6.6: Test Results Summary

Spec	Time	Tests	Passing	Failing
allocations_spec.js	00:18	4	1	3
benefits_spec.js	00:22	5	1	4
contributions_spec.js	00:17	4	1	3
dashboard_spec.js	00:15	4	1	3
general_spec.js	00:10	3	1	2
learn_spec.js	00:05	2	1	1
login_spec.js	00:19	8	5	3
logout_spec.js	00:04	3	3	0
memos_spec.js	00:17	4	1	3
profile_spec.js	00:18	5	2	3
research_spec.js	00:11	4	1	3
signup_spec.js	00:13	5	3	2
tutorial_spec.js	00:05	13	13	0
Total (11 of 13 failed)		64	34	30

Problematyczna okazała się implementacja porównania haseł w pliku `user-dao.js`. W nowej wersji aplikacji, hasła były zapisywane w bazie danych w postaci hashu, co wymagało zmiany sposobu porównywania haseł. Błędy mogły wynikać z różnych implementacji tych funkcji kryptograficznych, które wynikały z podzielenia projektu do analizy na części.

6.7.4. Testy bezpieczeństwa po wprowadzeniu zmian

Po wprowadzeniu zmian, przeprowadziłem testy bezpieczeństwa aplikacji. W tym celu wykorzystałem narzędzie OWASP ZAP, które pozwala na przeprowadzenie testów bezpieczeństwa aplikacji internetowych.

OWASP ZAP

Tabela 6.7: Wyniki skanowania aplikacji NodeGoat przy użyciu skanera OWASP-zap

		Risk			
		High (= High)	Medium (>= Medium)	Low (>= Low)	Informational (>= Informational)
site	http://localhost:4000	0 (0)	5 (5)	3 (8)	5 (13)

OWASP ZAP wykrył 13 podatności, w tym 5 o średnim ryzyku, 3 o niskim ryzyku i 5 informacyjnych. Wśród nich były podatności takie jak:

Tabela 6.8: Summary of Security Alert Types, Risks, and Counts

Alert Type	Risk Level	Count (%)
CSP: Wildcard Directive	Medium	16 (123.1%)
Content Security Policy (CSP) Header Not Set	Medium	18 (138.5%)
Directory Browsing	Medium	2 (15.4%)
Missing Anti-clickjacking Header	Medium	18 (138.5%)
Vulnerable JS Library	Medium	2 (15.4%)
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Low	45 (346.2%)
Timestamp Disclosure - Unix	Low	1 (7.7%)
X-Content-Type-Options Header Missing	Low	25 (192.3%)
Authentication Request Identified	Informational	1 (7.7%)
Information Disclosure - Suspicious Comments	Informational	16 (123.1%)
Modern Web Application	Informational	16 (123.1%)
User Agent Fuzzer	Informational	120 (923.1%)
User Controllable HTML Element Attribute (Potential XSS)	Informational	13 (100.0%)

CodeQL

Tabela 6.9: Streszczone wyniki skanowania aplikacji NodeGoat przy użyciu skanera CodeQL

Issue	Description	Level	File	Occurrences
Polynomial regular expression	Vulnerable to DoS attacks. Slow on strings with many '0' repetitions or starting with '@'.	Warning	profile.js, session.js	2
Clear-text logging of sensitive info	Exposes sensitive data like passwords to attackers.	Error	user-dao.js	Multiple
Database query from user input	Vulnerable to code injection by the user.	Error	user-dao.js	2
Missing CSRF middleware	Lack of CSRF protection can allow malicious request submissions.	Error	server.js	Multiple
Rate limiting	Lack of rate limiting can lead to DoS attacks.	Warning	index.js	1
Server-side request forgery	Network requests with user data in URL can allow for attacks.	Error	research.js	1
DOM text as HTML	Can lead to cross-site scripting vulnerabilities.	Warning	bootstrap.js	Multiple
Unsafe jQuery plugin	Constructs HTML from options, potential XSS vulnerability.	Warning	bootstrap.js	Multiple

Po przeprowadzeniu popraw wykrytych zostało mniej podatności, co udowadnia o zwiększeniu bezpieczeństwa, dzięki wprowadzonym zmianom. Niestety zmiany te sprawiają, że aplikacja nie jest w pełni funkcjonalna, co może się przyczyniać do lepszych wyników skanu bezpieczeństwa.

