Universidade Estadual de Londrina

PEDRO ANTÔNIO MESSIAS LEMOS

DESENVOLVIMENTO E AVALIAÇÃO DE UM FUZZER PARA TESTES EM APIS REST

LONDRINA

2024

PEDRO ANTÔNIO MESSIAS LEMOS

DESENVOLVIMENTO E AVALIAÇÃO DE UM FUZZER PARA TESTES EM APIS REST

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação do Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Bruno Bogaz Zarpelão

LONDRINA

2024

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

P372d Lemos, Pedro Antonio Messias Lemos.
Desenvolvimento e Avaliação de um Fuzzer para Testes em APIs REST / Pedro Antonio Messias Lemos Lemos. - Londrina, 2024.
45 f. : il.

Orientador: Bruno Bogaz Zarpelão Zarpelão.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Graduação em Ciência da Computação, 2024.
Inclui bibliografia.

1. Fuzzing - TCC. 2. Testes Automatizados - TCC. 3. REST - TCC. 4. APIs Web - TCC. I. Zarpelão, Bruno Bogaz Zarpelão. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Graduação em Ciência da Computação. III. Título.

CDU 519

PEDRO ANTÔNIO MESSIAS LEMOS

DESENVOLVIMENTO E AVALIAÇÃO DE UM FUZZER PARA TESTES EM APIS REST

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação do Departamento de Computação da Universidade Estadual de Londrina, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

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

Prof. Dr. Gilberto Fernandes Júnior
Universidade Estadual de Londrina – UEL

Gabriel Esteves Messas
Universidade Estadual de Londrina – UEL

Londrina, 24 de abril de 2024.

AGRADECIMENTOS

Agradeço imensamente ao meu orientador, Prof. Dr. Bruno Bogaz Zarpelão, por todo o apoio, auxílio e mentoria indispensáveis ao desenvolvimento deste trabalho e durante meu percurso acadêmico.

Estendo meus agradecimentos aos colegas e amigos do curso, Rafael, Pedro, Blenda, Kristiano e Guilherme, pela colaboração e companheirismo que enriqueceram minha jornada.

Por fim, expresso minha profunda gratidão à minha família, pelo incentivo constante, e à Luana, minha namorada, por sua paciência e compreensão, que foram cruciais para que eu superasse os desafios e concluísse esta importante etapa da minha vida.

LEMOS, P. A. M.. **Desenvolvimento e Avaliação de um Fuzzer para Testes em APIs REST**. 2024. 43f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2024.

RESUMO

Com o aumento na utilização de APIs (*Application Programming Interfaces*) REST (*Representational State Transfer*) em sistemas contemporâneos, garantir a segurança dessas interfaces tornou-se um desafio crítico. Um *fuzzer* é uma ferramenta projetada para testes, introduzindo entradas aleatórias em um programa, com o objetivo de identificar possíveis falhas. No entanto, muitas ferramentas de *fuzzing* enfrentam limitações quando aplicadas às APIs, incluindo inadequação ao protocolo HTTP, dificuldades com mecanismos de autenticação e autorização, e desafios na manipulação de estruturas de dados complexas. Adicionalmente, a identificação precisa de erros é desafiadora devido à ambiguidade nas respostas das APIs, o uso de códigos de erro personalizados, e a possibilidade de erros silenciosos ou mascarados, tornando a detecção de vulnerabilidades uma tarefa complexa. Diante dessa problemática, este trabalho foca no desenvolvimento de um *fuzzer* especializado para APIs REST. O objetivo é identificar erros e vulnerabilidades através da análise de respostas da API, considerando especificidades como *endpoints*, argumentos e códigos de retorno esperados. O *fuzzer* também abordará desafios como respostas ambíguas, autenticação e validações específicas.

Palavras-chave: Fuzzing. Testes automatizados. REST. API Web.

LEMOS, P. A. M.. **Development and Evaluation of a Fuzzer for Testing REST APIs**. 2024. 43p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2024.

ABSTRACT

With the increasing use of REST (Representational State Transfer) Application Programming Interfaces (APIs) in contemporary systems, ensuring the security of these interfaces has become a critical challenge. A fuzzer is a tool designed for testing by introducing random inputs into a program, aiming to identify potential vulnerabilities. However, many fuzzing tools face limitations when applied to APIs, including incompatibility with the HTTP protocol, difficulties with authentication and authorization mechanisms, and challenges in handling complex data structures. Additionally, the precise identification of errors is challenging due to the ambiguity in API responses, the use of custom error codes, and the possibility of silent or masked errors, making the detection of vulnerabilities a complex task. In light of these issues, this work focuses on the development of a fuzzer specialized for REST APIs. The goal is to identify errors and vulnerabilities through the analysis of API responses, considering specifics such as endpoints, arguments, and expected return codes. The fuzzer will also address challenges such as ambiguous responses, authentication, and specific validations.

Keywords: Fuzzing. Automated testing. REST. Web API.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

No mundo contemporâneo, as tecnologias da Web permeiam diversos aspectos das nossas vidas cotidianas. À medida que a dependência de aplicações Web se intensifica, as Interfaces de Programação de Aplicativos (APIs - *Application Programming Interfaces*) assumem um papel crucial, servindo como o elo que permite a comunicação eficaz entre diferentes programas [1]. Entretanto, o incremento na complexidade das APIs também resulta no aumento de riscos de segurança associados a elas [2].

Deste modo, a segurança de APIs emerge como um tópico de grande relevância e interesse na comunidade científica e no setor industrial [3]. Há uma diversidade de métodos empregados para salvaguardar a segurança das APIs, os quais incluem a análise estática de código, a implementação de *firewalls* de aplicativos da Web e a utilização de *scanners* de vulnerabilidades. O *fuzzing* — uma técnica de teste automatizado que envolve o fornecimento de entradas malformadas, aleatórias ou semi-aleatórias a um programa — tem se mostrado eficaz na identificação de vulnerabilidades até então desconhecidas [4].

Contudo, ferramentas de *fuzzing* originalmente não foram desenvolvidas com um foco específico em APIs Web [5], já que possuem características intrínsecas, como mecanismos de autenticação, limitação de taxa e *logs* complexos, que muitas vezes não são adequadamente abordados por *fuzzers* tradicionais [6]. Portanto, devido a esses desafios, tais ferramentas frequentemente não atingem um nível de adequação satisfatório para testes nesse tipo de APIs [6].

O presente trabalho é motivado pela necessidade de desenvolver um *fuzzer* focado na avaliação de segurança de APIs REST. O objetivo central é criar e avaliar uma ferramenta que possa inserir diversos tipos de entradas aleatórias, considerando as peculiaridades das APIs e as limitações que *fuzzers* tradicionais possuem, tais como lidar com autenticação, analisar códigos de resposta, fornecer uma forma eficiente e personalizável de identificar vulnerabilidades e erros, bem como gerenciar diferentes formatos de entrada (parâmetros, JSON, etc.). O desenvolvimento, experimentos e avaliações propostos serão conduzidos no CIA-Agro UEL (Centro de Inteligência Artificial no Agro da UEL, financiado pela Fundação Araucária).

No Capítulo 2, é apresentada a fundamentação teórica, desenvolvendo conceitos de APIs, REST, segurança de APIs REST, testes de segurança, *fuzzing*, *fuzzing* de APIs e ferramentas de *fuzzing*. O Capítulo 3 explica o desenvolvimento e a implementação do *fuzzer* concebido nesse trabalho, incluindo detalhes técnicos e escolhas metodológicas. No Capítulo 4 são apresentados os resultados e as discussões acerta deles, obtidos a partir da execução do *fuzzer*. Por fim, o Capítulo 5 apresenta a conclusão deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 APIs

As APIs são conjuntos de protocolos, rotinas e ferramentas que permitem a comunicação e interação entre diferentes sistemas de software. Elas abrangem uma vasta gama de soluções, incluindo bibliotecas, *frameworks*, serviços Web e outros códigos reutilizáveis, evidenciando a diversidade e versatilidade dessas interfaces no mundo do desenvolvimento [7]. Conforme a definição fornecida pela AWS [8], uma API atua como uma porta através da qual diferentes serviços de *software* podem interagir, permitindo que um sistema solicite informações de outro e receba uma resposta em um formato padronizado.

Redhad *et al.* [9] também destacam que as APIs são utilizadas para a integração de sistemas, servindo como contratos que definem como os componentes de *software* devem interagir. Além disso, um dos grandes valores das APIs é a capacidade de reutilizar o código, permitindo que os desenvolvedores construam *softwares* com mais complexidade de maneira mais eficiente ao aproveitar o trabalho já realizado por outros. Esta reutilização não apenas acelera o processo de desenvolvimento, mas também garante que as melhores práticas e soluções testadas sejam incorporadas nos novos projetos [7].

A importância das APIs na arquitetura de *software* moderna é inegável [1] [10]. Elas desempenham um papel fundamental na construção de aplicações distribuídas, permitindo que diferentes serviços e componentes se comuniquem de forma eficiente e majoritariamente segura [9]. Com o aumento no uso de arquiteturas baseadas em microsserviços [11] e a proliferação de dispositivos conectados na Internet das Coisas (IoT, do inglês *Internet of Things*), as APIs tornaram-se ainda mais críticas para o funcionamento eficaz dos sistemas [12].

2.2 APIs Web

As APIs Web representam um subconjunto específico de APIs operando predominantemente sobre protocolos Web, como HTTP ou HTTPS, projetadas para serem consumidas por aplicações Web, dispositivos móveis e outros sistemas. Diferentemente das APIs tradicionais, as APIs Web são acessíveis remotamente e frequentemente utilizam formatos de dados padronizados, como JSON ou XML, para comunicação. Enquanto as APIs REST são um exemplo notório de APIs Web, existem outros estilos arquiteturais, como SOAP e GraphQL, demonstrando a versatilidade e a ampla aplicação das APIs no ambiente digital moderno [13]. Neste trabalho, nos concentraremos nas APIs REST, que serão detalhadas a seguir.

2.2.1 APIs REST

O paradigma REST (do inglês, *REpresentational State Transfer*) estabelece um conjunto de princípios arquiteturais que têm se mostrado fundamentais para o desenvolvimento de sistemas distribuídos na Web. As APIs que seguem esses princípios são frequentemente denominadas APIs REST. Essas interfaces são organizadas em uma coleção de serviços Web REST, cada um encarregado de gerenciar operações *CRUD* (Criar, Ler, Atualizar, Deletar, do inglês, *Create*, *Read*, *Update*, *Delete*) sobre determinados recursos [1].

