UNIVERSIDADE ESTADUAL DO RIO GRANDE DO SUL
CURSO SUPERIOR DE ENGENHARIA DE COMPUTAÇÃO

BRUNA QUINHONES DE MELO

DESENVOLVIMENTO DE UM SOFTWARE FUZZER DE ENTRADA ALEATÓRIA PARA IDENTIFICAR VULNERABILIDADES EM UMA APLICAÇÃO DE MARKETPLACE

GUAÍBA
2023

BRUNA QUINHONES DE MELO

DESENVOLVIMENTO DE UM SOFTWARE FUZZER DE ENTRADA ALEATÓRIA PARA IDENTIFICAR VULNERABILIDADES EM UMA APLICAÇÃO DE MARKETPLACE

Trabalho de Conclusão de Curso, apresentado ao Curso de graduação em Engenharia de Computação da Universidade Estadual do Rio Grande do Sul como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Profa. Dra. Margrit Reni Krug
Orientadora

GUAÍBA
2023

BRUNA QUINHONES DE MELO

DESENVOLVIMENTO DE UM SOFTWARE FUZZER DE ENTRADA ALEATÓRIA PARA IDENTIFICAR VULNERABILIDADES EM UMA APLICAÇÃO DE MARKETPLACE

Trabalho de Conclusão de Curso, apresentado ao Curso de graduação em Engenharia de Computação da Universidade Estadual do Rio Grande do Sul como requisito para a obtenção do título de Bacharel em Engenharia de Computação.

Aprovado em ....../....../.......

BANCA EXAMINADORA:

____

Profa. Dra. Adriane Parraga
Universidade Estadual do Rio Grande do Sul

____

Prof. Dr. Celso Maciel da Costa
Universidade Estadual do Rio Grande do Sul

____

Profa. Dra. Margrit Reni Krug
Orientadora

AGRADECIMENTOS

Estar hoje escrevendo meus agradecimentos é emocionante, é marcante. A minha jornada acadêmica foi marcada por altos e baixos, muita resiliência e noites em claro fazendo trabalhos (como este, por exemplo). Brincadeiras à parte, esta jornada foi marcada por pessoas extraordinárias, que contribuíram muito para o meu sucesso e crescimento.

Em primeiro lugar, minha eterna gratidão à minha família, que sempre me deu apoio e base sólida para ser a pessoa que sou hoje, e alcançar essa tão esperada conquista. Não tenho palavras pra dizer o quanto vocês foram e são importantes em toda a minha trajetória. (Pai Magnus, Pai João, Vó, Vô, Tia Gabriela, Tio Lucio).

Ao meu pai, em especial, agradeço por sempre acreditar em mim. Agradeço por me ensinar muito sobre responsabilidade, força, altruísmo e resiliência. Sem a tua motivação, nada disso seria, hoje, realidade. Tu és a minha grande inspiração.

À minha namorada, Bianca, que me incentiva, me ouve, me ajuda, e quando estou cansada, me leva até Guaíba (foi o fim do conceito E no meu histórico).

Minha gratidão eterna a todos os professores e funcionários da unidade de Guaíba, que tive o prazer de conviver nesses anos. Em especial, gostaria de estender os meus agradecimentos aos professores Celso, Margrit e Adriane, que estiveram comigo desde o início, e me proporcionaram ensinamentos muito além da sala de aula.

Por último, e não menos importante, agradeço aos colegas e amigos que fiz ao longo desses anos - "Ninguém faz nada grande sozinho".

A todos vocês, meu mais profundo agradecimento por fazerem parte desta significativa etapa da minha vida.

SUMÁRIO

RESUMO **6**
ABSTRACT **7**
1. INTRODUÇÃO **8**
1.1. Justificativa 10
2. REVISÃO BIBLIOGRÁFICA **11**
2.1. Teste de Software 11
2.1.1. Níveis de teste de software 13
2.2. Teste de Correção 13
2.2.1. Teste Caixa Branca 14
2.2.2. Teste Caixa Preta 14
2.3. Teste de Segurança 15
2.4. Teste de Fuzzing 16
2.4.1. Fuzzing de Caixa Branca 17
2.4.2. Fuzzing Baseado em Modelos 17
2.5. Trabalhos Relacionados 18
2.5.1. SonarQube 18
2.5.2. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. 19
3. METODOLOGIA **21**
4. PROJETO **23**
4.1. Descrição Geral do Projeto 23
4.2. Diagrama Geral do Projeto 23
4.3. Apresentação do Software Fuzzer 24
5. RESULTADOS **32**
5.1. Primeira Entrevista para Levantamento de Requisitos 32
5.2. Segunda Entrevista - Apresentação inicial do Software Fuzzer 33
5.3. Apresentação e Análise do Relatório Final Gerado 34
5.4. Comparação com Trabalhos Relacionados 35
6. CONCLUSÃO **36**
7. REFERÊNCIAS **37**

RESUMO

Os *marketplaces*, que desempenham um papel significativo no cenário do comércio eletrônico, enfrentam desafios substanciais devido à crescente ameaça de crimes cibernéticos. Este estudo aborda essa problemática ao propor a implementação de um software *Fuzzer* de entrada aleatória para realizar testes e identificar vulnerabilidades em uma aplicação de *marketplace*. A técnica de fuzzing revelou-se eficaz durante os testes, utilizando um gerador de caracteres aleatórios para expor diversas fragilidades no sistema, incluindo falhas de segurança. A análise qualitativa dos resultados demonstra que a aplicação contínua e integrada do software *Fuzzer* assegura a correção proativa de vulnerabilidades, tanto no código atual quanto em futuras atualizações.

Palavras-chave: Fuzzing, Testes de Segurança, Segurança de Software, Fuzzer.

ABSTRACT

Marketplaces, playing a significant role in the e-commerce landscape, face substantial challenges due to the growing threat of cybercrimes. This study addresses this issue by proposing the implementation of a random input Fuzzer software to conduct tests and identify vulnerabilities in a marketplace application. The fuzzing technique proved effective during testing, using a random character generator to expose various weaknesses in the system, including security flaws. Qualitative analysis of the results demonstrates that the continuous and integrated application of Fuzzer software ensures proactive correction of vulnerabilities in both the current code and future updates.

Keywords: Fuzzing, Security Testing, Software Security, Fuzzer.

1. INTRODUÇÃO

Com o avanço tecnológico e a transição crescente de estabelecimentos físicos para plataformas *online*, a segurança do software tornou-se uma preocupação cada vez mais relevante. A pandemia de COVID-19 teve um impacto significativo no crescimento dos *marketplaces*, devido ao aumento das compras *online* decorrente das medidas de distanciamento social. De acordo com relatórios da Associação Brasileira de Comércio Eletrônico, em 2020, os *marketplaces* representaram 78% do faturamento total do *e-commerce* no país.(ABCOMM, 2021). Além disso, em relação a essa mudança de mercado, conforme apontado por Sharif e Mohammed, houve um aumento de 600% nos crimes cibernéticos, também influenciado pela pandemia. Segundo a revista Cyber Security Ventures, espera-se que esse tipo de crime cresça em torno de 15% ao ano nos próximos cinco anos. Diante dessa realidade, torna-se essencial adotar práticas e medidas de segurança de software, uma vez que vulnerabilidades nesse aspecto podem resultar em consequências graves, como perda financeira e violação da privacidade do usuário.(SHARIF e MOHAMMED, 2022).

Considerando o aumento dos riscos cibernéticos associados à expansão das plataformas *online*, a adoção da técnica de *fuzzing* durante o processo de desenvolvimento de software destaca-se como uma medida eficaz de segurança, visando assegurar a resiliência e a robustez do software. Conforme a definição de Barton Miller em 1995, o programa *fuzz* consiste em um gerador de caracteres aleatórios que, por meio de testes de caixa preta, busca identificar vulnerabilidades e falhas de segurança exploráveis por invasores. Ao aplicar o fuzzing de forma contínua e integrada em uma aplicação de *marketplace*, é possível garantir a correção proativa de vulnerabilidades no código atual e em futuras atualizações do software.(MILLER, 2008).