6.8. IDENTYFIKACJA PODATNOŚCI BEZ PODPOWIEDZI W KODZIE

W kodzie aplikacji NodeGoat znajdowało się wiele wskazówek dotyczących podatności. Znacznie to ułatwia proces analizy kodu dla dużych modeli językowych, ponieważ mogę skupić się na konkretnych, wskazanych fragmentach kodu. W celu sprawdzenia jakie podatności wykryje GPTester bez podpowiedzi w kodzie, przeprowadziłem analizę kodu aplikacji NodeGoat bez podpowiedzi w kodzie.

Wyniki analizy kodu aplikacji NodeGoat bez podpowiedzi w kodzie

1. **MongoClient Connection String Exposure:** The `db` variable likely contains the connection string to MongoDB, which includes credentials. This sensitive information can be exposed if the configuration file is not properly secured.
2. **Insecure Express Session Configuration:** The session middleware in `server.js` uses a `cookieSecret` from a config file for signing cookies, but it does not have secure attributes such as `httpOnly`, `secure`, `sameSite`, as well as not using a store which can lead to security issues.
3. **SWIG Template Auto-Escape Disabled:** In `server.js`, the SWIG template engine is used with `autoescape: false`, making it vulnerable to Cross-Site Scripting (XSS) attacks if user input is included in templates.
4. **Use of Marked Library for Markdown Parsing:** The `marked` library is being used in `server.js` with the `sanitize` option set to `true`, which is good for preventing XSS. However, if library updates change the default behavior, it could lead to vulnerabilities.
5. **Database Reset Script:** In `db-reset.js`, the process exits with a different status code in case of an error. This can lead to inconsistent behavior and potentially leak information about the database status or structure.
6. **Lack of HTTPS:** The server in `server.js` is started with HTTP and not HTTPS, leading to the exposure of data in transit.
7. **Open Docker Ports in Development:** The `docker-compose.yml` file shows ports open which should not be exposed in a production environment.
8. **Hardcoded Credentials in db-reset.js:** Usernames and passwords are hardcoded in the `db-reset.js` script, which is a bad practice for managing credentials.
9. **Insecure Password Storage (user-dao.js):** Passwords should not be stored in plaintext. The `addUser` method currently stores the password received from the user directly into the database.
10. **Insecure Password Verification (user-dao.js):** The `validateLogin` method uses a simple string comparison to validate passwords, which would only be secure if the passwords were properly hashed and salted before storage and then compared using a secure function.