Neste cenário, um "recurso" é uma entidade que pode ser disponibilizada na Web, podendo ser tão variada quanto um vídeo, uma imagem ou um pedido de compra. Estes recursos são comumente acessíveis através de Identificadores de Recursos Uniformes (URIs, do inglês *Uniform Resource Identifier*), o que os torna localizáveis e manipuláveis através de protocolos de aplicação, como o HTTP (do inglês, *HyperText Transfer Protocol*). Cada URI que aponta para um ou mais desses recursos é conhecida como um *endpoint* de API [1].

As APIs REST adotam um conjunto padronizado de diretrizes para a implementação de métodos HTTP, conforme descrito a seguir [1]:

- **GET /Ler**: utilizado para acessar e recuperar informações sobre um recurso ou conjunto de recursos;
- **POST /Criar**: empregado para a criação de novos recursos;
- **PUT /Alterar**: serve para modificar um recurso previamente existente;
- **DELETE /Deletar**: aplicado para a remoção de um recurso específico.

Por exemplo, considere uma API REST que gerencia uma biblioteca. Uma possível URI HTTP apontando para o recurso, poderia ser **'/livros'**. Neste caso, a operação HTTP **'GET /livros'** é usada para recuperar a lista de todos os livros e **'POST /livros'** poderia ser utilizada para adicionar um novo livro na biblioteca.

A URI do recurso e os métodos HTTP podem aceitar parâmetros de entrada para especificar informações adicionais para a execução das operações da API, como o identificador do livro a ser recuperado (por exemplo, '**/livros/{idLivro}**') ou um objeto estruturado para ser adicionado à coleção usando o método POST.

2.3 Ameaças à Segurança de APIs Web

Com a evolução das arquiteturas de *software* modernas, as APIs Web têm sido incorporadas em diversas soluções tecnológicas. Devido à sua ampla utilização e ao fato

de estarem entre os principais vetores de ataque [10], evidencia-se a relevância de se adotar padrões de segurança rigorosos no desenvolvimento e manutenção dessas interfaces [3, 10].

A segurança das APIs Web é comprometida por uma série de vulnerabilidades potenciais, muitas das quais são destacadas no OWASP API Security Top 10 [14]. A OWASP (do inglês, *Open Web Application Security Project*) é uma organização sem fins lucrativos que se dedica a melhorar a segurança dos softwares e, devido à sua abordagem aberta e colaborativa, as informações e ferramentas que ela fornece são amplamente reconhecidas pela comunidade global de segurança da informação. O documento da OWASP API Security Top 10 descreve as dez principais vulnerabilidades de segurança de APIs. Essas vulnerabilidades abrangem desde falhas em mecanismos de autenticação e autorização até a exposição indevida de dados sensíveis. Tais fragilidades não apenas ameaçam a integridade dos sistemas que dependem desses serviços, mas também podem resultar em violações de dados e impactos negativos nas operações comerciais [3]. A lista a seguir apresenta as dez principais vulnerabilidades de segurança de APIs, conforme definido pela OWASP [14] em 2023.

- **API1:2023 Broken Object Level Authorization**: trata-se de situações em que os *endpoints* de API são suscetíveis à manipulação do ID de um objeto enviado na solicitação, permitindo que atacantes acessem ou modifiquem objetos aos quais não deveriam ter acesso;
- **API2:2023 Broken Authentication**: vulnerabilidades surgem quando os mecanismos de autenticação de uma API são insuficientes ou mal implementados, tornando-os alvos fáceis para ataques;
- **API3:2023 Broken Object Property Level Authorization**: destaca-se pela exposição inadequada de propriedades de um objeto através de *endpoints* de API, particularmente em APIs REST e GraphQL;
- **API4:2023 Unrestricted Resource Consumption**: surge quando uma API não impõe limites adequados no consumo de recursos, como largura de banda, CPU, memória e armazenamento;
- **API5:2023 Broken Function Level Authorization**: representa situações em que uma API permite que usuários não autorizados acessem funções ou recursos específicos;
- **API6:2023 Unrestricted Access to Sensitive Business Flows**: relaciona-se à exposição de fluxos de negócios críticos sem restrições de acesso adequadas;
- **API7:2023 Server Side Request Forgery**: evidencia-se quando um *endpoint* da API busca um recurso remoto com base em URLs fornecidas pelo usuário, sem validação adequada;

- **API8:2023 Security Misconfiguration**: manifesta-se quando uma API opera com configurações de segurança inadequadas;
- **API9:2023 Improper Inventory Management**: resulta da falta de controle e documentação sobre as versões e *endpoints* de uma API;
- **API10:2023 Unsafe Consumption of APIs**: surge devido à ausência de controle e documentação adequados sobre as versões e *endpoints* de uma API.

Além da lista específica para segurança de APIs, a OWASP também elaborou o renomado "Top 10" que destaca as principais vulnerabilidades de segurança em aplicações Web em geral. Embora não seja focada em APIs, esta lista é relevante para o contexto deste trabalho, pois muitas das vulnerabilidades listadas também podem ser exploradas através de APIs. A lista a seguir apresenta as dez principais vulnerabilidades de segurança em aplicações Web, conforme definido pela *OWASP* [15] em 2021.

- **A1:2021 Broken Access Control**: mecanismos de autenticação de uma API insuficientes ou mal implementados facilitam ataques;
- **A2:2021 Cryptographic Failures**: uso de algoritmos criptográficos fracos ou implementação incorreta de algoritmos robustos;
- **A3:2021 Injection**: possibilidade de injeção de código malicioso, como SQL, *Cross-Site Scripting (XSS)* e injeção de comandos;
- **A4:2021 Insecure Design**: decisões de design que levam à falta de validação de entrada, ausência de mecanismos de autenticação, autorização e uso de componentes vulneráveis;
- **A5:2021 Security Misconfiguration**: configurações inadequadas, como padrões inseguros, armazenamento em nuvem mal configurado e definições incorretas de segurança do servidor Web;
- **A6:2021 Vulnerable and Outdated Components**: uso de componentes com vulnerabilidades reconhecidas;
- **A7:2021 Identification and Authentication Failures**: falhas na identificação e autenticação, como senhas fracas, métodos inseguros e falta de proteção contra-ataques de força bruta;
- **A8:2021 Software and Data Integrity Failures**: falhas na integridade de dados e *software*, como ausência de validações e proteção contra-ataques de repetição;
- **A9:2021 Security Logging and Monitoring Failures**: ausência de mecanismos adequados de monitoramento e *logging*;

- **A10:2021 Server-Side Request Forgery**: *endpoints* da API que buscam recursos remotos baseados em URLs do usuário sem validação apropriada.

2.4 Testes de segurança

Considerando a vasta gama de possíveis vulnerabilidades e os riscos inerentes associados ao desenvolvimento de *software*, a implementação de testes de segurança emerge como uma prática importante. Essa abordagem proativa é útil para identificar e mitigar falhas potenciais antes que elas se transformem em problemas reais, garantindo assim a integridade, a disponibilidade e a confidencialidade dos sistemas e dados. Ao antecipar as vulnerabilidades através de testes rigorosos, os desenvolvedores podem prevenir uma série de ameaças e fortalecer as defesas de suas aplicações contra ataques mal-intencionados.

Existe uma variedade de abordagens diferentes para testes de segurança, cada qual com suas vantagens e desvantagens, sendo que nenhuma será perfeita. Em um alto nível, existem três abordagens principais, testes do tipo *white box* (do inglês, caixa branca), *black box* (do inglês, caixa preta) e *gray box* (do inglês, caixa cinza) [16]. A diferença entre eles é definida pelo nível de acesso do testador aos recursos relacionados ao *software*. Em um dos extremos, *white box* requer um acesso completo ao código fonte, diagramas de design, etc. Em um outro extremo, *black box* implica pouco ou nenhum acesso aos recursos. Enquanto isso, em testes do tipo *gray box*, existe um acesso parcial a recursos, como documentação, por exemplo [17, 16].

2.4.1 Caixa Branca

Testes que utilizam a abordagem *white box*, também conhecido como *clear-box testing* (do inglês, teste de caixa transparente), teste estrutural ou teste baseado em código, têm como base a análise das estruturas internas de uma aplicação ou sistema [17, 16]. Nesse tipo de teste, o testador tem acesso à estrutura interna, design e implementação do código-fonte, arquitetura e outros detalhes técnicos do aplicativo ou sistema em teste [17, 16].

O teste de caixa branca pode ser realizado em diferentes níveis do ciclo de vida do desenvolvimento de *software* (SDLC, do ingles *Software Development Life Cycle*), incluindo testes unitários, testes de integração e testes de sistema [17]. Ele pode ser executado usando diferentes técnicas, como análise estática e análise dinâmica [17].

A análise estática envolve uma revisão do código-fonte ou do código binário de uma aplicação sem executá-la, utilizando ferramentas como revisores de código e verificadores de sintaxe. O objetivo é identificar defeitos e vulnerabilidades antes que a aplicação seja implantada [17].

A análise dinâmica, por outro lado, envolve a análise do comportamento de uma aplicação enquanto está em execução, utilizando ferramentas como depuradores, perfis de desempenho e monitores de desempenho. Tem como objetivo identificar defeitos e vulnerabilidades que podem aparecer apenas durante a execução [17].

2.4.2 Caixa Preta

Testes do tipo *black box* (do inglês, caixa preta) implicam que o testador tem conhecimento apenas do que pode observar externamente no comportamento do sistema ou aplicativo em teste. Este tipo de teste é orientado pelos resultados visíveis das ações do usuário, sem considerar como o *software* processa as entradas e produz as saídas. A ênfase está em validar a funcionalidade e conformidade do sistema com os requisitos especificados e em identificar quaisquer desvios ou defeitos comportamentais do ponto de vista do usuário final [17, 18].

Dentre as formas de teste do tipo caixa preta, destacam-se [17, 18]:

- testes de invasão: uma técnica em que o testador age como um atacante e tenta encontrar e explorar vulnerabilidades no sistema ou aplicação sem ter conhecimento prévio do seu funcionamento interno;
- testes aleatórios, incluindo *fuzzing*: onde o *software* é submetido a entradas aleatórias ou inválidas para verificar se ele pode lidar de forma segura e sem falhas;
- ferramentas de *proxy*: podem ser usadas para interceptar e modificar solicitações e respostas HTTP, permitindo que o testador examine o tráfego entre o navegador e o servidor e identifique vulnerabilidades;
- ferramentas de varredura automática: podem ser usadas para identificar vulnerabilidades em aplicativos Web sem a necessidade de interação manual do testador.