Neste contexto, o objetivo principal deste trabalho foi desenvolver um software *Fuzzer* de entrada aleatória para realizar testes e identificar vulnerabilidades específicas em uma aplicação de *marketplace*. Para alcançar o objetivo principal, os seguintes objetivos específicos foram necessários: levantar os requisitos da aplicação; identificar as funcionalidades críticas a serem testadas e os requisitos de segurança que deveriam ser considerados, para possibilitar a definição dos componentes que deveriam ser testados, assim como a definição dos casos de uso que seriam abrangidos, e; avaliar a aplicação proposta com um projeto piloto.

Em suma, este estudo teve como propósito aprimorar a segurança e confiabilidade da aplicação de *marketplace* por meio da implementação de um software *Fuzzer* de entrada aleatória. Ademais, espera-se que este trabalho contribua para o avanço da área de segurança de software e na aplicação da técnica de *Fuzzing*.

Ressalta-se que a empresa objeto deste estudo é uma empresa norte-americana do segmento de comércio eletrônico, categorizada, de acordo com a definição de porte de estabelecimentos segundo o número de empregados do Sebrae, como uma empresa de médio porte. Vale salientar que a empresa em

questão não autorizou a divulgação de seu nome neste trabalho, sendo, portanto, tratada anonimamente como Empresa XYZ. Essa escolha visa preservar a confidencialidade da organização, permitindo uma análise mais franca e objetiva do projeto implementado.

1.1. Justificativa

O cenário pós pandêmico gerou um aumento incontestável no uso de plataformas de compras online, associado à crescente de crimes e ameaças cibernéticas. Nesse contexto, incorporar práticas para a identificação e prevenção de vulnerabilidades proativas durante o desenvolvimento e teste de software torna-se essencial, a fim de evitar ataques que possam gerar perdas financeiras e vazamento de dados.(ABCOMM, 2021).

A técnica de *fuzzing* de entradas aleatórias surge como uma abordagem eficiente e abrangente para a identificação de falhas e melhorias na qualidade do software, uma vez que o gerar uma grande quantidade de entradas, a técnica permite explorar uma ampla gama de cenários teste, que, consequentemente, aumenta a cobertura, expondo o sistema a diferentes situações que podem revelar comportamentos inesperados.(SHARIF e MOHAMMED, 2022).

O projeto teve por foco, portanto, projetar um software Fuzzer e integrá-lo ao processo de desenvolvimento e teste do produto voltado ao comércio eletrônico da Empresa XYZ. Além disso, buscou-se avaliar a sua eficácia a partir dos resultados obtidos em cada execução, para que então fosse possível minimizar os riscos de falhas críticas e preservar a integridade dos dados e das funcionalidades do sistema.

2. REVISÃO BIBLIOGRÁFICA

2.1. Teste de Software

Segundo Myers (2011), o teste de software é um processo no qual um programa é executado com o objetivo de encontrar defeitos. É por meio do teste que problemas relacionados aos requisitos, erros de programação e comportamentos inesperados são identificados e corrigidos antes da implantação no ambiente de produção. O teste de software desempenha um papel fundamental na garantia da qualidade do software, assim sendo, o autor ressalta que a qualidade do software não é determinada apenas pela ausência de defeitos, mas também por uma combinação complexa de fatores, como: usabilidade, eficiência, confiabilidade, segurança e manutenibilidade. Resumidamente, Neto (2007, p. 54) entende que:

> [...] testar um software significa verificar através de uma execução controlada se o seu comportamento corre de acordo com o especificado. O objetivo principal desta tarefa é revelar o número máximo de falhas dispondo do mínimo de esforço, ou seja, mostrar aos que desenvolvem se os resultados estão ou não de acordo com os padrões estabelecidos.

Existem diferentes modelos e padrões que auxiliam na compreensão e avaliação da qualidade do software. O Modelo proposto por McCall, Richards e Walters apresenta uma categorização útil dos fatores que afetam a qualidade do software. (PRESSMAN, 2005). Esse modelo divide os fatores de qualidade em 11 categorias, proporcionando uma visão abrangente dos aspectos a serem considerados.

Um padrão relevante, relacionado às boas práticas de projeto de desenvolvimento de software que levam a um produto de qualidade refere-se à norma ISO 9126 (PRESSMAN, 2005), a qual estabelece diretrizes internacionais para a avaliação da qualidade do software. Essa norma abrange características como: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade, fornecendo critérios e indicadores para a análise e medição da qualidade do software.

Conforme Pressman (2005), o uso de normas e modelos na intenção de fornecer direcionamentos para o planejamento, execução e avaliação dos testes, permite a definição de uma abordagem abrangente e estruturada para garantir a qualidade do software desenvolvido.

A busca pela qualidade do software requer aprimoramento dos processos de produção, identificando e analisando causalmente os defeitos presentes, de acordo com Kalinowski (2007). Nesse contexto, é importante definir as diferenças entre erros, falhas e defeitos. O IEEE (*Institute of Electrical and Engineer*s) (1990) conceitua-os como:

- Defeitos são falhas ou anomalias no software que podem levar a comportamentos incorretos, indesejados ou não conformes com os

requisitos especificados. Eles podem ser originados por erros na lógica ou problemas de implementação, como por exemplo uma instrução ou comando incorreto;

- Erros ocorrem quando há discrepâncias entre o comportamento real do software e o esperado. Essas discrepâncias podem ser causadas por falhas na lógica, implementação inadequada ou outros problemas relacionados ao software, resultando em comportamentos indesejados ou incorretos, e;
- Falhas podem ser entendidas como uma deficiência ou mau funcionamento no sistema ou componente que impede sua operação de acordo com as expectativas definidas. Vale ressaltar que falhas são causadas por diversos erros, porém nem todo erro causa uma falha.

Neto (2007, p.55) também destaca que existem elementos essenciais que compõem e auxiliam na atividade de teste, sendo eles:

> Caso de Teste: descreve uma condição particular a ser testada e é composto por valores de entrada, restrições para a sua execução e um resultado ou comportamento esperado (CRAIG e JASKIEL, 2002).
>
> Procedimento de Teste: é uma descrição dos passos necessários para executar um caso (ou um grupo de casos) de teste (CRAIG e JASKIEL, 2002).
>
> Critério de Teste: serve para selecionar e avaliar casos de teste de forma a aumentar as possibilidades de provocar falhas ou, quando isso não ocorre, estabelecer um nível elevado de confiança na correção do produto (ROCHA et al., 2001). Os critérios de teste podem ser utilizados como:
>
> - Critério de Cobertura dos Testes: permitem a identificação de partes do programa que devem ser executadas para garantir a qualidade do software e indicar quando o mesmo foi suficientemente testado (RAPPS e WEYUKER, 1982). Ou seja, determinar o percentual de elementos necessários por um critério de teste que foram executados pelo conjunto de casos de teste;
> - Critério de Adequação de Casos de Teste: Quando, a partir de um conjunto de casos de teste T qualquer, ele é utilizado para verificar se T satisfaz os requisitos de teste estabelecidos pelo critério. Ou seja, este critério avalia se os casos de teste definidos são suficientes ou não para avaliação de um produto ou uma função (ROCHA et al., 2001), e;
> - Critério de Geração de Casos de Teste: quando o critério é utilizado para gerar um conjunto de casos de teste T adequado para um produto ou função, ou seja, este critério define as regras e diretrizes para geração dos casos de teste de um produto que esteja de acordo com o critério de adequação definido anteriormente (ROCHA et al., 2001).

Além disso, a aplicação de diferentes níveis e tipos de testes, propostos por Beizer (2003) e Myers (2011), desempenham um papel crucial na identificação de defeitos em várias camadas do sistema. Cada nível aborda diferentes aspectos do software, permitindo a detecção e correção de problemas em estágios específicos

do processo de desenvolvimento, contribuindo para a melhoria contínua e aprimoramento do produto final.