11. **Potential NoSQL Injection (research-dao.js, getBySymbol):** The `getBySymbol` method creates a query without proper sanitation or parameterized queries, which may open up the application to NoSQL injection attacks.
12. **Potential NoSQL Injection (profile-dao.js, updateUser):** The `updateUser` method directly uses the incoming parameter `userId` after parsing it as an integer. Although this reduces the risk, it's still a good habit to use a parameterized query.
13. **Lack of Input Validation:** Across various DAO functions, there is a lack of input validation to ensure that the values passed to the database operations do not contain malicious input.
14. **NoSQL Injection in Allocations (allocations-dao.js):** The method `getByIdAndThreshold` is susceptible to NoSQL injection as it constructs a query using a `$where` operator with user input, which can be manipulated.
15. **Insecure Configuration in Github Workflow (e2e-test.yml):** The configuration file uses hardcoded version "4.0" for the MongoDB Docker image, which might be outdated and contain known vulnerabilities.
16. **Arbitrary Redirect in Index Route (index.js):** The `/learn` route redirects to a user-specified URL without validation, which can be exploited for phishing attacks.
17. **Insufficient Logging and Monitoring in Error Handler (error.js):** The error handling middleware logs the error message but doesn't notify the team or use a centralized logging system.
18. **Cross-Site Scripting (XSS) in Profile Data Rendering (profile.js):** Directly embedding user input from `doc.website` in the HTML without encoding it, which can lead to Cross-Site Scripting attacks.
19. **Insufficient Password Strength Validation (session.js):** The `PASS_RE` regex allows for weak passwords that do not require a mix of character types.
20. **Remote Code Execution in Config File (config.js):** User input (`finalEnv`) is used to construct a file path without proper validation, which could allow an attacker to traverse the filesystem or execute arbitrary code.
21. **Insecure Direct Object References (IDOR) in Allocations (allocations.js):** User input from `req.params.userId` is used directly to query the database, which could allow an unauthorized user to access other users' data.
22. **Server-Side Request Forgery (SSRF) in Research (research.js):** The `url` and `symbol` parameters from user input are concatenated and used in a GET request without validation, allowing SSRF attacks.
23. **Cross-Site Scripting (XSS) in Memos (memos.js):** User input `req.body.memo` is directly inserted into the database and later rendered without proper encoding.
24. **Command Injection in Contributions (contributions.js):** Use of `eval` with user input `req.body.preTax`, `req.body.afterTax`, and `req.body.roth`, enabling command injection attacks.

Wyniki pokazują, że nawet bez podpowiedzi w kodzie, GPTester jest w stanie wykryć podatności w kodzie aplikacji NodeGoat. Są to bardzo zbliżone wyniki do tych otrzymanych z podpowiedziami.

6.9. WNIOSKI Z ANALIZY NODEGOAT

Na podstawie przeprowadzonych testów na aplikacji NodeGoat przy użyciu GPTester oraz technik LLM można wyciągnąć następujące wnioski:

1. **Potencjał LLM w identyfikacji podatności:** Large Language Models, takie jak GPT-4 wykorzystany w GPTester, wykazują potencjał w identyfikacji podatności w kodzie aplikacji. Ich zdolność do rozumienia i analizy kodu pozwala na efektywne wyszukiwanie zagrożeń bezpieczeństwa. Niestety wyniki trzeba zaktualizować o badania kodu nie posiadającego podpowiedzi, których NodeGoat posiada wiele.
2. **Ograniczenia LLM w kontekście analizy bezpieczeństwa:** Pomimo zaawansowanych możliwości, LLM napotyka ograniczenia związane z rozmiarem kodu, który mogą przetworzyć, oraz z interpretacją skomplikowanych zależności w aplikacjach. Wymaga to od programistów i specjalistów ds. bezpieczeństwa uzupełnienia analizy o ekspertyzę ludzką.
3. **Znaczenie interakcji człowieka z LLM:** Wyniki generowane przez GPTester i inne narzędzia wykorzystujące LLM mogą wymagać weryfikacji i doprecyzowania przez ludzi. Skuteczność LLM w analizie podatności wzrasta przy ścisłej współpracy z ekspertami ds. bezpieczeństwa.
4. **Koszty i ograniczenia zasobów:** Wykorzystanie LLM, zwłaszcza w dużych projektach, może być związane z wysokimi kosztami i ograniczeniami zasobów (np. limitami API). Optymalizacja procesu analizy pod kątem wyboru kodu do skanowania jest kluczowa dla zarządzania kosztami i efektywnością.
5. **Potrzeba integracji z innymi narzędziami:** Aby uzyskać kompleksowy obraz bezpieczeństwa aplikacji, zaleca się integrację LLM z tradycyjnymi narzędziami do skanowania podatności, takimi jak OWASP ZAP czy Nessus. Takie połączenie pozwala na bardziej wszechstronną analizę.
6. **Dynamika rozwoju LLM:** Szybki rozwój technologii LLM wskazuje na możliwość dalszego zwiększania ich skuteczności w analizie bezpieczeństwa aplikacji. Potencjał wykazuje również rozszerzenie parametrów dla OpenAI Assistant API. Kontynuacja badań i eksperymentów w tym obszarze jest zatem istotna.