2.4.3 Caixa Cinza

No meio do caminho entre os testes de caixa branca e caixa preta, estão os testes *gray box* (do inglês, caixa cinza). Nesse contexto o testador tem conhecimento parcial do aplicativo. Nesse caso, informações sobre entrada do usuário, controles de validação de entrada e armazenamento de dados podem ser conhecidas pelo testador. O objetivo é verificar como a entrada do usuário é processada pelo aplicativo e armazenada no sistema de back-end [17].

2.5 Fuzzing

Fuzzing, abreviação de *fuzz testing* (do inglês, teste de *fuzzing*), é uma técnica automatizada de teste de software que emprega uma grande diversidade de entradas, tanto normais como anormais, para a aplicação alvo [19, 20]. Introduzido inicialmente por Miller *et al.* em 1988 [21], o processo envolve não apenas o fornecimento dessas entradas à aplicação, mas também o monitoramento contínuo dos estados de execução do *software* para identificar comportamentos inesperados, falhas ou travamentos. Esse monitoramento permite que os desenvolvedores e testadores detectem e corrijam vulnerabilidades que só se manifestam quando o *software* é submetido a condições extremas ou atípicas [20]. Em contraste com outras técnicas, o *fuzzing* destaca-se pela facilidade de implantação, extensibilidade e aplicabilidade, podendo ser executado com ou sem acesso ao código-fonte [19]. Adicionalmente, devido à sua natureza baseada na execução real do *software*, o *fuzzing* apresenta alta precisão [19].

A ferramenta que viabiliza essa técnica é denominada *fuzzer*. Atuando como um gerador de entradas, ele cria os dados de teste e injeta no programa alvo. Existem diferentes abordagens para a geração de testes, como os métodos baseados em mutação e gramática [6] — métodos de mutação modificam entradas de teste já existentes, enquanto técnicas baseadas em gramática geram novas entradas a partir de uma descrição estrutural da entrada. Além disso, os *fuzzers* podem operar em abordagens que vão desde a caixa-preta [5], sem conhecimento do código, até a caixa-branca, com total acesso ao código-fonte [20].

O panorama atual do desenvolvimento de *fuzzing* reflete um crescente interesse na abordagem de segurança de *software*. Nos últimos anos, a adoção de técnicas de *fuzzing* mostra um aumento constante [20, 19, 22] e tem sido reconhecida por sua eficácia para descobrir tanto falhas de implementação, quanto vulnerabilidades de segurança. Além disso, o *fuzzing* tem se integrado com outras técnicas avançadas, como algoritmo genético, análise de contaminação e execução simbólica [20]. A comunidade acadêmica demonstra um interesse crescente no assunto, como evidenciado pelo aumento nas publicações sobre *fuzzing* [3]. Este foco contínuo sugere que o *fuzzing* continuará a ser uma técnica valiosa no campo da segurança de software nos próximos anos [22].

2.5.1 Técnicas de *Fuzzing*

Existem diversas técnicas que foram introduzidas ao contexto de *fuzzing* para melhorar a geração de entradas aleatórias. O trabalho de Chen *et al.* [23] possui um levantamento das principais técnicas:

1. **Técnicas de Geração de Amostras**:

São empregadas para escolher e alterar sementes [1], além de limitar e criar novas amostras. Podem ser categorizadas em três grupos principais:

- ***Random mutation* (mutação randômica):** a ideia principal é utilizar mutação de algum campo para gerar uma semente que será utilizada para gerar novas entradas. Sistemas de *fuzzing* que tem como base essa técnica são rapidamente implementadas e possuem alta escalabilidade, já que não são computacionalmente exigentes. Porém, essa técnica é ineficiente para encontrar problemas complexos, já que não tem visão do estado de execução da aplicação e não possui algum tipo de parâmetro eficiente para as mutações;
- ***Grammar representation* (representação gramatical):** em contraste com a mutação randômica, essa técnica utiliza uma gramática – que pode ser feita automaticamente com técnicas como análise dinâmica ou manualmente – para gerar entradas mais direcionadas. É eficiente para aplicações com formatos de entrada mais complexos, mas necessita de um conhecimento do sistema para elaborar a gramática;
- ***Scheduling algorithms* (algoritmos de agendamento):** são métodos que tem como objetivo maximizar a saída do processo de *fuzzing*, com otimizações de escolha de estratégias para sementes e estratégia de mutação.

2. **Técnicas de análise dinâmica**:
   Tem como base a extração de informações dinâmicas a partir de um software em execução, podendo ser divididas em:

   - ***Dynamic symbolic execution* (execução dinâmica simbólica)**: consiste em determinar possíveis entradas de um programa usando valores simbólicos como entradas e gerando novos caminhos com base em restrições simbólicas coletadas;
   - ***Coverage feedback* (feedback de cobertura):** utiliza informações sobre quais partes do código de um programa foram executadas durante os testes para orientar a geração de casos de teste na próxima iteração. Isso resulta em uma cobertura de caminho aprimorada e aumenta a probabilidade de encontrar vulnerabilidades. É mais eficiente do que o *fuzzing* aleatório devido ao seu mecanismo de *feedback*;
   - ***Dynamic taint analysis*:** observa e identifica a propagação explícita e o uso inadequado de dados influenciados ou alterados por um invasor ou código malicioso na memória de um programa. Essa técnica utiliza dados de entrada com "rótulos" para especificar como o programa manipula esses dados de entrada e quais partes do programa foram afetadas pelos dados contaminados. Essa

[1] Refere-se a uma entrada inicial ou caso de teste que é usado como base para gerar novas entradas

técnica pode ser combinada com outras estratégias para aprimorar a precisão do *fuzzing*.

3. **Outros:**
   Uma outra técnica é o uso de aprendizado de máquina. Aprendizado de máquina pode ser usado para melhorar a eficiência e eficácia do processo de *fuzzing*, gerando melhores casos de teste e melhorando a cobertura de código. Também tem utilidade para identificar caminhos e seções potencialmente perigosos no programa, aprendendo com caminhos e seções em muitos programas que escondem vulnerabilidades. Outra maneira de usar o aprendizado de máquina em técnicas de *fuzzing* é ajudar a escolher uma estratégia de mutação após aprender os efeitos do uso de diferentes estratégias de mutação.

2.5.2 Vantagens e Desvantagens

Sendo uma ferramenta, das diversas que devem ser utilizadas para uma análise de segurança robusta, *fuzzing* tem vantagens e desvantagens, dentre as quais podemos citar as seguintes vantagens:

- pode ser relativamente simples e fácil de implementar [23];
- tem aplicação em uma ampla variedades de programas e sistemas [23, 4, 6];
- pode ser automatizado para executar testes repetitivos e de longa duração [23];
- tem a capacidade de detectar vulnerabilidades e *bugs* que não são facilmente identificáveis por outras técnicas de segurança [23];
- *fuzzing* pode ser utilizado para testar sistemas de *software* em tempo real, permitindo que problemas sejam corrigidos rapidamente [24].

Em contrapartida, destacam-se as seguintes desvantagens:

- pode ser difícil gerar casos de teste que explorem todas as possíveis vulnerabilidades [23, 25];
- pode produzir muitos resultados de teste falsos positivos ou falsos negativos [23];
- pode ser difícil determinar se um resultado de teste é uma vulnerabilidade conhecida ou uma vulnerabilidade desconhecida [23].
- pode ser difícil determinar a causa raiz de uma vulnerabilidade identificada pelo *fuzzing* [23].

2.6 Fuzzing de APIs REST

Para realizar o *fuzzing* de API REST, os testadores utilizam ferramentas especializadas que capturam e reproduzem o tráfego HTTP, analisam solicitações e respostas HTTP e realizam o processo de *fuzzing* com heurísticas predefinidas ou regras definidas pelo usuário [4, 6]. Algumas ferramentas também podem aproveitar especificações Swagger[2] para orientar o *fuzzing* e gerar casos de teste mais inteligentes.

Considerando que APIs REST possuem um formato de entrada que pode ser variado, *fuzzing* baseado em gramática se torna uma boa alternativa em comparação com entradas puramente aleatórias [6]. Não só isso, mas também requer a seleção cuidadosa de casos de teste e monitoramento do *feedback* dinâmico do serviço [4]. Ao combinar essas técnicas, os testadores podem alcançar uma cobertura de código elevada e detectar uma ampla gama de *bugs* e vulnerabilidades de segurança [4].

2.6.1 Ferramentas de *fuzzing* de APIs REST

Nos últimos anos, diversos avanços foram alcançados no campo do *fuzzing* de APIs REST, revelando ferramentas inovadoras e estratégias eficazes para a identificação de vulnerabilidades. Dentre as ferramentas disponíveis, destacam-se:

- **bBOXRT**: é uma ferramenta que tem como objetivo avaliar a robustez de serviços REST. A ferramenta usa um documento de descrição de serviço como entrada para gerar um conjunto de entradas inválidas que são enviadas ao serviço em combinação com parâmetros válidos. Além disso, a ferramenta pode operar como um proxy de injeção de falhas entre o cliente e o servidor. As respostas do serviço são analisadas preliminarmente em busca de casos suspeitos de falha e armazenadas para análise detalhada posterior pelo usuário da ferramenta [27];

- **EVOMASTER**: é uma ferramenta de código aberto que usa técnicas evolutivas para gerar casos de teste de nível de sistema para aplicativos corporativos e Web. Ele é projetado para ser uma ferramenta de caixa branca (mas também tem uma funcionalidade de caixa preta), o que significa que leva em consideração os detalhes internos do código do lado do servidor. Atualmente, ele se concentra em APIs REST em execução em JVMs (do inglês, *Java Virtual Machines*), mas pode ser estendido para outras linguagens e contextos de teste de sistema. É composto por dois componentes principais: um processo principal responsável pelas funcionalidades principais e um *driver* de processo que inicia, para e reinicia o sistema em teste e instrumenta

[2] Swagger permite a descrição da estrutura de APIs em um formato legível por máquinas, facilitando a geração automática de documentação interativa e bibliotecas de cliente em várias linguagens. Isso é alcançado ao fazer com que a API retorne um arquivo YAML ou JSON que adere à Especificação OpenAPI, detalhando operações, parâmetros, autorizações e informações adicionais sobre a API [26].