2.1.1. Níveis de teste de software

A proposta dos autores Beizer (2003) e Myers (2011) de dividir os testes em diferentes níveis, consiste em etapas distintas e progressivas para avaliar um sistema em diversos níveis de abstração. Essa abordagem estruturada e abrangente visa assegurar a qualidade e a confiabilidade do software ao longo de cada fase do desenvolvimento. Beizer (2003) apresenta a seguinte classificação e definição dos níveis de teste de software:

- Teste de unidade: concentra-se na verificação individual dos componentes e módulos da aplicação, explorando falhas ocasionadas por defeitos de lógica e implementação em métodos ou pequenos trechos de código;
- Teste de integração: avalia a interação e a integração entre os diferentes módulos, verificando se funcionam corretamente em conjunto;
- Teste de sistema: esse teste é realizado no sistema completo, testando sua funcionalidade e desempenho como um todo, e;
- Teste de aceitação: nesse nível, o teste é realizado por usuários finais, a fim de verificar se o software atende aos requisitos e expectativas estabelecidos.

Myers (2011) propôs uma abordagem semelhante à de Beizer (2003), com os níveis de teste, sendo eles: teste de unidade, teste de integração, teste de sistema e teste de aceitação. Além disso, o autor também inclui o teste de regressão, que irá verificar se as alterações ou correções realizadas não introduziram novos erros no sistema.

Essa abordagem estruturada e progressiva possibilita avaliar o sistema em vários níveis de abstração, e envolve a aplicação de diferentes técnicas de teste de software associadas em cada nível. De acordo com Khan (2010), as técnicas de teste de software mais prevalentes são: teste de correção, teste de segurança, teste de performance e teste de confiabilidade. Nas próximas seções abordarão com mais detalhes as técnicas de correção e de segurança.

2.2. Teste de Correção

Segundo Khan (2010, p. 12), "O teste de correção indica o comportamento correto do sistema em oposição ao incorreto." Ainda segundo o autor, tanto uma perspectiva caixa branca quanto de caixa preta podem ser adotadas no teste de software, uma vez que o testador pode ou não ter conhecimento detalhado do módulo de software em teste. Apesar de estar sendo abordado nesta seção, a utilização das técnicas de caixa branca e caixa preta não estão limitadas apenas para o teste de correção.

2.2.1. Teste Caixa Branca

Segundo Neto (2007), o teste de caixa branca, também conhecido como teste estrutural, é uma técnica de teste de software em que o testador possui conhecimento detalhado da estrutura interna do código-fonte, e seja capaz de criar casos de teste que garantam cobertura dos caminhos possíveis e ramificações de decisão. Essa técnica é utilizada para avaliar aspectos do software como: teste de condição, teste de fluxo de dados, teste de ciclos e teste de caminhos lógicos (PRESSMAN, 2005).

Existem prós e contras no teste caixa branca, segundo Khan (2010), são eles:

- Prós:
  - Efeitos colaterais são benéficos;
  - Erros em códigos ocultos são revelados;
  - Aproxima-se da partição feita pela equivalência de execução, e;
  - O desenvolvedor fornece cuidadosamente justificativas sobre a implementação.
- Contras:
  - É muito caro, e;
  - Pode deixar passar casos omitidos no código.

2.2.2. Teste Caixa Preta

Khan (2010) define o teste de caixa preta, também conhecido como teste funcional, como uma técnica de teste de software que se concentra na funcionalidade externa do sistema, sem considerar sua estrutura interna. Nesse tipo de teste, o testador não tem conhecimento detalhado do código-fonte, mas se baseia nas especificações e nos requisitos do software. Ainda segundo o autor, o objetivo é testar o quão bem o componente se conforma aos requisitos publicados, garantindo a integridade das informações externas. Existem várias vantagens e desvantagens do teste de caixa preta, são elas (KHAN, 2010, p. 13):

Vantagens:

1. O testador de caixa preta não possui "ligação" com o código;
2. A percepção do testador é muito simples;
3. Programador e testador são independentes um do outro, e;
4. Mais eficaz em unidades maiores de código do que o teste de caixa branca.

Desvantagens:

1. É difícil projetar casos de teste sem especificações claras;
2. Apenas um pequeno número de entradas possíveis pode ser testado na prática, e;
3. Algumas partes do backend não são testadas de forma alguma.

2.3. Teste de Segurança

O teste de segurança, começou a ganhar popularidade no início dos anos 2000 através dos autores Anderson e McGraw (2000) por conta do crescente número de incidentes cibernéticos e violações de dados que afetaram organizações em todo o mundo, e a necessidade de uma abordagem proativa para identificar e mitigar vulnerabilidades em sistemas e aplicativos. De acordo com Potter e McGraw (2004), o teste de segurança pode ser dividido em:

- Teste de Segurança Funcional: verifica se as funções de segurança do software estão implementadas corretamente e em conformidade com os requisitos de segurança. Os requisitos de segurança do software incluem principalmente confidencialidade, integridade, disponibilidade, autenticação, controle de acesso, proteção de privacidade, gerenciamento de segurança, etc, e;
- Teste de Vulnerabilidade de Segurança: este tipo de teste tem como objetivo descobrir vulnerabilidades de segurança que possam ser exploradas por invasores, resultando em um estado de insegurança. Vulnerabilidade refere-se às falhas no design, implementação, operação e gerenciamento do sistema.

Nesse contexto, Khan (2010) descreve os cinco conceitos principais que são abordados pelo teste de segurança, visando garantir a confidencialidade, integridade, autenticação, disponibilidade e autorização das informações, são elas segundo Khan (2010, p. 15):

1. Confidencialidade: Através do teste de segurança, garantimos a confidencialidade do sistema, ou seja, nenhuma divulgação de informações para uma parte desconhecida que não seja o destinatário pretendido;
2. Integridade: Por meio do teste de segurança, mantemos a integridade do sistema, permitindo que o receptor determine que as informações que ele está recebendo são corretas;
3. Autenticação: O teste de segurança mantém a integridade das autenticações do sistema;
4. Disponibilidade: As informações são sempre mantidas disponíveis para o pessoal autorizado quando necessário e garantem que os serviços de informação estejam prontos para uso sempre que necessário;
5. Autorização: O teste de segurança garante que apenas o usuário autorizado possa acessar as informações ou serviços específicos. O controle de acesso é um exemplo de autorização. (KHAN, 2010, p. 15)

Esses conceitos são colocados em prática por meio de métodos de teste de segurança de software. Os principais métodos incluem teste de segurança formal,

que envolve a construção de modelos matemáticos do software e a aplicação de provas de teorema ou verificação de modelos para garantir a validade e segurança do código; teste de segurança baseado em modelos, onde são criados modelos do comportamento e estrutura do software para derivar casos de teste; teste de segurança baseado em injeção de falhas, que simula comportamentos anormais injetando falhas em pontos de interação do software; teste de *fuzzing*, que consiste na injeção de dados aleatórios no programa para identificar falhas de segurança; teste de varredura de vulnerabilidades, que realiza análises automatizadas em busca de vulnerabilidades conhecidas; teste baseado em propriedades, que verifica se o software atende a propriedades de segurança especificadas; teste de segurança baseado em caixa branca, que realiza análise estática ou dinâmica do código em busca de vulnerabilidades; e teste de segurança baseado em risco, que combina análise de risco e teste de penetração para identificar e mitigar vulnerabilidades de alto risco (POTTER e MCGRAW, 2004).

Este trabalho foca no método de *fuzzing*, o qual será apresentado em detalhes na seção seguinte. Em especial, abordou-se na eficácia desse método na detecção de vulnerabilidades em sistemas de software.