PODSUMOWANIE

W pracy zbadano wykorzystanie dużych modeli językowych (LLM) w kontekście statycznej analizy kodu, skupiając się na ich zdolnościach do identyfikacji i naprawy błędów

programistycznych. Rozpatrzono różne aspekty stosowania LLM, w tym ich integrację z istniejącymi narzędziami do analizy kodu, potencjał w automatyzacji procesów weryfikacji kodu oraz wyzwania związane z ich praktycznym zastosowaniem. Praca porusza również kwestie związane z ograniczeniami modeli językowych, takie jak ich zależność od jakości i zakresu danych treningowych oraz konieczność humanitarnej nadzoru i weryfikacji wyników generowanych przez te systemy.

WNIOSKI

Duże modele językowe oferują obiecujące możliwości w analizie i naprawie kodu, jednak ich skuteczność jest ograniczona w złożonych scenariuszach cyberbezpieczeństwa. Wyniki badań podkreślają konieczność ludzkiej ekspertyzy w procesie weryfikacji i poprawy kodu. Dalsze badania są potrzebne do rozwoju metodologii i narzędzi, które pozwolą na pełniejsze wykorzystanie potencjału LLM w poprawie bezpieczeństwa aplikacji.

Na podstawie przeprowadzonych badań, można wyciągnąć następujące wnioski:

1. LLM mogą efektywnie identyfikować i proponować naprawy dla standardowych błędów w kodzie, ale ich skuteczność maleje wraz ze wzrostem złożoności zadania.
2. Modele te wymagają precyzyjnie sformułowanych zapytań i dobrze zdefiniowanych kontekstów, aby generować użyteczne wyniki. Konieczne są dodatkowe badania nad metodami wyboru i przygotowania danych wejściowych.
3. Ograniczenia LLM, takie jak brak głębokiego zrozumienia logiki programistycznej i kontekstu biznesowego, mogą prowadzić do nieoptymalnych lub niebezpiecznych sugestii.
4. Istotna jest ciągła interakcja i weryfikacja przez doświadczonych programistów, aby zapewnić bezpieczeństwo i poprawność proponowanych rozwiązań.
5. Rozwój narzędzi wspomagających, które integrują LLM z tradycyjnymi metodami statycznej analizy kodu, może zwiększyć skuteczność wykrywania i naprawy błędów.
6. Należy zachować ostrożność w kwestii etycznej i prawnej odpowiedzialności za błędy wprowadzone lub niezauważone przez modele językowe.
7. Dalsze badania powinny skupić się na usprawnieniu interakcji między LLM a programistami, zwłaszcza w kontekście uczenia się z interakcji i feedbacku.
8. Konieczne jest badanie wpływu na wydajność i jakość pracy programistów, w tym potencjalnych ryzyk związanych z nadmiernym poleganiem na automatycznych sugestjach.

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

3.1	Schemat blokowy działania aplikacji <i>gptester</i>	13
3.2	dane wejściowe w schemacie blokowym	14
3.3	Część schematu opisująca proces generatora testów funkcjonalnych . . .	14
3.4	Część schematu opisująca proces RAG	15
3.5	Część schematu opisująca proces RAG	16
3.6	Czarna skrzynka - LLM (Large Language Model)	16
3.7	Czarna skrzynka - LLM (Large Language Model)	17
3.8	Wiadomość pomocnicza aplikacji <i>gptester</i>	20
6.1	Wyniki testów funkcjonalnych aplikacji NodeGoat przed wprowadzeniem zmian	46
6.2	Błąd uruchomienia aplikacji NodeGoat po wprowadzeniu zmian	52
6.3	Błąd uruchomienia aplikacji NodeGoat po wprowadzeniu zmian ręcznie .	53
6.4	Błąd konfiguracji MongoDB po wprowadzeniu zmian ręcznie	54
6.5	Poprawne uruchomienie aplikacji NodeGoat po wprowadzeniu zmian ręcznie	54