seu código-fonte. A ferramenta é capaz de encontrar falhas em projetos de código aberto e é uma alternativa para testes manuais e outras ferramentas de geração de testes [28];

- **RESTest**: é uma ferramenta de teste de caixa-preta de código aberto para APIs REST que suporta várias técnicas, incluindo *fuzzing*. Ela recebe como entrada a especificação da API em formato OAS (do inglês, *OpenAPI Specification*) e gera, e opcionalmente executa, casos de teste usando técnicas como *fuzzing*, teste aleatório adaptativo e baseado em restrições. RESTest depende de geradores de dados de teste personalizados para gerar automaticamente dados realistas, como endereços de e-mail e códigos de idioma. O framework pode ser facilmente estendido com novos geradores, técnicas de geração e escritores de teste. RESTest já se mostrou útil na detecção automatizada de *bugs* do mundo real em APIs comerciais usadas por milhões de usuários em todo o mundo [29];

- **RestCT**: utiliza testes combinatórios para gerar casos de teste automaticamente. A ferramenta é capaz de testar as interações de um número específico de operações, bem como as interações de parâmetros de entrada específicos em cada operação. A ferramenta é totalmente automática e pode ser usada com qualquer especificação Swagger [30];

- **RESTler**: é a primeira ferramenta de *fuzzing* automático e inteligente para APIs REST. Ela analisa uma especificação Swagger, infere dependências entre tipos de solicitação e gera testes definidos como sequências de solicitações que satisfazem essas dependências. A ferramenta aprende dinamicamente quais sequências de solicitações são válidas ou inválidas, analisando as respostas do serviço a esses testes. RESTler é capaz de encontrar novos bugs em serviços REST, incluindo vulnerabilidades de segurança, e é uma ferramenta promissora para testar a segurança e confiabilidade de serviços em nuvem [4];

- **RestTestGen**: utiliza uma abordagem de caixa-preta automatizada para APIs REST. Ele gera casos de teste para cenários de execução nominal e de erro, com o objetivo de detectar defeitos de implementação usando dois oráculos distintos. A ferramenta usa o grafo de dependência de operações para modelar explicitamente as dependências de dados entre as operações, e aplica operadores de mutação para gerar valores de entrada adicionais para cada operação. A validação empírica da ferramenta em 87 APIs REST reais revelou um grande número de defeitos de implementação [31].

- **Schemathesis**: possui testes baseados em propriedades para encontrar erros semânticos e falhas em APIs da Web OpenAPI ou GraphQL. A ferramenta é capaz de fornecer cobertura de código por meio de instrumentação no lado do servidor ou

suporte para testes em processo. Em uma avaliação comparativa com outras oito ferramentas, Schemathesis superou todas as outras em encontrar defeitos e lidar com serviços-alvo sem erros internos fatais. A ferramenta é mantida como um projeto de código aberto pelos desenvolvedores e pode ser usada em vários casos de uso no desenvolvimento de software [32];

- **Pythia**: combina *fuzzing* baseado em gramática com *feedback* guiado por cobertura e mutações baseadas em aprendizado para *fuzzing* de API REST com estado. Ele gera casos de teste gramaticalmente válidos e prioriza os casos de teste que são mais propensos a encontrar *bugs*. A ferramenta foi avaliada experimentalmente em três serviços em nuvem de código aberto em escala de produção, mostrando que supera abordagens anteriores em cobertura de código e encontrou 29 novos *bugs*, que estão sendo relatados aos proprietários dos serviços [6];

- **foREST**: usa uma abordagem baseada em árvore para gerar casos de teste e detectar *bugs* e vulnerabilidades. A ferramenta modela as dependências de APIs com uma estrutura de árvore, o que reduz a complexidade das dependências e captura a prioridade das dependências de recursos. A ferramenta foi avaliada em experimentos com serviços REST do mundo real e superou outras ferramentas de *fuzzing* de API REST em termos de cobertura de código e desempenho. A ferramenta está disponível como código aberto no GitHub [33];

- **Miner**: tem um sistema híbrido de dados para melhorar o desempenho na descoberta de erros profundos e *bugs* de segurança. A ferramenta implementa três novos designs que trabalham juntos para abordar as limitações dos *fuzzers* de API REST existentes. Miner utiliza um modelo de rede neural para melhorar a qualidade da geração de solicitações em um modelo de sequência e um verificador de regras de segurança baseado em dados para capturar erros causados por parâmetros indefinidos. A abordagem é genérica e pode ser aplicada a maioria dos *fuzzers* de API REST [34].

3 FUZZER PARA APIS REST

O objetivo do *fuzzer* desenvolvido neste trabalho é realizar testes em APIs REST com entradas válidas e inválidas, bem como analisar seus resultados. O *fuzzer* disponibiliza parâmetros e configurações que podem ser definidos a partir de um arquivo JSON. Além disso, a ferramenta conta com a capacidade de realizar requisições assíncronas, trazendo uma boa eficiência na execução e cobertura ampla de múltiplos cenários. Este conjunto de funcionalidades é importante para uma análise sólida das APIs, avaliando não apenas a precisão das respostas em condições normais, mas também o comportamento sob condições atípicas e potencialmente adversas, como entradas mal formadas ou que fogem do que a API espera receber.

3.1 Funcionamento do Fuzzer

Figura 1 – Fluxo de Execução

O *fuzzer* consiste da sequência de processos que é apresentada na Figura 1. O `'Carregamento da Configuração'` começa com a validação de um arquivo de configuração formatado em JSON. Este arquivo contém as definições dos parâmetros de cada teste, especificando detalhes como *endpoints*, métodos HTTP permitidos, e as condições sob as quais as respostas da API são consideradas válidas ou inválidas. A configuração inicial deve ser precisa, pois guia o comportamento subsequente da ferramenta e determina a relevância dos resultados obtidos.

Na segunda etapa, o *fuzzer* aciona o módulo de `'Geração de Dados de Teste'`, o qual cria uma variedade de entradas. As entradas são elaboradas para representar tanto casos de uso esperados quanto condições atípicas ou extremas, explorando assim a capacidade da API de lidar com um espectro amplo de requisições. Esses dados gerados são utilizados na fase de `'Envio de Requisições'` à API, onde o mecanismo de envio opera de maneira assíncrona, permitindo eficiência e escalabilidade no processo de teste. As respostas obtidas são então comparadas com os resultados esperados na etapa de `'Avaliação de Respostas'`, onde qualquer divergência é registrada. Por fim, os resultados dos testes são compilados em *logs* na etapa de `'Geração de Logs e Gráficos'`. Opcionalmente, nessa mesma fase, representações gráficas podem ser geradas, oferecendo

uma análise compreensiva do comportamento da API sob as condições de teste especificados. O restante da seção detalhará cada um dos blocos e o funcionamento de cada aspecto.

3.1.1 Carregamento da Configuração

O arquivo de configuração JSON é definido pelo usuário e tem como objetivo especificar os *endpoints*, argumentos e parâmetros gerais da API. Com essas especificações o *fuzzer* tem a capacidade de criar dados válidos e inválidos e depois comparar com as respostas esperadas para determinar se existe algum tipo de comportamento inesperado.

O componente de especificação da API requer uma URL base, definida pelo parâmetro `base_url` no arquivo, uma lista de `endpoints` e um parâmetro opcional `custom_header` que determina *headers* personalizados. A inclusão desse parâmetro opcional traz uma flexibilidade considerável para a ferramenta, permitindo simulações de testes mais próximas de cenários de uso real. Por exemplo, *headers* personalizados podem ser usados para testar APIs que requerem algum tipo de autenticação, bem como simular diferentes clientes.

A estrutura do arquivo de configurações segue conforme demonstrado na Figura 2:

Figura 2 – Exemplo de Configuração

```
{
  "base_url": "http://127.0.0.1:8000/api/v1/",
  "custom_header": {
    "API-Key": "chave"
  },
  "endpoints": [
        ...
        "arguments": [ ... ]
  ]
}
```

A configuração também permite que o usuário especifique cada um dos *endpoints*, seus atributos e parâmetros. Essa especificação é validada pelo *fuzzer* e mapeada para uma estrutura interna chamada `Endpoint` que será utilizada palas etapas de `Geração de Dados de Teste` — para saber os padrões que devem ser utilizados para a geração de testes — e pelo módulo de `Envio de Requisições` — para determinar os parâmetros de execução.

Um exemplo de configuração de *endpoint* é descrito na Figura 3.

Os parâmetros disponíveis para configurações no JSON são:

- `path`: indica o caminho para o *endpoint*, por exemplo, `/modelo`;

Figura 3 – Exemplo de Endpoint

```
{
    ...
    "endpoints": [
        {
          "path": "modelo",
          "method": "GET",
          "argument_type": "query_parameters",
          "expected_response_for_valid": [ "200" ],
          "expected_response_for_invalid": [ "400" ],
          "n_inputs": 100,
          "arguments": [ ... ]
        },
    ]
}
```

- `method`: representa o método HTTP que deve ser utilizado no request, por exemplo, `GET`, `POST`, etc;
- `argument_type`: indica se a requisição deve ser feito com os argumentos por parâmetros ou no formato JSON. Pode ser do tipo `query_parameters`, `json` ou `template`. O tipo template permite ao usuário indicar um arquivo JSON que será utilizado como base para os *requests*;
- `expected_response_for_valid` e `expected_response_for_invalid`: definem duas listas que o usuário precisa indicar, que definem para o *fuzzer* quais códigos de resposta HTTP são esperados para uma entrada válida e inválida. O *fuzzer*, então, compara a resposta recebida com as esperadas e, caso elas não coincidam pode-se identificar um comportamento inesperado;
- `n_inputs`: determina o número de *requests* feitos para o *endpoint*;
- `arguments`: com exceção dos casos em que o tipo de argumento é `template`, uma lista de argumentos é necessária para o *endpoint*;
- `template_filename`: quando o tipo template for utilizado, é necessário especificar o nome do arquivo que deve ser utilizado;
- `custom_header`: indica *headers* personalizados para o *endpoint*, caso existam.

Para cada *endpoint* do tipo `query_parameters` ou `json`, é necessário indicar uma lista de argumentos, especificados pela chave `arguments`. Cada parâmetro permite uma customização para que o usuário possa especificar o que deve ser considerado válido e, então, o *fuzzer* poderá criar dados inválidos. Cada argumento é definido conforme o exemplo apresentado na Figura 4.