2.4. Teste de Fuzzing

Fuzzing, também conhecido como teste de *fuzz*, é uma técnica de teste de segurança que envolve o fornecimento de entradas inválidas ou inesperadas. O primeiro artigo sobre *fuzzing* foi publicado em 1990 por Barton Miller, da Universidade de Wisconsin. Ele foi o responsável pelo desenvolvimento inicial da técnica de *fuzzing* associada à abordagem de caixa-preta em 1989. O objetivo do artigo era testar a confiabilidade dos utilitários UNIX para explorar suas vulnerabilidades, e revelou que mais de 24% dos mais de 90 programas testados falharam com entradas inválidas. (POTTER e MCGRAW, 2004)

Miller (2020) reproduziu esse mesmo estudo e publicou um artigo na revista IEEE *Transactions on Software Engineering* intitulado "*The Relevance of Classic Fuzz Testing: Have We Solved This One*?". Nesse estudo, ele testou os utilitários UNIX nos sistemas Linux, FreeBSD e MacOS. Mesmo após três décadas, os *fuzzers* desenvolvidos para realizar os testes mostraram-se capazes de explorar e identificar 24 falhas nos utilitários propostos, reafirmando a eficácia dos fuzzers de abordagem simples na detecção de falhas. (MILLER, ZHANG, HEYMANN, 2020, p. 2028-2029).

Embora eficazes, os métodos tradicionais de *fuzzing* podem apresentar limitações em relação à eficácia dos testes. Segundo Pan (2013), a fim de superar essas limitações, surgiu o conceito de *fuzzing* inteligente, uma abordagem avançada que combina diferentes tecnologias para aprimorar a detecção de falhas de segurança. Pan (2013) classifica o *fuzzing* inteligente como caixa branca, evolutivo e baseado em modelo.

Tanto o *fuzzing* de caixa branca quanto de evolução exigem um profundo conhecimento em depuração de software e engenharia reversa, e as práticas de

teste geralmente são tediosas e consomem tempo. Em contraste, o *fuzzing* baseado em modelos define casos de teste com base em modelos de entrada e simplifica o processo de análise geral. Portanto, a abordagem baseada em modelos é amplamente adotada, e a maioria dos *fuzzers* bem-sucedidos inclui recursos para modelar a estrutura dos dados que serão gerados. (PAN, 2013, p. 645).

2.4.1. Fuzzing de Caixa Branca

A técnica de *fuzzing* caixa branca combina *fuzz testing* com geração dinâmica de testes, executando o programa com entradas iniciais bem formadas, de forma concreta e simbólica, e utilizando a execução simbólica para criar restrições nas entradas do programa. Essas restrições são usadas para gerar novas entradas que percorrem diferentes caminhos de controle do programa em teste. O objetivo é descobrir bugs de forma eficiente, utilizando várias heurísticas de busca (GODEFROID, KIEZUN e LEVIN, 2008).

De acordo com Godefroid, Kiezun e Levin (2008), o *fuzzing* de caixa branca tem sido eficaz na detecção de vulnerabilidades de segurança em várias aplicações. No entanto, ele enfrenta limitações ao testar aplicações com entradas altamente estruturadas, como compiladores e interpretadores, devido ao grande número de caminhos de controle e à possibilidade de derrotar a execução simbólica nos estágios iniciais de processamento. Essas limitações podem dificultar o alcance de partes mais profundas da aplicação.

Pan (2013) avalia a abordagem caixa branca como não eficaz, considerando que o número de caminhos de controle viáveis pode ser infinito. Godefroid, Kiezun e Levin (2008) complementam que além da explosão de caminhos, a própria execução simbólica pode ser derrotada nos primeiros estágios de processamento.

2.4.2. Fuzzing Baseado em Modelos

Schieferdecker, Grossmann e Schneider (2012) explicam que a ideia básica do teste baseado em modelos é gerar automaticamente casos de teste a partir de um ou mais modelos do sistema em teste, em vez de criá-los manualmente. Enquanto a automação de testes substitui a execução manual de testes por scripts automatizados, o teste baseado em modelos substitui o design manual de testes pela geração automatizada de testes. Os autores destacam a escassez de pesquisas sobre teste de segurança baseado em modelos, apesar da existência de vários artigos abordando o assunto.

No contexto do teste de segurança, *fuzzers* baseados em blocos e baseados em modelos desempenham um papel crucial ao utilizar seu conhecimento sobre a estrutura das mensagens para gerar sistematicamente mensagens que contenham tanto dados válidos quanto inválidos (SCHIEFERDECKER, GROSSMANN e SCHNEIDER, 2012). Ao contrário da abordagem de randomização completa dos *fuzzers* tradicionais, esses aproveitam seu conhecimento sobre a estrutura das mensagens para aumentar a probabilidade de produzir casos de teste

que sejam aceitos pelo Sistema em Teste (SET). Ao injetar sistematicamente dados inválidos entre os dados válidos, ele conseguem identificar de forma eficaz vulnerabilidades e falhas na segurança do sistema.

Além disso, a aplicação das técnicas de *fuzzing* vai além da geração de mensagens atípicas; também envolve modificar a aparência e a ordem típica das mensagens para revelar falhas sutis de *design* e vulnerabilidades que podem não ser evidentes com testes convencionais de entrada inválida. Por exemplo, uma sequência válida e aprovada de mensagens pode ser reorganizada, repetida ou até mesmo alterada em tipo para criar uma sequência atípica e desconhecida, revelando possíveis fragilidades no comportamento do sistema (SCHIEFERDECKER, GROSSMANN e SCHNEIDER, 2012).

Em geral, a combinação de teste baseado em modelos e técnicas de *fuzzing* oferece uma abordagem eficiente para o teste de segurança. Ao automatizar a geração de casos de teste e aproveitar o conhecimento sobre a estrutura das mensagens, esses métodos fornecem uma maneira sistemática e eficiente de identificar vulnerabilidades de segurança em sistemas. Embora ainda seja necessário mais pesquisa no campo do teste de segurança baseado em modelos, a integração de técnicas de *fuzzing* baseadas em blocos e modelos mostra potencial para melhorar a segurança de sistemas de software. (SCHIEFERDECKER, GROSSMANN e SCHNEIDER, 2012).

2.5. Trabalhos Relacionados

Nesta seção, serão apresentadas duas ferramentas amplamente reconhecidas e utilizadas no mercado que são pertinentes ao contexto desta pesquisa: SonarQube e Code Intelligence.

2.5.1. SonarQube

O SonarQube destaca-se como uma ferramenta consolidada no campo de desenvolvimento de software, conhecida por sua eficácia na análise estática de código-fonte. Esta ferramenta é projetada para fornecer *insights* abrangentes sobre a qualidade do código, destacando possíveis problemas, padrões de codificação e vulnerabilidades. (SONARQUBE, 2023).

No entanto, sua natureza paga e repleta de funcionalidades pode introduzir desafios para as equipes como:

- Custo elevado no valor da licença, e;
- Complexidade na configuração inicial, uma vez que isto requer conhecimento técnico avançado, o que pode demandar recursos consideráveis.

Em resumo, enquanto o SonarQube oferece benefícios substanciais, as equipes devem avaliar cuidadosamente os aspectos financeiros, de treinamento e

de recursos antes da adoção, garantindo uma implementação eficiente e eficaz da ferramenta.

O presente projeto e o SonarQube compartilham o objetivo de garantir a qualidade e segurança do software. Ambos buscam identificar vulnerabilidades, mas diferem na abordagem. Enquanto o software Fuzzer foca em execução de casos específicos, usando a técnica de fuzzing, o SonarQube realiza análise estática do código, identificando problemas sem a necessidade de execução.

2.5.2. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source

O artigo "*Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source*" aborda uma abordagem inovadora para testar bibliotecas de aprendizado profundo, ou *Deep-Learning* (DL) por meio de testes automatizados de *fuzzing* em nível de API. Em comparação com métodos anteriores de teste em nível de modelo, que se assemelham a testes de sistema, o teste em nível de API é mais granular, assemelhando-se aos testes de unidade. A vantagem desse método é a capacidade de oferecer uma abordagem mais geral e sistemática para testar bibliotecas de DL. (WEI, 2022).

A principal contribuição do trabalho de Wei (2022) foi a introdução da ferramenta FreeFuzz, que realiza testes de *fuzzing* em bibliotecas de aprendizado profundo usando informações dinâmicas obtidas de execuções reais de modelos e APIs. O FreeFuzz utiliza três fontes principais de informação: trechos de código da documentação da biblioteca, testes desenvolvidos pelos criadores da biblioteca e modelos de aprendizado profundo encontrados em repositórios públicos. Essas fontes alimentam um espaço de valores para cada API, permitindo a realização de testes de mutação com estratégias variadas, como mutação de tipo, mutação de valor aleatório e mutação de valor de banco de dados. (WEI, 2022).