SPIS LISTINGÓW

3.1	Kod tworzący reprezentację wektorową tekstu za pomocą API OpenAI, domyślnie 'text-embedding-ada-002', (models.py)	15
3.2	Kod porównujący semantyczną odległość (models.py)	15
3.3	Kod używany do komunikacji z API OpenAI (ai/assistant.py)	26
5.1	Kod źródłowy błędnego skrawka kodu <i>damnvuln.c</i>	39
5.2	Wynik działania programu w formacie diff na kodzie źródłowym <i>damnvuln.c</i>	40
5.3	Fragment kodu odpowiadający za wyświetlenie informacji o błędzie . . .	42
5.4	Fragment kodu odpowiadający za wyświetlenie informacji o błędzie . . .	43

Spis tabel

2.1	Metody i środki wykorzystane w projekcie i badaniu.	11
5.1	Vulnerable Code Snippets – oryginał – możliwe odpowiedzi w komentarzach	44
6.1	Wyniki skanowania aplikacji NodeGoat przy użyciu skanera OWASP-zap .	47
6.2	Podsumowanie typów alertów i powiązanych zagrożeń	47
6.3	Podsumowanie wyników analizy CodeQL	49
6.4	Extended Scanning Results with CodeQL	49
6.5	Wyniki testów funkcjonalnych aplikacji NodeGoat po wprowadzeniu zmian ręcznie	55
6.6	Test Results Summary	56
6.7	Wyniki skanowania aplikacji NodeGoat przy użyciu skanera OWASP-zap .	56
6.8	Summary of Security Alert Types, Risks, and Counts	57
6.9	Streszczone wyniki skanowania aplikacji NodeGoat przy użyciu skanera CodeQL	57

Dodatki

A. DODATEK 1

Surowe wyniki analizy kodu aplikacji NodeGoat bez podpowiedzi w kodzie

1. **MongoClient Connection String Exposure:** The `db` variable likely contains the connection string to MongoDB, which includes credentials. This sensitive information can be exposed if the configuration file is not properly secured.
2. **Insecure Express Session Configuration:** The session middleware in `server.js` uses a `cookieSecret` from a config file for signing cookies, but it does not have secure attributes such as `httpOnly`, `secure`, `sameSite`, as well as not using a store which can lead to security issues.
3. **SWIG Template Auto-Escape Disabled:** In `server.js`, the SWIG template engine is used with `autoescape: false`, making it vulnerable to Cross-Site Scripting (XSS) attacks if user input is included in templates.
4. **Use of Marked Library for Markdown Parsing:** The `marked` library is being used in `server.js` with the `sanitize` option set to `true`, which is good for preventing XSS. However, if library updates change the default behavior, it could lead to vulnerabilities.
5. **Database Reset Script:** In `db-reset.js`, the process exits with a different status code in case of an error. This can lead to inconsistent behavior and potentially leak information about the database status or structure.
6. **Lack of HTTPS:** The server in `server.js` is started with HTTP and not HTTPS, leading to the exposure of data in transit.
7. **Open Docker Ports in Development:** The `docker-compose.yml` file shows ports open which should not be exposed in a production environment.
8. **Hardcoded Credentials in db-reset.js:** Usernames and passwords are hardcoded in the `db-reset.js` script, which is a bad practice for managing credentials.