Para os argumentos, os parâmetros disponíveis no JSON são:

Figura 4 – Exemplo de Argumento

```
"arguments": [
    {
      "name": "num",
      "type": "int",
      "min_acceptable_size": 0
    },
    {
      "name": "ponto_flutuante",
      "type": "float",
      "max_acceptable_size": 100,
      "min_acceptable_size": 0,
      "precision": 2
    },
    {
      "name": "parametro_limitado",
      "type": "string",
      "alphabet": "abcde123",
      "max_acceptable_size": 100,
      "min_acceptable_size": 0
    },
    {
      "name": "parametro_padrao",
      "type": "string",
      "alphabet": "/default",
      "max_acceptable_size": 100,
      "min_acceptable_size": 0
    },
  ]
```

- `name`: indica o nome do argumento;
- `type`: o tipo do argumento, que pode ser `int`, `float` ou `string`;
- `max_acceptable_size` e `min_acceptable_size`: todos os argumentos têm um parâmetro opcional de tamanho mínimo e máximo. Para inteiros e números de ponto flutuante esses valores representam o maior e menor número aceito e, no caso de *strings*, o comprimento máximo e mínimo. Quando um desses argumentos não é definido pelo usuário, são utilizados o menor e maior valor para inteiros (-2147483648 e 2147483647).
- `alphabet`: o tipo *string* também permite a definição opcional de um alfabeto ou selecionar um dos alfabetos disponibilizados pelo *fuzzer*. A definição é determinada por uma *string* que contém todos os caracteres que podem ser utilizados.

No tipo de entrada `template` (demonstrado na figura 5), é especificado um arquivo adicional que define o formato de envio de dados. Este arquivo servirá como modelo de JSON a ser enviado: as chaves definidas pelo usuário permanecem as mesmas, enquanto as chaves que terminarem com `-FUZZABLE` terão seus dados gerados automaticamente du-

rante a fase de `Geração de Dados de Testes`. Esta abordagem permite uma flexibilidade maior na definição de dados para teste com estruturas mais complexas, mantendo o formato básico do JSON intacto e gerando valores automaticamente para as chaves especificadas como *fuzzable*.

As definições de dados seguem o padrão: `STR min max /default`. O prefixo `STR` indica que será gerada uma `string`. O valor `min` representa o tamanho mínimo da `string`, enquanto `max` representa o tamanho máximo. Por fim, `/default` especifica o alfabeto que deve ser considerado válido, podendo ser uma *string* de caracteres ou selecionar um dos alfabetos padrões disponíveis com o uso de "`/`" e o nome do alfabeto.

A figura 5 representa um exemplo dessa estrutura.

Figura 5 – Exemplo de Arquivo de Template

```
{
  "customerID-FUZZABLE": "STR 0 10 /default",
  "customerName-FUZZABLE": "STR 0 10 /default",
  "customerAge": "10",
  "dict-FUZZABLE": {
    "id-FUZZABLE": "STR 0 10 abcdefgh",
    "name-FUZZABLE": "STR 0 10 abcdefgh",
    "dictInsideDict-FUZZABLE": {
      "id-FUZZABLE": "STR 0 10 /default"
    },
    "age": 10,
    "times": 10
  },
  "shippingAddress": {
    "street": "123 Main St",
    "city": "Springfield",
    "state": "IL",
    "zip": "62704"
  },
  "paymentMethod": {
    "type": "credit card",
    "cardNumber": "4111111111111111",
    "expirationDate": "12/25",
    "cvv": "123"
  }
}
```

3.1.2 Geração de Dados de Testes

Nesse módulo, são implementadas funcionalidades para a geração de dados de teste com base no que foi definido pelo processo de `Carregamento da Configuração`. São definidos os alfabetos disponíveis para *strings* e três geradores de dados, um para cada tipo de argumento possível: inteiro, número de ponto flutuante e *strings*. A estrutura armazena os códigos esperados para válidos, inválidos, bem como o número de inputs que devem ser gerados e os tamanhos máximos e mínimos. O modelo responsável pela geração

de `string` possui um atributo a mais que armazena o alfabeto e o de *float* possui um para precisão.

Para cada um dos geradores, são definidos 4 processos:

- `_generate_valid_input_data`: método responsável por criar um dado válido, respeitando os parâmetros passados para o gerador. Ex.: no caso de um inteiro entre 0 e 100, esse método retornará um número aleatório dentro desse intervalo;
- `_generate_invalid_input_data`: gera um dado inválido extrapolando os parâmetros definidos. Utilizando o também o exemplo de um inteiro entre 0 e 100, esse método retornará um número inteiro fora do intervalo determinado ou uma *string* aleatória;
- `_generate_valid_inputs`: executa o método `_generate_valid_input_data` N vezes e retorna uma lista de elementos da classe `Input` com o dado gerado, o código esperado e um booleano indicando que a entrada gerada é válida;
- `_generate_invalid_inputs`: análogo ao método anterior, mas as instâncias geradas são do tipo inválido.

Ainda dentro desse módulo, é definida uma estrutura interna `Input` que armazena o dado que será inserido no argumento e um atributo que indica se esse dado é válido ou não. Com esse atributo sabe-se se o código esperado é para um dado válido ou inválido. O objetivo dessa estrutura é permitir que o *fuzzer*, no módulo de `Avaliação das Respostas`, identifique qual o código esperado pela requisição, de modo que caso uma requisição tenha vários argumentos, um único que seja inválido determina que o código de retorno da requisição deve ser o de dados inválidos, caso todos sejam válidos, o código esperado é o de dados válidos.

O método `generate`, implementado pelo módulo, gera uma lista de entradas misturando entradas válidas e inválidas. Primeiro, ele cria duas listas separadas: uma com entradas válidas e outra com entradas inválidas para aquele argumento. Depois, divide a lista de entradas válidas em duas partes: os primeiros 30% dessas entradas formam a primeira parte, e o restante das entradas válidas são combinadas com todas as inválidas para formar a segunda parte. Essa segunda parte é então embaralhada (misturada aleatoriamente). Por fim, o método retorna uma nova lista, começando com a primeira parte das entradas válidas seguida pela segunda parte misturada de entradas válidas e inválidas. Essa divisão visa otimizar a eficiência da fase de `Análise de Respostas` do *fuzzer*, ao estabelecer uma linha de base confiável com as primeiras requisições válidas. Dessa forma, a identificação de comportamentos anormais da API é facilitada, permitindo uma avaliação mais precisa sobre como entradas inválidas impactam sua estabilidade e segurança.

Esse sistema de geração de dados é utilizado pela classe `Endpoint` através do método `generate_inputs`. Esse método é projetado para criar uma lista de objetos da classe `Request`, selecionando a estratégia de geração com base no tipo de argumento especificado. Se o atributo `argument_type` for "template", ele chama `_generate_templates()` para produzir entradas com base em templates pré-definidos. Caso contrário, recorre a `_generate_inputs_regular()` para gerar entradas de maneira padrão. Esses métodos percorrem a lista de *inputs* passada pelo arquivo de configuração, criam as instâncias da classe `Input` e depois agrupam os dados (os diferentes parâmetros do *request*) para formar os `Requests` — uma estrutura interna que armazena todos os dados referentes as requisições.

A `Request` é definida como descrito na figura 6.

Figura 6 – Estrutura Request

```
class Request:
        data
        expected_codes
        method
        url
        headers
        actual_code
        text_response
        curl
        datetime
        is_valid
        is_error
```

Os atributos são:

- `data`: contém o nome dos parâmetros e os dados gerados;
- `expected_codes`: armazena os códigos esperados para a requisição;
- `method`: tem o código HTTP que deve ser utilizado;
- `url`: define a URL do endpoint;
- `headers`: atribui os *headers* personalizados, caso tenha;
- `actual_code`: um atributo que vai armazenar o código recebido da API;
- `text_response`: armazenará o texto da resposta da API;
- `curl`: guarda o comando cURL[1] para executar o *request*. Permite o usuário verificar e reproduzir requests que tiveram algum problema;

[1] comando utilizado em sistemas Unix e baseados em Unix para transferir dados de ou para um servidor, suportando diversos protocolos, como HTTP, HTTPS, FTP, entre outros [35]

- `datetime`: arquiva o horário em que o *request* foi feito. É utilizado para ordenar os requests por ordem cronológica;
- `is_valid`: é um booleano que armazena se o *request* deve ser tratado como válido ou inválido;
- `is_error`: define se a API lidou com o *request* de forma esperada ou não.

Para as entradas do tipo `template_json`, são feitas cópias da estrutura JSON com os dados gerados, de modo que as chaves que terminam com `-FUZZABLE` são atualizadas com os dados gerados, e o sufixo `-FUZZABLE` é removido da chave.

3.1.3 Envio de Requisições

Para cada *endpoint* especificado no arquivo de configurações, uma lista de `Request` é criada no processo de `Geração de Dados de Teste`. Posteriormente, nesse módulo, esta lista é processada assincronamente, a uma taxa de dez requisições por segundo, utilizando os dados mantidos na estrutura. À medida que os *requests* são realizados, os demais atributos ainda não definidos, tais como `actual_code` e `text_response`, são atualizados. Esse procedimento assegura que, ao término da execução, a lista de `Request` esteja organizada em sequência de execução com todos os dados relevantes preenchidos, facilitando assim a análise subsequente no módulo de `Avaliação das Respostas`.

3.1.4 Avaliação das Respostas

Após a realização das requisições, procede-se com uma análise detalhada de cada uma, identificando aquelas cujo código de retorno diverge do esperado. Para esses casos, o `Request` é marcado, e o incidente é registrado no *log*. Além disso, conduz-se uma avaliação específica para detectar *requests* que possam ter comprometido a estabilidade da API. Esta análise consiste em identificar se há ocorrências de erro isoladas, caracterizadas por um *request* com falha precedido unicamente por operações bem-sucedidas e seguido exclusivamente por falhas. Tal padrão sugere uma possível causa de interrupção no serviço. Cabe destacar que, dada a coleta abrangente de dados sobre todos os *requests* realizados, o conjunto de informações disponíveis oferece amplas possibilidades para futuras análises detalhadas.

3.1.5 Logs e Gráficos

Durante a execução do programa, são exibidos *logs* diretamente no console para indicar o *endpoint* atual e o passo do processo. Além disso, os *logs* são registrados em arquivos de texto para análises futuras.