A avaliação inicial do FreeFuzz em bibliotecas Python populares como PyTorch e TensorFlow mostrou resultados promissores. A ferramenta foi capaz de rastrear informações dinâmicas válidas para realizar testes de *fuzzing* em 1158 APIs, em comparação com 59 APIs cobertas por técnicas de ponta em TensorFlow. Além disso, FreeFuzz identificou 49 *bugs*, dos quais 38 foram confirmados pelos desenvolvedores como falhas desconhecidas anteriormente. (WEI, 2022).

Em conclusão, o FreeFuzz representa uma contribuição significativa para o campo de teste de bibliotecas de aprendizado profundo, introduzindo uma abordagem inovadora que combina informações dinâmicas de várias fontes para realizar testes de *fuzzing* automatizados em nível de API. A ferramenta demonstrou sua eficácia na detecção de *bugs* e na cobertura de APIs em comparação com métodos existentes. (WEI, 2022).

Tanto os Software Fuzzer quanto a presente pesquisa compartilham o objetivo comum de garantir a qualidade e segurança em desenvolvimento de software, embora abordem contextos distintos. Enquanto os testes do Software Fuzzer se concentra em avaliações de segurança específicas, como tentativas de

acesso não autenticado e inclusão de valores aleatórios nos parâmetros, utilizando fuzzing, a pesquisa de Wei (2022) explora uma abordagem inovadora para testar bibliotecas de aprendizado profundo usando a ferramenta FreeFuzz. Esta ferramenta realiza testes de fuzzing automatizados em nível de API, destacando a importância de métodos granulares e sistemáticos para avaliar bibliotecas de DL.

3. METODOLOGIA

O presente projeto adotou a metodologia de estudo de caso, que, segundo Toledo (2009), é definido como uma estratégia que visa justificar um conjunto de decisões tomadas em um contexto particular. Inicialmente foi feito um levantamento bibliográfico das técnicas de teste de software, com o intuito de identificar aquelas que poderiam ser empregadas para abranger a parte de teste de segurança na Empresa XYZ.

Após definida, foi feito um levantamento de requisitos através de uma entrevista, que foi conduzida com o engenheiro de software sênior do time de *backend* da Empresa XYZ. Este profissional, com formação em ciência da computação e uma experiência consolidada de 8 anos na área de desenvolvimento, foi escolhido como representante qualificado para abordar questões específicas relacionadas ao sistema em questão.

Durante a entrevista, o foco estava em identificar informações essenciais para o desenvolvimento da estratégia de testes que seria adotada no software Fuzzer. As seguintes questões foram exploradas:

1. Quais são os tipos de teste existentes no código de *backend* da aplicação de *marketplace* em relação à segurança?
2. Quais são os tipos de teste existentes no código de *backend* da aplicação de *marketplace*?
3. Qual é a importância de validar caminhos e parâmetros aleatórios dos *endpoints*?
4. Qual é a importância de validar requisições sem token ou com token aleatório no contexto da segurança do sistema?

Ao adotar essa abordagem específica, buscamos garantir uma discussão aprofundada e sem restrições, permitindo que o engenheiro compartilhasse suas percepções e expertise sobre as expectativas em relação aos testes que seriam desenvolvidos, com ênfase nas áreas de segurança e qualidade do código.

A partir desse levantamento, uma versão inicial do projeto foi desenvolvida, servindo como base para a segunda entrevista com o engenheiro de software. Nessa etapa, o foco foi coletar informações valiosas sobre a relevância dos resultados obtidos e estratégias para alimentar o software com os *endpoints* a serem testados.

Durante a segunda entrevista, adotou-se uma abordagem estruturada para avaliar a eficácia da versão inicial do software. As perguntas fundamentais foram:

1. Os resultados obtidos são considerados relevantes para os objetivos do projeto de *backend*?
2. Quais estratégias seriam mais apropriadas para alimentar o software com os *endpoints* a serem testados de maneira abrangente e eficaz?

Ao aplicar uma análise qualitativa dos relatórios gerados inicialmente pelo software, buscou-se compreender não apenas os números e métricas, mas também os contextos e nuances subjacentes aos resultados. Essa abordagem alinhava-se com a perspectiva de Gil (1999) e Cervo e Bervian (2002) sobre pesquisa qualitativa, destacando a importância da interpretação dos significados atribuídos aos fenômenos estudados, em detrimento de uma abordagem estritamente quantitativa.

Dessa forma, a segunda entrevista não apenas validou a utilidade da versão inicial do projeto, mas também proporcionou *insights* valiosos para aprimorar o software, considerando tanto a relevância dos resultados quanto estratégias eficientes para alimentar os *endpoints* de teste de forma robusta.

Após a conclusão da segunda entrevista, a versão final do projeto foi desenvolvida e apresentada ao time de desenvolvimento da Empresa XYZ que suporta a aplicação de *marketplace*, composto por 12 desenvolvedores. Durante essa apresentação, o propósito principal foi avaliar a criticidade dos erros identificados e incorporá-los como defeitos ou débitos técnicos no quadro de trabalho do time.

4. PROJETO

4.1. Descrição Geral do Projeto

Este projeto teve como objetivo o desenvolvimento de um software Fuzzer, utilizando as linguagens de programação Python e Javascript, destinado a testar a API de uma aplicação de *marketplace*. O software apresenta uma interface gráfica intuitiva e funcionalidades projetadas para simplificar o seu uso. Uma estratégia-chave adotada foi a integração com o Swagger (SWAGGER, 2023), uma ferramenta que descreve a estrutura de uma API, incluindo os *endpoints*, parâmetros, respostas, esquemas de dados, entre outras informações.

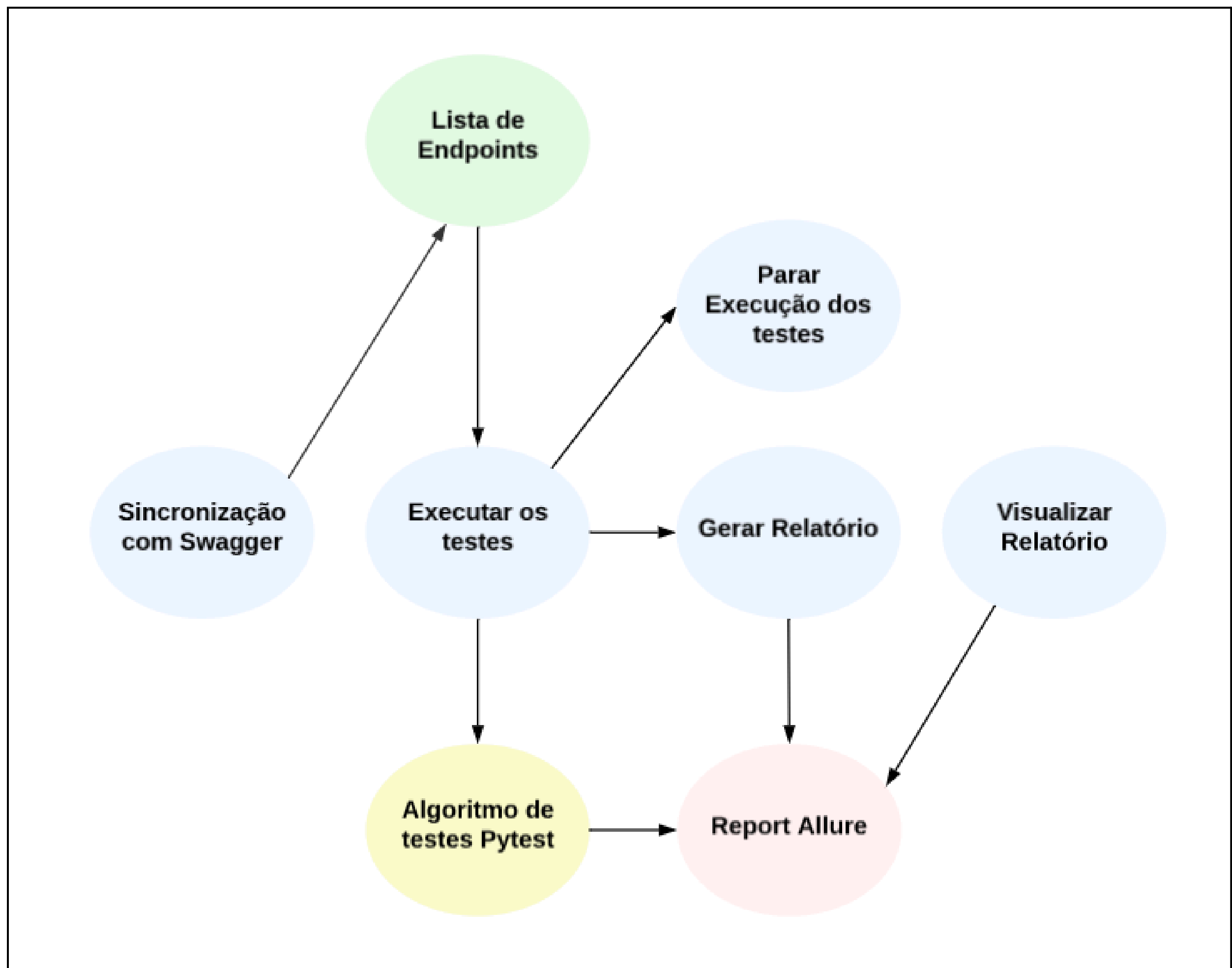
A integração com o Swagger proporciona retroalimentação constante ao software, tornando sua utilização mais acessível e eficiente. Além disso, o projeto incorpora a interface interativa, permitindo que tanto a equipe de testes quanto a equipe de desenvolvimento backend visualizem e interajam com as funcionalidades do software Fuzzer.