Here are the corrected segments of code for each vulnerability found:

1. MongoClient Connection String Exposure:

```
[] // No diffs displayed as it requires infrastructural practices like environment variable management or secret management systems, which can't be presented in source code directly.
```

2. Insecure Express Session Configuration:

```

-   app.use(session({
-       secret: cookieSecret,
-       saveUninitialized: true,
-       resave: true
-   }));
+   app.use(session({
+       secret: cookieSecret,
+       saveUninitialized: true,
+       resave: false,
+       cookie: {
+           httpOnly: true,
+           secure: true,
+           sameSite: "lax"
+       },
+       store: /* instance of secure session storage */
+   }));

```

3. SWIG Template Auto-Escape Disabled:

```

-   swig.setDefaults({
-       // Autoescape disabled
-       autoescape: false
-   });
+   swig.setDefaults({
+       % Autoescape enabled
+       autoescape: true
+   });

```

4. Use of Marked Library for Markdown Parsing: The current code has correctly set the sanitize option, but developers should remain vigilant for future library updates.

5. Database Reset Script (no explicit fix, informational only): Ensure consistent process exit codes and safeguard against information leakage.

6. Lack of HTTPS:

```

[]
-   http.createServer(app).listen(port, () => {
+   const https = require('https');
+   const fs = require('fs');
+
+   const options = {
+       key: fs.readFileSync('path/to/key.pem'),
+       cert: fs.readFileSync('path/to/cert.pem')
+   };
+
+   https.createServer(options, app).listen(port, () => {
+       console.log(`Express http server listening on port ${port}`);
+   });

```

7. Open Docker Ports in Development: Adjust the `docker-compose.yml` configuration for production to not expose sensitive ports publicly.

8. Hardcoded Credentials in `db-reset.js`:

```
-     "password": "Admin_123",
+     // Use environment variable or secure secrets management
+     // to retrieve password
+     "password": process.env.ADMIN_PASSWORD,
```

1. Insecure Password Storage (`user-dao.js`): Passwords should not be stored in plaintext. The `addUser` method currently stores the password received from the user directly into the database.

```
% Use the 'lstlisting' environment for code highlighting
- password %received from request param
+ password: bcrypt.hashSync(password, bcrypt.genSaltSync(8)) % Hash the password
```

2. Insecure Password Verification (`user-dao.js`): The `validateLogin` method uses a simple string comparison to validate passwords, which would only be secure if the passwords were properly hashed and salted before storage and then compared using a secure function.

```
- const comparePassword = (fromDB, fromUser) => {
-   return fromDB === fromUser;
- };
+ const comparePassword = (fromDB, fromUser) => {
+   return bcrypt.compareSync(fromUser, fromDB);
+ };
```

3. Potential NoSQL Injection (`research-dao.js`, `getBySymbol`): The `getBySymbol` method creates a query without proper sanitation or parameterized queries, which may open up the application to NoSQL injection attacks.

```
-   symbol
+   symbol: symbol
+   // The searchCriteria function should return the properly constructed query object
+   db.collection('research').find(searchCriteria()).toArray(callback);
```


4. **Potential NoSQL Injection (profile-dao.js, updateUser):** The updateUser method directly uses the incoming parameter userId after parsing it as an integer. Although this reduces the risk, it's still a good habit to use a parameterized query.

```
-          symbol
+          symbol: symbol
+          // The searchCriteria function should return the properly constructed query object
+          db.collection('research').find(searchCriteria()).toArray(callback);
```

5. **Lack of Input Validation:** Across various DAO functions, there is a lack of input validation to ensure that the values passed to the database operations do not contain malicious input.

For this point, code modifications would be more extensive and not as straightforward to display in a patch format because proper input validation would need to be implemented throughout each function that takes external input.

A.0.0.0.1. Vulnerability 6: NoSQL Injection in Allocations File: NodeGoat/app/data/allocations-dao.js

Issue: The method getByUserIdAndThreshold is susceptible to NoSQL injection as it constructs a query using a \$where operator with user input, which can be manipulated.