Os *logs* nos arquivos de texto são organizados da seguinte forma: é criada uma pasta com o nome `fuzzer` seguido do ID do processo. Dentro dessa pasta, há um arquivo chamado `fuzzer.log` que armazena os *logs* da execução. Também são criados subdiretórios para cada *endpoint* testado, nomeados com o método HTTP e o caminho do *endpoint*. Esses subdiretórios contêm *logs* de erros válidos (`valid_error.log`) e inválidos (`invalid_error.log`), bem como *logs* de operações bem-sucedidas, quando o modo verboso está ativado.

Os *logs* de erros incluem detalhes sobre o código de status retornado, os códigos esperados, os dados da requisição, cabeçalhos, método, caminho, o comando cURL correspondente e a resposta textual. Essas informações facilitam a identificação e análise de falhas.

Adicionalmente, gráficos representando os códigos de status das respostas de cada *endpoint* são salvos nos subdiretórios correspondentes, proporcionando uma visualização gráfica do comportamento das requisições ao longo do teste.

Um exemplo da estrutura de arquivos gerados após a execução segue conforme a figura 7.

Figura 7 – Estrutura de Arquivos

```
fuzzer4222/
|--- fuzzer.log
|--- graphs/
     |--- POST_tokens.png
|--- POST_tokens/
     |--- invalid_error.log
     |--- valid_error.log
     |--- ok.log
|--- GET_tokens/
     |--- ok.log
```

4 AVALIAÇÃO E RESULTADOS

Para a avaliação do *fuzzer* desenvolvido neste estudo, adotaram-se duas abordagens distintas. Inicialmente, aplicou-se o *fuzzer* ao ConsistencIA, um sistema real em desenvolvimento usado pelo CIA-Agro (Centro de Inteligência Artificial no Agro) da Universidade Estadual de Londrina. Essa avaliação visou testar a eficácia do *fuzzer* em um ambiente operacional real. Em seguida, para ampliar a gama de cenários testados e aumentar a robustez dos resultados, o *fuzzer* foi também aplicado a uma API sintética, a qual foi elaborada e descrita no artigo [25]. Esta abordagem dupla permitiu não apenas validar a ferramenta em condições reais de uso, mas também explorar sua capacidade de identificar falhas em um ambiente controlado e diversificado.

4.1 Testes no ConsistencIA

O ConsistencIA, desenvolvido no âmbito do CIA-Agro — um centro de pesquisa focado na aplicação de Inteligência Artificial para resolver problemas do setor agropecuário —, é uma ferramenta que integra dados de sensores meteorológicos para prever anomalias nos conjuntos de dados coletados, utilizando para isso algoritmos de IA.

Esta ferramenta é acessível através de uma API REST que oferece operações CRUD (*Create, Read, Update, Delete*). Os *endpoints* que permitem a entrada de dados pelos usuários, e que foram submetidos a testes utilizando o *fuzzer*, são enumerados na Tabela 1.

Endpoint	Método	Dados esperados	Códigos esperados de retorno
/algoritmo	POST	name: string até 36 caracteres version: string até 8 caracteres library: string até 36 caracteres	201: criação bem-sucedida 400: parâmetros incorretos
	DELETE	id: int maior que 1	204: remoção bem-sucedida 200: não encontrado 400: parâmetros incorretos
/feature	POST	name: string até 16 caracteres	201: criação bem-sucedida 400: parâmetros incorretos
	DELETE	id: int maior que 1	201: criação bem-sucedida 400: parâmetros incorretos
/modelo	GET	id_feature: int maior que 1	200: sucesso 400: não encontrado ou erro nos parâmetros
	POST	id_feature: int maior que 1 id_algorithm: int maior que 1 hyperparameters: string até 64 caracteres	201: criação bem-sucedida 400: parâmetros incorretos
	DELETE	id_model: int	204: remoção bem-sucedida 200: não encontrado 400: parâmetros incorretos
/sample	GET	data_hora_inicial: string data_hora_final: string id_sensor: int	200: sucesso 400: não encontrado ou erro nos parâmetros
/sensor	POST	id_station: int maior que 1 id_feature: int maior que 1 id_model: int maior que 1 name_sensor: string	201: criação bem-sucedida 400: parâmetros incorretos
	DELETE	id_sensor: int maior que 1	204: remoção bem-sucedida 200: não encontrado 400: parâmetros incorretos
/station	POST	name: string até 128 caracteres id_idr: int maior que 1 latitude: float longitude: float descrição: string até 256 caracteres	201: criação bem-sucedida 400: parâmetros incorretos

Tabela 1 – Endpoints ConsistencIA

Para iniciar o processo de *fuzzing*, configurou-se um arquivo JSON, detalhando os *endpoints* que seriam submetidos a testes, os tipos de dados correspondentes e os códigos de retorno esperados, como descrito na Tabela 1. Vale ressaltar que esse arquivo de configuração também especifica um cabeçalho personalizado contendo uma chave de API, necessário para a autorização das requisições. Para cada *endpoint*, foram realizados 100 requisições.

Durante os testes, a execução do *fuzzer* levou a alguns problemas de funcionamento da API, que demandaram reinicializações. Quando isso ocorreu, o *fuzzer* foi reiniciado para prosseguir com os testes nos *endpoints* restantes. O principal objetivo destes testes, ao executar a ferramenta, foi assegurar que a API realiza a devida validação das entradas do usuário, assegurando não apenas a precisão dos tipos de dados, mas também a correta emissão dos códigos de resposta, conforme as expectativas para dados válidos e inválidos.

Ao final da execução, a análise dos *logs* do *fuzzer* revelou várias falhas em diferentes *endpoints*. As causas desses erros foram identificadas por meio de uma análise detalhada

dos *logs* produzidos pela API.

Dentre o que foi encontrado para `/algoritmo`, destacam-se os seguintes pontos:

- a API não valida adequadamente os parâmetros recebidos, resultando em códigos de retorno inesperados, como 500 em vez de 400 para strings inválidas;
- erros nas operações do banco de dados — como a tentativa de inserção de um dado que não respeita os atributos da tabela — não são tratados adequadamente, causando falhas persistentes em futuras operações até que o servidor seja reiniciado. Esse problema torna-se evidente ao examinar o gráfico da figura 8, produzido durante a execução do *fuzzer*. Os dados válidos consistem em strings que respeitam a restrição de tamanho estabelecida, enquanto os dados inválidos são representados por strings que excedem esse limite máximo. O gráfico na figura 8 revela que a primeira tentativa de inserção no banco de dados com uma *string* maior que o tamanho permitido resultou em uma falha persistente, causando o erro observado.

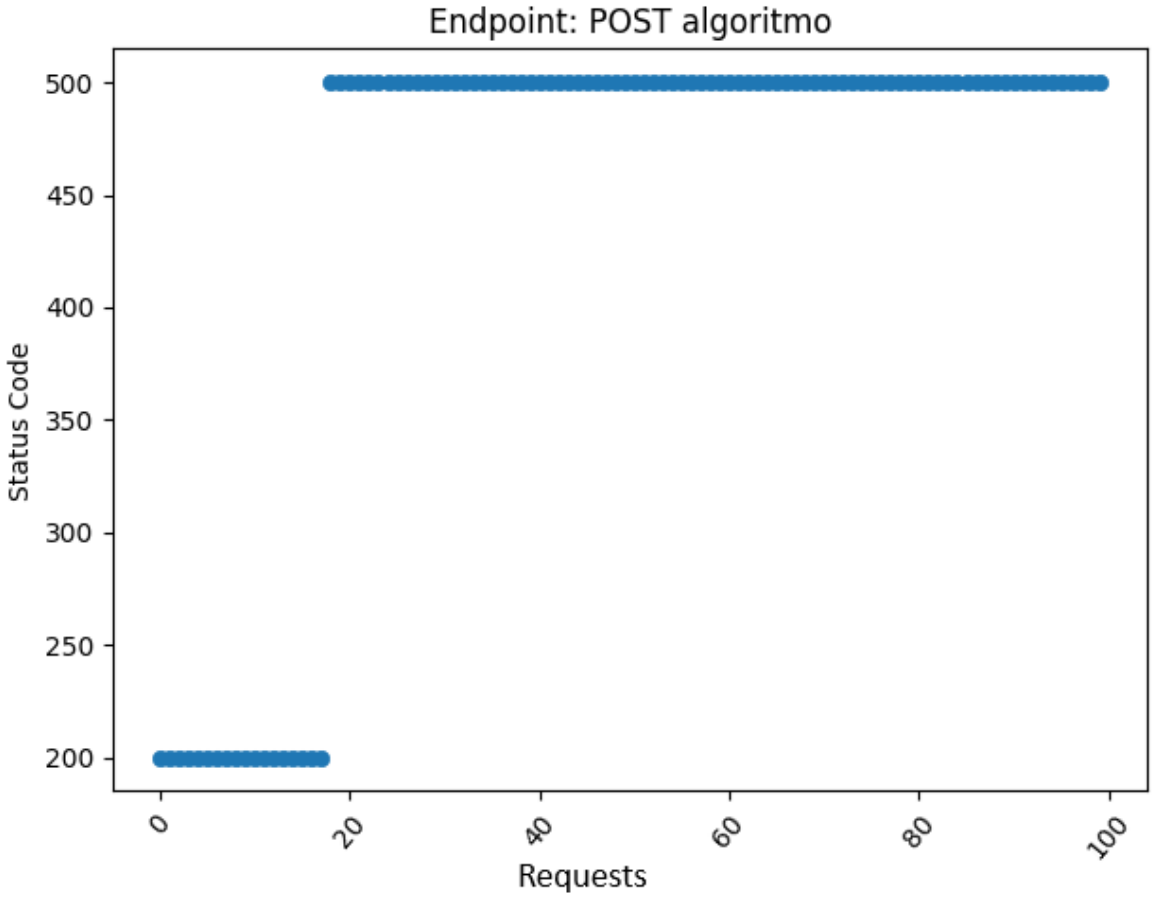

Figura 8 – POST /algoritmo

Em relação a `/feature`, podemos observar os seguintes aspectos importantes:

- similar ao `algoritmo`, as operações `POST` e `DELETE` sofrem de falta de validação de parâmetros, levando a códigos de retorno errôneos, como 500 em vez de 400 para entradas inválidas.

No contexto de `/modelo`, são importantes os seguintes elementos:

- Problemas semelhantes com falta de validação de parâmetros e tratamento inadequado de erros no SQL Alchemy[1], resultando em códigos de retorno inesperados, principalmente 500 para entradas inválidas.