Os atores envolvidos no processo de utilização do software abrangem a equipe de testes e a equipe de desenvolvimento *backend* da empresa estudada. A utilização do software está planejada como parte do ciclo de testes de regressão. Com base na análise dos relatórios gerados após a execução, foram identificados os pontos que requerem ajustes antes da efetivação da próxima liberação em produção.

4.2. Diagrama Geral do Projeto

A Figura 1 apresenta um diagrama com o fluxo geral do sistema, que tem como objetivo fornecer uma visão geral das funcionalidades do software desenvolvido.

Figura 1- Diagrama do fluxo geral do sistema



Fonte: Elaborado pela autora

O diagrama da Figura 1 serve como um guia visual para compreender as funcionalidades-chave do software, que incluem:

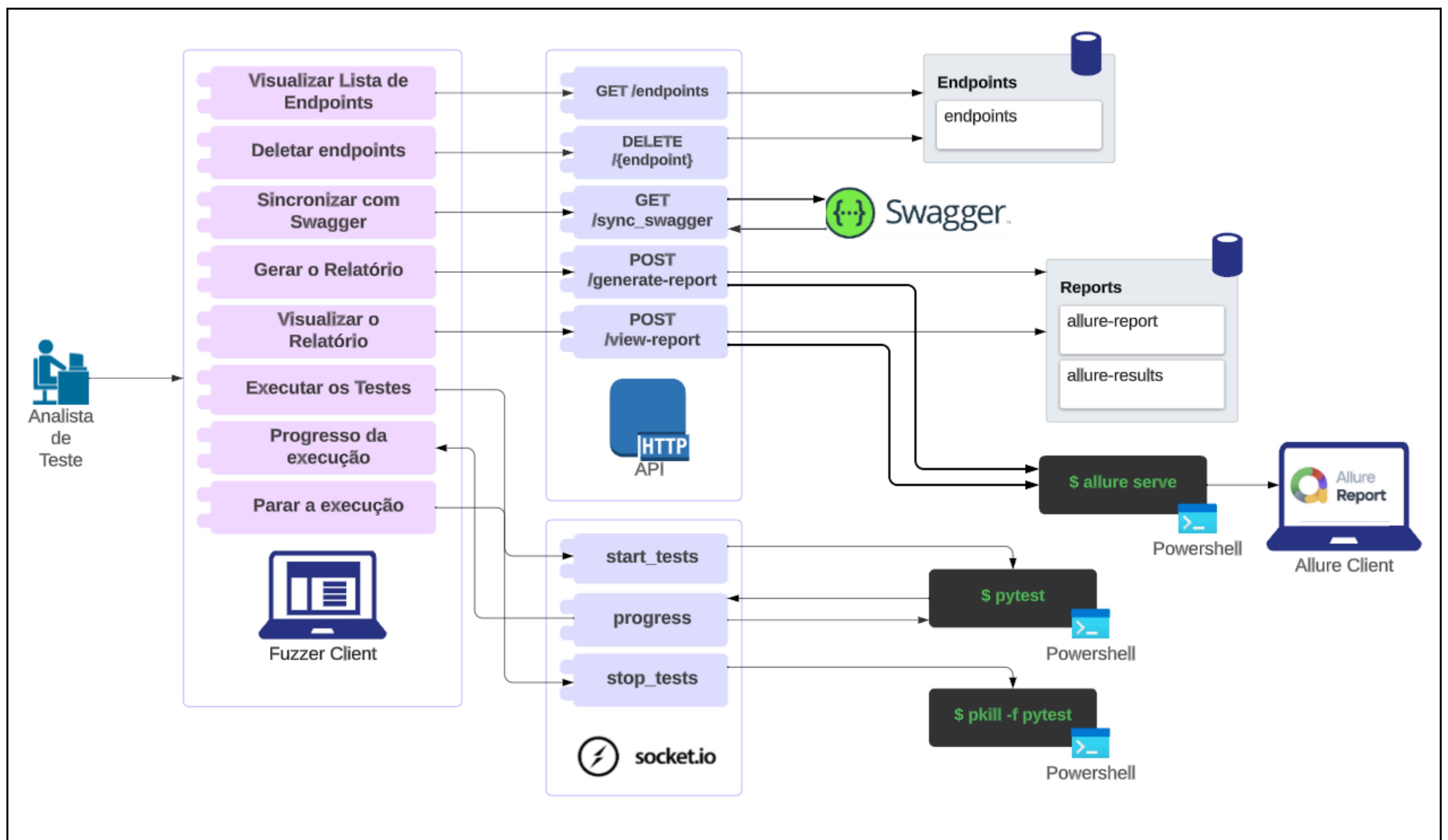
- Sincronização com Swagger: Busca no Swagger os *endpoints* publicados, armazenará em um arquivo json do software e este arquivo servirá para popular a tabela com a lista de *endpoints*;
- Execução dos testes: Ação para iniciar a execução dos scripts de teste;
- Parar Execução dos testes: Ação para interromper a execução dos testes;
- Geração de relatório: Após cada nova execução, o usuário poderá gerar um novo relatório utilizando a ferramenta Allure Report;
- Visualização do relatório: O usuário poderá a qualquer momento visualizar o último relatório gerado pela ferramenta Allure Report;

4.3. Apresentação do Software Fuzzer

Para o desenvolvimento do software, foram adotadas estratégias para

aprimorar a usabilidade do software, garantindo, ao mesmo tempo, sua eficácia e versatilidade, conforme ilustrado na Figura 2.