Vulnerability Fix:

```
this.getByUserIdAndThreshold = (userId, threshold, callback) => {
-   const parsedUserId = parseInt(userId);
-   const searchCriteria = () => {
-       if (threshold) {
-           return {
-               $where: `this.userId == ${parsedUserId} && this.stocks > '${thr
-           };
+   const parsedUserId = parseInt(userId)
+   let query = { userId: parsedUserId };
+
+   if (threshold) {
+       let numericThreshold = parseFloat(threshold);
+       if (!isNaN(numericThreshold)) {
+           query.stocks = { $gt: numericThreshold };
+       }
-   return {
-       userId: parsedUserId
```

```

-     };
-   };
+   }
-   allocationsCol.find(searchCriteria()).toArray(...);
+   allocationsCol.find(query).toArray(...);
  };

```

A.0.0.0.2. Vulnerability 7: Insecure Configuration in Github Workflow File: NodeGoat/.github/workflows

Issue: The configuration file uses hardcoded version "4.0" for the MongoDB Docker image, which might be outdated and contain known vulnerabilities.

Vulnerability Fix:

```

- docker run -d -p 27017:27017 mongo:4.0
+ docker run -d -p 27017:27017 mongo:latest

```

A.0.0.0.3. Vulnerability 8: Arbitrary Redirect in Index Route File: NodeGoat/app/routes/index.js

Issue: The /learn route redirects to a user-specified URL without validation, which can be exploited for phishing attacks.

Vulnerability Fix:

```

app.get("/learn", isLoggedIn, (req, res) => {
-   return res.redirect(req.query.url);
+   const allowedUrls = ["https://trustedresource.com/learn", "https://anothertrustedsource.com/resources"];
+   const requestedUrl = req.query.url;
+   if (allowedUrls.includes(requestedUrl)) {
+       return res.redirect(requestedUrl);
+   } else {
+       return res.status(400).send("Invalid URL provided for redirection.");
+   }
+ });

```

A.0.0.0.4. Vulnerability 9: Insufficient Logging and Monitoring in Error Handler

File: NodeGoat/app/routes/error.js

Issue: The error handling middleware logs the error message but doesn't notify the team or use a centralized logging system.

Vulnerability Fix:

```
+ const {logger} = require("../log"); // Hypothetical logging module that should be created

const errorHandler = (err, req, res,next) => {
-   console.error(err.message);
-   console.error(err.stack);
+   logger.error(err.message, {stack: err.stack, req});
  // ...
};
```

A.0.0.0.5. Vulnerability 10: Cross-Site Scripting (XSS) in Profile Data Rendering

File: NodeGoat/app/routes/profile.js

Issue: Directly embedding user input from `doc.website` in the HTML without encoding it, which can lead to Cross-Site Scripting attacks.

Vulnerability Fix:

```
this.displayProfile = (req, res, next) => {
  // ...
  doc.userId = userId;
-  doc.website = ESAPI.encoder().encodeForHTML(doc.website);
+  doc.website = ESAPI.encoder().encodeForURL(doc.website);
  // ...
};
```

A.0.0.0.6. Vulnerability 11: Insufficient Password Strength Validation

File: NodeGoat/app/routes/session.js

Issue: The `PASS_RE` regex allows for weak passwords that do not require a mix of character types.

Vulnerability Fix:

```
const validateSignup = (...) => {
+   const PASS_RE = /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,18}$/;
-   const PASS_RE = /^.{1,20}$/;

-   if (!PASS_RE.test(password)) {
+   if (!PASS_RE.test(password)) {
      errors.passwordError = "Password must be 8 to 18 characters" +
-        " including numbers, lowercase and uppercase letters.";
+        " including at least one number, one lowercase letter, and one uppercase letter.";
      return false;
    }
  // ...
};
```

A.0.0.1. Analyzing Vulnerabilities

A.0.0.1.1. Vulnerability 1: Remote Code Execution in Config File File: NodeGoat/config/config.js

Issue: User input (`finalEnv`) is used to construct a file path without proper validation, which could allow an attacker to traverse the filesystem or execute arbitrary code.