Para `/sample`, evidenciam-se os seguintes pontos:

- a operação `GET` revelou uma limitação no *fuzzer*: a ausência de suporte para formatar entradas de *string* de acordo com padrões específicos. Para o *endpoint* em questão, o padrão exigido é o formato de data: `2016-01-01 00:00:00`, por exemplo. O *endpoint* verifica se os parâmetros `data_hora_inicial` e `data_hora_final` estão conforme o padrão de data esperado. Como o *fuzzer* não é capaz de gerar dados nesse formato específico, não conseguimos produzir dados válidos para testar completamente o *endpoint*. Apesar disso, o teste identificou que, quando a data fornecida era inválida, o servidor retornava um código 200 com uma mensagem de erro, em vez do esperado código 400.

Para `/sensor`, salientam-se as seguintes características:

- todas as operações (`GET`, `POST` e `DELETE`) demonstram a mesma falta de validação de parâmetros, resultando em códigos de retorno inesperados, especificamente 500 para entradas inválidas. Contudo, a operação `POST` enfrentou problemas adicionais quando tratava de inteiros superiores ao limite suportado pelo tipo inteiro, o que não foi adequadamente gerenciado pelo servidor. Isso causou uma falha persistente, que exigiu a reinicialização do servidor — um problema similar ao observado no *endpoint* `/algoritmo` com o método `POST`.

No que tange a `/station`, as particularidades são:

- similar às outras entidades, a operação `POST` do *endpoint* `/station` não valida adequadamente os parâmetros, resultando em códigos de retorno incorretos. Uma tentativa de inserção com um ID maior que o teto suportado pelo tipo inteiro resultou em uma falha persistente semelhante à observada na operação `GET` de sensor e `POST` de algoritmo.

[1] ORM para python utilizada na aplicação sendo testada

Uma análise geral dos erros indicados nos trouxe as seguintes conclusões:

- a falta de validação de parâmetros é uma falha crítica na API, resultando em comportamento inesperado e potencialmente inseguro;
- os erros no SQL Alchemy não são tratados adequadamente, levando a falhas persistentes que afetam todas as operações futuras até que o servidor seja reiniciado;
- alguns códigos de retorno para requisições com dados inválidos não estão retornando o código 400;
- a partir dos resultados do *fuzzer* foi visto que é importante realizar uma revisão completa do código para implementar validação adequada de parâmetros e tratamento de erros robusto para garantir a estabilidade e segurança da API.

A utilização do *fuzzer* no sistema ConsistencIA revelou que a ferramenta foi eficaz na identificação de uma série de falhas. Os testes permitiram descobrir deficiências em relação à validação de parâmetros e ao tratamento de erros. As características do *fuzzer*, tais como a geração automática de entradas (válidas e inválidas) variadas e a capacidade de comparar o código de retorno recebido com o esperado, desempenharam um papel importante para expor esses problemas.

4.2 Testes na API sintética Rest-Faults

Em [25], Zhang *et al.* apresentam um longo estudo sobre diferentes *fuzzers*, buscando compará-los em temos de eficácia na geração de casos de teste para APIs REST. Eles avaliam vários critérios, como cobertura de código, cobertura de mutação e a capacidade de identificar falhas específicas, destacando a importância de considerar essas variáveis ao comparar as ferramentas de *fuzzing*. Para realizar este estudo, eles construíram uma API sintética (*Rest-Faults*) que reúne um conjunto bastante diverso de falhas que foram utilizadas como base para comparação entre as 7 ferramentas. Essa API sintética também foi utilizada para testar o *fuzzer* desenvolvido nesse trabalho.

Seguem os tipos de falha que estão presentes na API sintética:

1. Exceção lançada na lógica de negócios da API, resultando em um código de status HTTP 500;
2. Um *endpoint* retorna uma carga útil onde um campo possui o tipo errado, com base no esquema;
3. Um *endpoint* retorna um código de status de sucesso (ou seja, na família 2xx) que não está declarado no esquema — definido pelo arquivo `schema.json`;

4. Uma carga útil requer que um campo numérico seja limitado a um certo valor máximo, mas essa restrição é ignorada na API;
5. Uma carga útil requer que um campo de *string* seja limitado a uma certa enumeração de valores, mas essa restrição é ignorada na API;
6. Uma carga útil requer que um campo booleano esteja presente, mas essa restrição é ignorada na API;
7. Um *endpoint* POST retorna um cabeçalho de localização (apontando para o novo recurso criado) que é inválido, resultando em um código de status 404 se for usado em uma chamada GET;
8. Um *endpoint* DELETE retorna o código de status 200, mas não exclui nada (ou seja, um GET na mesma URL ainda retornaria o recurso não excluído);
9. Um *endpoint* PUT aplica erroneamente uma atualização parcial (com semântica semelhante à RFC 7396 JSON Merge Patch) em vez de fazer uma substituição completa de recursos (o que pode ser detectado ao buscar o recurso modificado com um GET), conforme a semântica HTTP;
10. Um *endpoint* PUT aplica erroneamente uma modificação não idempotente. Portanto, chamar o mesmo *endpoint* mais de uma vez leva a resultados diferentes, que podem ser verificados ao buscar o recurso modificado com um GET. Lembre-se de que todos os verbos HTTP são idempotentes, exceto POST e PATCH.

De acordo com [25], dessas 10 falhas implantadas na API, a 1ª foi detectada por cinco dos sete *fuzzers*, enquanto a 2ª foi detectada por apenas um e a 3ª por apenas dois. As demais falhas não foram detectadas corretamente por nenhum *fuzzer*. Detectar uma falha, dentro do contexto do artigo, significa não só detectar o erro, mas também indicar o que faz o teste ser inválido.

Para a execução do *fuzzer*, o arquivo `schema.json` (um arquivo de definição aderente à especificação OpenAPI) disponibilizado pela API foi convertido para o formato de configuração aceito pelo *fuzzer* proposto nesse trabalho. O único ajuste necessário foi incluir o código de retorno HTTP 400 como padrão para dados inválidos e foram realizados 100 requisições para cada *endpoint*.

Durante a execução do *fuzzer*, foram detectados com sucesso erros nos *endpoints* com as falhas 1, 3 e 4. Os erros são registrados nos *logs*, cabendo ao usuário analisar e identificar suas causas, algo que é um possível ponto de melhoria: indicar para o usuário qual é o problema com o *request* feito.

O *endpoint* `/x` (que não possui parâmetros) com o método GET, continha a falha 1 de modo que qualquer request feito retorna um código 500 com uma mensagem de

erro, enquanto no esquema o código definido como esperado para requisições era 200. O *fuzzer* detectou corretamente que o código retornado pelo *endpoint* não era o esperado e registrou no *log* de erro que havia uma inconsistência.

As falhas 2 e 3 estão presentes no *endpoint* `/y`, que também não possui parâmetros do usuário. Esse *endpoint* é do tipo GET e retorna o código 201, enquanto no esquema o código declarado é 200. A falha 3 foi detectada corretamente pelo *fuzzer* de modo que o código 201 retornado foge do esperado 200 e é reportado nos *logs* de erro. O erro 2 não foi corretamente detectado porque o *fuzzer* não faz uma análise detalhada do conteúdo do retorno das requisições, mas apenas do código de status.

Por fim, o *endpoint* `/y` com o método POST possui as falhas 4, 5, 6 e 7 e possui 3 parâmetros:

1. Um inteiro com tamanho máximo de 100;
2. Um booleano;
3. Uma string que deve ser “FOO” ou “BAR”.

No esquema, o código esperado para a requisição é 201 indicando que um objeto foi corretamente criado, porém nenhuma verificação é realizada por parte da API e independente dos dados passados, o código de retorno é 201. O *fuzzer* detectou corretamente que o código de retorno foi o esperado para um dado válido mesmo com inteiros maiores que o máximo especificado, portanto o erro 4 foi encontrado pelo *fuzzer*. Os erros 5 e 6, também relacionados com a validação de dados de entrada, não foram corretamente detectados. Isso seria um ponto de melhoria para a ferramenta e não apresenta grandes desafios para uma possível implementação. Seria necessário adicionar o tipo de argumento *booleano* e *enum* como possibilidades no arquivo de configuração, bem como criar geradores para esses tipos de dados — com o núcleo do *fuzzer* já desenvolvido, essa implementação não seria muito trabalhosa.

O *fuzzer* não está preparado para detectar os erros 7, 8, 9 e 10, pois eles requerem a análise de sequências de requisições e respostas REST, algo que a ferramenta, atualmente, não faz. Esse também é um ponto de melhoria para a ferramenta, mas pode representar um grande desafio, já que não existe nenhum suporte para isso neste momento.

Em comparação com o RESTler, um *fuzzer* bastante popular e avaliado por Zhang *et al.*[25], o *fuzzer* desenvolvido neste trabalho apresenta características distintas que podem ser consideradas tanto vantagens quanto desvantagens, dependendo do contexto de uso e das necessidades específicas do usuário.

O *fuzzer* desenvolvido permite uma personalização mais precisa do usuário, uma característica particularmente útil para testadores que desejam exercer um controle deta-

lhado sobre os parâmetros de seus testes. Essa capacidade de personalização é maior do que a oferecida pelo RESTler, que, embora seja eficaz na geração automática de casos de teste, pode apresentar limitações quanto ao detalhamento e especificidade dos testes gerados. No entanto, essa maior flexibilidade do *fuzzer* desenvolvido vem com a necessidade de criar um arquivo de configuração específico, o que pode representar tempo maior de aprendizado inicial para novos usuários.

Outro ponto de destaque é a geração de *logs* detalhados pelo *fuzzer* deste estudo, o que é uma funcionalidade relevante para a análise pós-teste. Isso permite aos usuários uma compreensão profunda das falhas encontradas, embora também exija uma análise manual desses *logs* para identificar o tipo de falhas ocorridas. Por outro lado, o RESTler, ao integrar-se facilmente com ferramentas de teste e relatórios existentes, pode diminuir a necessidade de intervenção manual em algumas etapas do processo de teste, o que facilita sua inserção em fluxos de trabalho automatizados.

Além disso, enquanto o RESTler tem sua eficácia na geração de testes vinculada à precisão da documentação da API, o *fuzzer* desenvolvido não requer nenhum tipo de documentação — além do arquivo de configuração — proporcionando uma abordagem flexível em situações onde a documentação pode não estar completamente atualizada ou detalhada. Este aspecto é especialmente relevante em ambientes de desenvolvimento ágil, onde as mudanças podem ser frequentes e a documentação pode não acompanhar o ritmo dessas atualizações.