Figura 2 - Arquitetura do Software

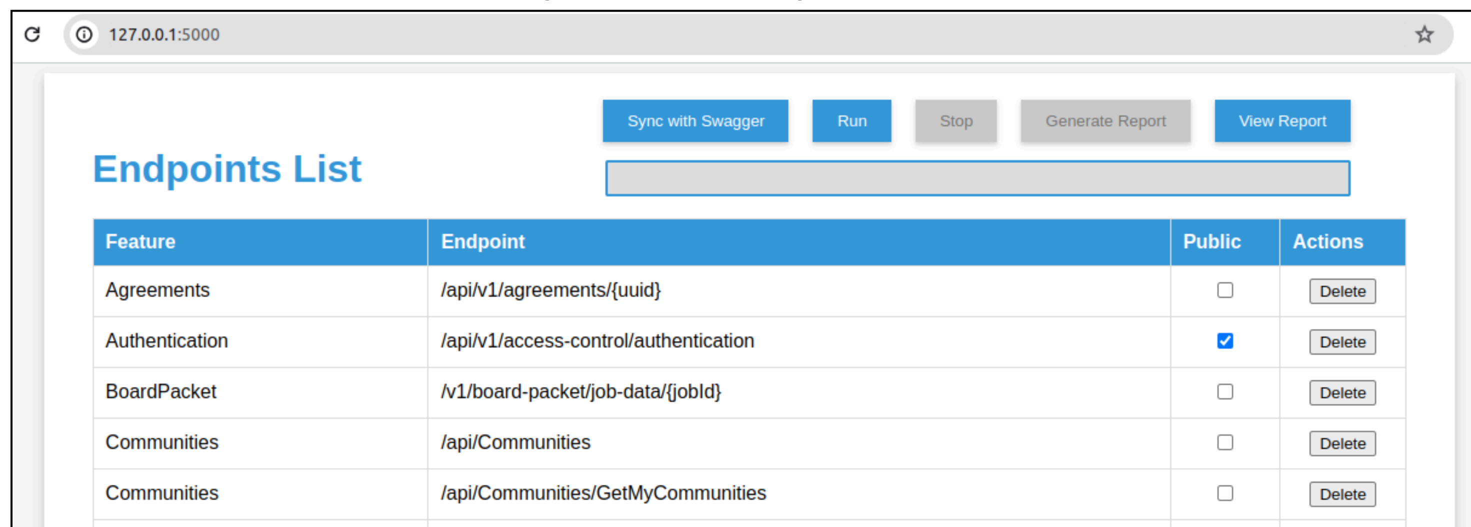


Fonte: Elaborado pela autora

Conforme apresentado pela Figura 2, foram criadas rotas de servidor para cada uma das funcionalidades: listagem de *endpoints*, deleção de *endpoints*, sincronização com Swagger, geração de relatório e visualização de relatório. Para a implementação destas rotas foi utilizado o framework Flask (FLASK 2023), que se destaca por sua simplicidade, flexibilidade e eficiência na criação de APIs. A comunicação entre o cliente e o servidor é realizada principalmente por meio do protocolo HTTP. No entanto, três funcionalidades específicas requerem comunicação direta com o terminal *PowerShell*: iniciar execução dos testes, interromper a execução e feedback de progresso, o que é alcançado por meio do uso de sockets.

A fim de tornar a experiência do software mais acessível, foi criada uma interface gráfica utilizando tecnologias web, HTML, CSS e JavaScript, conforme ilustrado na Figura 3.

Figura 3 - Interface gráfica do software



Fonte: Elaborado pela autora

Essa interface demonstrada na Figura 3 permite que os usuários visualizem os *endpoints* que serão testados e interajam com as funcionalidades implementadas, incluindo sincronização com o Swagger, execução dos testes, interrupção da execução, geração de relatórios, visualização de relatórios e exclusão de *endpoints*. Essas funcionalidades foram projetadas para simplificar o uso da aplicação, eliminando a necessidade de conhecimento técnico profundo. Ademais, foi implementada uma coluna na tabela chamada "*Public*", onde o usuário poderá selecionar os *endpoints* que devem aceitar solicitações sem autenticação, ou seja, são funcionalidades públicas.

Para avaliar a segurança e a qualidade das requisições do método HTTP GET da aplicação de *marketplace* da Empresa XYZ, foram desenvolvidos quatro testes empregando a técnica de *fuzzing*. A linguagem Python em conjunto com a ferramenta Pytest (PYTEST 2023) foram utilizadas para a criação dos scripts de casos de teste, a qual oferece diversas vantagens que justificam a sua escolha, dentre elas: sintaxe simples e clara, o que facilita a criação de testes compreensíveis; parametrização de testes, que permite que o mesmo teste execute com diferentes conjuntos de dados de entrada; oferece uma ampla gama de extensões para atender necessidades específicas, uma delas, a execução dos testes em paralelo sem a necessidade de muito esforço na configuração. Cada um dos casos de teste se concentrou em um aspecto específico da requisição, são eles:

1. Inclusão de parâmetros de consulta aleatórios nos *endpoints*;
2. Inclusão de valores aleatórios nos parâmetros de consulta existentes;
3. Inclusão de valores aleatórios nos parâmetros de rota dos *endpoints*, e;
4. Tentativa de acesso aos *endpoints* sem autenticação.

A Figura 4 apresenta a implementação do teste de tentativa de acesso aos *endpoints* sem autenticação.Essa avaliação é essencial para fortalecer a segurança

da aplicação, uma vez que verifica se a aplicação efetivamente impede acessos não autorizados aos seus *endpoints*, prevenindo potenciais exposições de informações confidenciais. Ao configurar o *endpoint* como público, conforme indicado na Figura 3, é possível assegurar que requisições desse tipo devem retornar com sucesso, já que não exigem autenticação.

Figura 4 - Implementação do teste de tentativa de acesso aos *endpoints* sem autenticação

```python
@allure.epic("Tokens")
@allure.feature("No token")
@pytest.mark.parametrize("endpoint", get_endpoints())
def test_no_token(endpoint):
    allure.dynamic.title(f"{endpoint['path']}")
    if "/api/v" in endpoint["path"]:
        allure.dynamic.story("/api/v1 - /api/v2")
    else:
        allure.dynamic.story("/api/")
    is_public = endpoint["public"]

    path_param = endpoint.get("pathParam", [])

    if path_param:
        path_param_name = path_param[0]["name"]
        endpoint = endpoint["path"].replace(f"{{{path_param_name}}}", "1")
    else:
        endpoint = endpoint["path"]

    with allure.step(f"{endpoint} - public: {is_public}"):
        try:
            response = get(endpoint, token=None)
            if is_public:
                assert response.status_code in [200, 404, 400]
            else:
                assert response.status_code not in [200, 503]
        except AssertionError as e:
            allure.attach(
                f"Error detected.\n\nStatus: {response.status_code}.\n\n"
                f"URL: {response.url}\n\nResponse:\n {response.text}",
                str(e),
                allure.attachment_type.TEXT,
            )
            pytest.fail()
```

Fonte: Elaborado pela autora

Além disso, o teste demonstrado na Figura 4 simula tentativas massivas de acesso não autenticado, comumente conhecidas como ataques de força bruta, e verifica que a aplicação não retornará sucesso ou serviço indisponível. Isso permite avaliar a resistência do sistema e verificar a implementação adequada de medidas como bloqueio ou temporização após múltiplas tentativas falhas.

As Figuras 5, 6 e 7 representam diferentes trechos de código dos testes implementados, cada um focando em aspectos distintos da segurança da aplicação. Embora tenham propósitos específicos, esses testes estão interconectados na busca por fortalecer a robustez do sistema considerando manipulações e entradas

não convencionais.

Figura 5 - Trecho de código da implementação do teste de inclusão de valores aleatórios nos parâmetros de rota dos *endpoints*

```
for query_param in query_params:
    query_param_name = query_param["name"]
    query_param_type = query_param["paramType"]

    if query_param_type == "string":
        query_parameters[query_param_name] = random_words()
    elif query_param_type == "integer":
        query_parameters[query_param_type] = random_number()

for i in range(10):
    endpoint_request = None
    for param in path_params:
        param_name = param["name"]
        param_type = param["paramType"]

        if param_type == "string":
            value = random.choice(
                [random_string, random_string_with_special_char, random_words]
            )
            endpoint_request = endpoint["path"].replace(
                f"{{{param_name}}}", value()
            )
        elif param_type == "integer":
            endpoint_request = endpoint["path"].replace(
                f"{{{param_name}}}", f"{random_number()}"
            )

    with allure.step(f"Execution {i}. Parameters: {endpoint_request}"):
        try:
            response = get(endpoint_request, query_parameters)
            assert response.status_code in [200, 404, 400, 204]
```

Fonte: Elaborado pela autora

Na Figura 5, o teste concentra-se na inclusão de valores aleatórios nos parâmetros de rota dos *endpoints*, visando identificar possíveis vulnerabilidades associadas à manipulação desses dados.