Vulnerability Fix:

```
const validateSignup = (...) => {  
+   const PASS_RE = /^.*(?:=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,18}$/;  
-   const PASS_RE = /^.{1,20}$/;  
  
-   if (!PASS_RE.test(password)) {  
+   if (!PASS_RE.test(password)) {  
       errors.passwordError = "Password must be 8 to 18 characters" +  
-       " including numbers, lowercase and uppercase letters.";   
+       " including at least one number, one lowercase letter, and one uppercase letter.";   
       return false;  
   }  
   // ...  
};
```

A.0.0.1.2. Vulnerability 2: Insecure Direct Object References (IDOR) in Allocations

File: NodeGoat/app/routes/allocations.js

Issue: User input from `req.params.userId` is used directly to query the database, which could allow an unauthorized user to access other users' data.

Vulnerability Fix:

```
this.displayAllocations = (req, res, next) => {  
  
   const {  
-     userId  
+     userId: rawUserId  
   } = req.params;  
  
+   // Verify the user ID from the session  
+   const {  
+     userId: sessionUserId  
+   } = req.session;  
  
+   const isAuthorized = rawUserId == sessionUserId; // Use proper authorization check  
+   if (!isAuthorized) {  
+     return res.status(403).json({ error: "Unauthorized access" });  
+   }  
  
   const {  
     threshold
```

```
    } = req.query;  
    // ...
```

A.0.0.1.3. Vulnerability 3: Server-Side Request Forgery (SSRF) in Research File: NodeGoat/app/routes/research.js

Issue: The `url` and `symbol` parameters from user input are concatenated and used in a GET request without validation, allowing SSRF attacks.

Vulnerability Fix:

```
this.displayResearch = (req, res) => {  
  
    if (req.query.symbol) {  
-        const url = req.query.url + req.query.symbol;  
+        const allowedDomains = ["https://api.example.com"]; // Replace with the actual domain you want to allow  
+        const defaultResearchUrl = allowedDomains[0] + "/stock_info"  
+        const safeSymbol = encodeURIComponent(req.query.symbol); // URI encode the symbol to avoid manipulation  
+        const url = defaultResearchUrl + "?symbol=" + safeSymbol;  
  
        return needle.get(url, ...);  
    }  
    // ...
```

A.0.0.1.4. Vulnerability 4: Cross-Site Scripting (XSS) in Memos File: NodeGoat/app/routes/memos.js

Issue: User input `req.body.memo` is directly inserted into the database and later rendered without proper encoding.

Vulnerability Fix:

```
const MemosDAO = require("../data/memos-dao").MemosDAO;  
const {  
    environmentalScripts  
} = require("../config/config");  
+const ESAPI = require("node-esapi");  
+  
function MemosHandler(db) {  
    // ...  
    this.addMemos = (req, res, next) => {  
+  
+        // Encode memo content to avoid XSS  
+        const encodedMemo = ESAPI.encoder().encodeForHTML(req.body.memo);  
  
-        memosDAO.insert(req.body.memo, (err, docs) => {  
+        memosDAO.insert(encodedMemo, (err, docs) => {  
    // ...
```

A.0.0.1.5. Vulnerability 5: Command Injection in Contributions File: NodeGoat/app/routes/contributions.js

Issue: Use of `eval` with user input `req.body.preTax`, `req.body.afterTax`, and `req.body.roth`, enabling command injection attacks.

Vulnerability Fix:

```
this.handleContributionsUpdate = (req, res, next) => {  
  
  -      const preTax = eval(req.body.preTax);  
  -      const afterTax = eval(req.body.afterTax);  
  -      const roth = eval(req.body.roth);  
  +      const preTax = parseFloat(req.body.preTax);  
  +      const afterTax = parseFloat(req.body.afterTax);  
  +      const roth = parseFloat(req.body.roth);  
  
      const {  
        userId  
      } = req.session;  
  // ...  
}
```