Em suma, enquanto o RESTler é conhecido por sua facilidade de uso e integração, o *fuzzer* desenvolvido neste trabalho destaca-se pela sua capacidade de personalização. A escolha entre essas ferramentas dependerá das necessidades específicas dos testadores e do contexto em que serão aplicadas, com cada ferramenta oferecendo conjuntos distintos de vantagens.

5 CONCLUSÃO

Este trabalho teve como objetivo central desenvolver um *fuzzer* especializado para avaliação de segurança em APIs REST. A importância da segurança em APIs é cada vez mais evidente à medida que essas interfaces se tornam vitais para a comunicação entre diferentes sistemas. No entanto, as ferramentas de *fuzzing* tradicionais muitas vezes não são adequadas para lidar com as peculiaridades das APIs REST, como mecanismos de autenticação e formatos de entrada variados.

A ferramenta desenvolvida neste trabalho busca preencher essa lacuna, permitindo a realização de testes que simulam diversos cenários. Com a capacidade de lidar com entradas válidas e inválidas, bem como lidar com diferentes formas de autenticação, o *fuzzer* demonstra ser uma ferramenta útil para identificar falhas em APIs REST.

Durante os testes realizados, no ConsistencIA e Rest-Faults, foram identificadas diversas falhas de segurança, como falta de validação de parâmetros e tratamento inadequado de erros. Esses resultados destacam a importância da utilização de ferramentas especializadas para garantir a segurança das APIs.

Assim como em outras áreas de segurança cibernética, a detecção de vulnerabilidades em APIs REST é um desafio em constante evolução. Portanto, deve-se continuar investindo em pesquisas e desenvolvendo técnicas mais avançadas para detecção e mitigação de ameaças.

Para trabalhos futuros, recomenda-se explorar técnicas de defesa mais sofisticadas, que possam distinguir de forma eficaz entre exemplos adversários e dados normais. Além disso, seria interessante investigar variações nos parâmetros dos modelos de *fuzzing* para avaliar sua robustez em diferentes contextos e aplicar ataques mais elaborados, especialmente projetados para APIs REST.

Esses esforços contribuirão para fortalecer a segurança e a confiabilidade das APIs REST, garantindo uma melhor proteção contra ameaças cibernéticas e promovendo um ambiente digital mais seguro e confiável para todos os usuários.

REFERÊNCIAS

[1] SEGURA, S. et al. Metamorphic Testing of RESTful Web APIs. *IEEE Transactions on Software Engineering*, v. 44, n. 11, p. 1083–1099, nov. 2018. ISSN 1939-3520. Conference Name: IEEE Transactions on Software Engineering.

[2] IDRIS, M.; SYARIF, I.; WINARNO, I. Development of Vulnerable Web Application Based on OWASP API Security Risks. In: *2021 International Electronics Symposium (IES)*. [S.l.: s.n.], 2021. p. 190–194.

[3] DíAZ-ROJAS, J. A. et al. Web API Security Vulnerabilities and Mitigation Mechanisms: A Systematic Mapping Study. In: *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*. [S.l.: s.n.], 2021. p. 207–218.

[4] ATLIDAKIS, V.; GODEFROID, P.; POLISHCHUK, M. *REST-ler: Automatic Intelligent REST API Fuzzing*. arXiv, 2018. ArXiv:1806.09739 [cs]. Disponível em: <http://arxiv.org/abs/1806.09739>.

[5] TSAI, C.-H.; TSAI, S.-C.; HUANG, S.-K. *REST API Fuzzing by Coverage Level Guided Blackbox Testing*. arXiv, 2021. ArXiv:2112.15485 [cs]. Disponível em: <http://arxiv.org/abs/2112.15485>.

[6] ATLIDAKIS, V. et al. *Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations*. arXiv, 2020. ArXiv:2005.11498 [cs]. Disponível em: <http://arxiv.org/abs/2005.11498>.

[7] THAYER, K.; CHASINS, S. E.; KO, A. J. A Theory of Robust API Knowledge. *ACM Transactions on Computing Education*, v. 21, n. 1, p. 8:1–8:32, jan. 2021. Disponível em: <https://dl.acm.org/doi/10.1145/3444945>.

[8] SERVICES, I. A. W. *What is an API? - Application Programming Interfaces Explained*. 2023. Acessado em: 2023-08-30. Disponível em: <https://aws.amazon.com/what-is/api/>.

[9] HAT, I. R. *What is an API?* 2023. Acessado em: 2023-08-30. Disponível em: <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces>.

[10] IDRIS, M.; SYARIF, I.; WINARNO, I. Web Application Security Education Platform Based on OWASP API Security Project. *EMITTER International Journal of Engineering Technology*, p. 246–261, dez. 2022. ISSN 2443-1168. Disponível em: <https://emitter.pens.ac.id/index.php/emitter/article/view/705>.

[11] WANG, X. et al. Exploring Efficient Microservice Level Parallelism. In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. [S.l.: s.n.], 2022. p. 223–233. ISSN: 1530-2075.

[12] AL-MASRI, E. Enhancing the Microservices Architecture for the Internet of Things. In: *2018 IEEE International Conference on Big Data (Big Data)*. [S.l.: s.n.], 2018. p. 5119–5125.

[13] LAURET, A. *The Design of Web APIs*. [S.l.]: Manning, 2019. ISBN 978-1-61729-510-2.

[14] FOUNDATION, O. *API Security Project*. 2023. Acessado em: 2023-09-02. Disponível em: <https://owasp.org/www-project-api-security/>.

[15] FOUNDATION, O. *OWASP Top Ten*. 2023. Acessado em: 2023-09-02. Disponível em: <https://owasp.org/Top10/>.

[16] SUTTON, M.; GREENE, A.; AMINI, P. *Fuzzing: Brute Force Vulnerability Discovery*. [S.l.]: Addison-Wesley Professional, 2007.

[17] WSTG - v4.2 | OWASP Foundation. Disponível em: <https://owasp.org/www-project-web-security-testing-guide/v42/>.

[18] SHAHBAZI, A.; MILLER, J. Black-Box String Test Case Generation through a Multi-Objective Optimization. *IEEE Transactions on Software Engineering*, v. 42, n. 4, p. 361–378, abr. 2016. ISSN 1939-3520. Conference Name: IEEE Transactions on Software Engineering. Disponível em: <https://ieeexplore.ieee.org/document/7293669>.

[19] LI, J.; ZHAO, B.; ZHANG, C. Fuzzing: a survey. *Cybersecurity*, v. 1, n. 1, p. 6, jun. 2018. ISSN 2523-3246. Disponível em: <https://doi.org/10.1186/s42400-018-0002-y>.

[20] LIANG, H. et al. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, v. 67, n. 3, p. 1199–1218, set. 2018. ISSN 1558-1721. Conference Name: IEEE Transactions on Reliability.

[21] MILLER, B. P.; FREDRIKSEN, L.; SO, B. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, v. 33, n. 12, p. 32–44, dez. 1990. ISSN 0001-0782. Disponível em: <https://dl.acm.org/doi/10.1145/96267.96279>.

[22] BOEHME, M.; CADAR, C.; ROYCHOUDHURY, A. Fuzzing: Challenges and Reflections. *IEEE Software*, v. 38, n. 3, p. 79–86, maio 2021. ISSN 1937-4194. Conference Name: IEEE Software.

[23] CHEN, C. et al. A systematic review of fuzzing techniques. *Computers & Security*, v. 75, p. 118–137, 2018. ISSN 0167-4048. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167404818300658>.

[24] GODEFROID, P.; HUANG, B.-Y.; POLISHCHUK, M. Intelligent REST API data fuzzing. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (ESEC/FSE 2020), p. 725–736. ISBN 978-1-4503-7043-1. Disponível em: <https://dl.acm.org/doi/10.1145/3368089.3409719>.

[25] ZHANG, M.; ARCURI, A. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology*, v. 32, n. 6, p. 144:1–144:45, set. 2023. ISSN 1049-331X. Disponível em: <https://dl.acm.org/doi/10.1145/3597205>.

[26] SWAGGER. *What Is Swagger*. 2023. Acessado em: 2023-11-20. Disponível em: <https://swagger.io/docs/specification/2-0/what-is-swagger/>.

[27] LARANJEIRO, N.; AGNELO, J.; BERNARDINO, J. A Black Box Tool for Robustness Testing of REST Services. *IEEE Access*, v. 9, p. 24738–24754, 2021. ISSN 2169-3536. Conference Name: IEEE Access. Disponível em: <https://ieeexplore.ieee.org/document/9344640>.

[28] ARCURI, A. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. [s.n.], 2018. p. 394–397. ArXiv:1901.04472 [cs]. Disponível em: <http://arxiv.org/abs/1901.04472>.

[29] MARTIN-LOPEZ, A.; SEGURA, S.; RUIZ-CORTéS, A. RESTest: automated black-box testing of RESTful web APIs. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2021. (ISSTA 2021), p. 682–685. ISBN 978-1-4503-8459-9. Disponível em: <https://dl.acm.org/doi/10.1145/3460319.3469082>.

[30] WU, H. et al. Combinatorial testing of RESTful APIs. In: *Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022. (ICSE '22), p. 426–437. ISBN 978-1-4503-9221-1. Disponível em: <https://dl.acm.org/doi/10.1145/3510003.3510151>.

[31] VIGLIANISI, E.; DALLAGO, M.; CECCATO, M. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. [s.n.], 2020. p. 142–152. ISSN: 2159-4848. Disponível em: <https://ieeexplore.ieee.org/document/9159077>.

[32] HATFIELD-DODDS, Z.; DYGALO, D. Deriving Semantics-Aware Fuzzers from Web API Schemas. In: *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. [s.n.], 2022. p. 345–346. ISSN: 2574-1926. Disponível em: <https://ieeexplore.ieee.org/document/9793781>.

[33] LIN, J. et al. *foREST: A Tree-based Approach for Fuzzing RESTful APIs*. arXiv, 2022. ArXiv:2203.02906 [cs]. Disponível em: <http://arxiv.org/abs/2203.02906>.

[34] LYU, C. et al. *MINER: A Hybrid Data-Driven Approach for REST API Fuzzing*. arXiv, 2023. ArXiv:2303.02545 [cs]. Disponível em: <http://arxiv.org/abs/2303.02545>.

[35] CURL. *Curl*. 2024. Acessado em: 2024-03-21. Disponível em: <https://curl.se/>.