Figura 6 - Trecho de código da implementação do teste de inclusão de valores aleatórios nos parâmetros de consulta existentes

```
for i in range(10):
    random_params = {}

    for param in params:
        param_name = param["name"]
        param_type = param["paramType"]
        if param_type == "string":
            random_params[param_name] = random_string()
        elif param_type == "integer":
            random_params[param_name] = random_number()
        elif param_type == "boolean":
            random_params[param_name] = random.choice([False, True])
        elif param_type == "array":
            random_params[param_name] = random_array(param["itemsType"], min_items: 0, max_items: 100)
        else:
            raise ValueError(f"Param type is not supported: {param_type}")

        with allure.step(f"Execution {j}. Parameters: {random_params}"):
            try:
                response = get(endpoint, random_params)
                assert response.status_code in [200, 404, 400]
```

Fonte: Elaborado pela autora

Já na Figura 6, o enfoque se volta para a inclusão de valores aleatórios nos parâmetros de consulta existentes, explorando a capacidade da aplicação em lidar com consultas variáveis de forma segura.

Por fim, a Figura 7 apresenta a inclusão de parâmetros de consulta aleatórios nos endpoints, ampliando a análise para o comportamento do sistema diante de consultas diversificadas.

Figura 7 - Trecho de código da implementação do teste de inclusão de parâmetros de consulta aleatórios nos *endpoints*

```
for i in range(10):
    random_choice_key = random.choice(
        [
            random_string,
            random_string_with_special_char,
            random_number,
            random_words,
        ]
    )
    random_choice_value = random.choice(
        [
            random_string,
            random_string_with_special_char,
            random_number,
            random_words,
        ]
    )

    fuzzed_parameter = {
        f"{random_choice_key()}": random_choice_value(),
    }

    with allure.step(f"Execution {i}. Parameters: {fuzzed_parameter}"):
        try:
            response = get(endpoint, fuzzed_parameter)
            assert response.status_code in [200, 400, 404]
```

Fonte: Elaborado pela autora

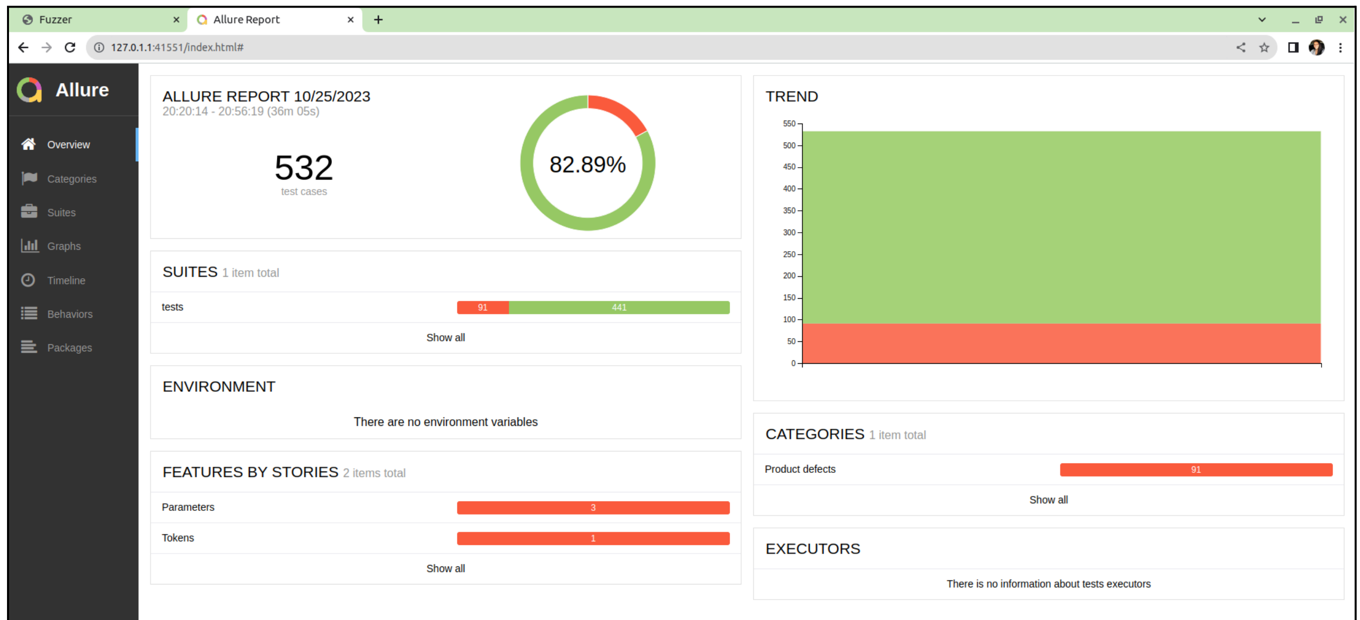
Em conjunto, os testes das Figuras 5, 6 e 7 compõem uma abordagem abrangente para verificar a resistência da aplicação contra manipulações maliciosas nos dados de entrada. Eles contribuem para identificar e mitigar vulnerabilidades, garantindo que a aplicação seja capaz de validar e processar entradas variáveis de maneira segura. Ao explorar diferentes aspectos, como parâmetros de rota e consulta, esses testes coletivamente reforçam a integridade e a segurança do sistema, proporcionando uma defesa abrangente contra potenciais ataques que visam explorar falhas na manipulação de dados.

Os valores aleatórios empregados nos testes foram gerados por meio da biblioteca Python "random", especializada em proporcionar aleatoriedade. Cada caso de teste foi devidamente parametrizado e executado de maneira paralela, utilizando a biblioteca "xdist". A execução paralela otimiza o tempo de teste, permitindo uma cobertura mais rápida e eficiente de todos os *endpoints* existentes no momento da execução.

Os resultados de cada execução são documentados em um relatório gerado pela ferramenta Allure (ALLURE 2023), que se sobressai neste segmento por ser gratuita e fornecer uma interface gráfica interativa e fácil de usar, oferecendo uma

visão abrangente do histórico de testes, incluindo gráficos de falhas, a duração das execuções e outras métricas relevantes, como ilustrado na Figura 8.

Figura 8 - Interface gráfica dos resultados



Fonte: Elaborado pela autora

A utilização da interface gráfica (Figura 8) proporcionou uma análise completa do desempenho da aplicação em termos de segurança e qualidade do código, permitindo a identificação de eventuais vulnerabilidades e áreas de melhoria de maneira eficaz e abrangente.

5. RESULTADOS

Este capítulo destina-se à exposição e análise detalhada dos dados obtidos ao longo da pesquisa. Aqui, os resultados obtidos são apresentados de forma organizada e interpretativa, proporcionando uma compreensão aprofundada das descobertas em relação aos objetivos estabelecidos.

5.1. Primeira Entrevista para Levantamento de Requisitos

A finalidade da primeira entrevista foi compreender os testes já implementados na plataforma de *backend* da aplicação de *marketplace*. O objetivo era ter uma visão mais nítida do que realmente é importante ser testado em termos de qualidade e segurança no contexto desta aplicação. As respostas do engenheiro sênior foram as seguintes:

1. Quais são os tipos de teste existentes no código de *backend* da aplicação de *marketplace* em relação à segurança?

   R: Não temos testes voltados especificamente para segurança.

2. Quais são os tipos de teste existentes no código de *backend* da aplicação de *marketplace* em relação à qualidade do código?

   R: Temos somente testes de integração, que fazem validações a nível de regras de negócio.

3. Qual é a importância de validar caminhos e parâmetros aleatórios dos *endpoints*?

   R: Acredito que houve uma grande evolução no desenvolvimento de software nos últimos anos, principalmente voltado a boas práticas de programação, código limpo e qualidade de código. Entradas maliciosas podem deixar nosso sistema vulnerável se não forem tratadas da maneira correta. Além disso, o não correto tratamento dos erros, podem causar problemas na utilização das funcionalidades. Acho que explorar por meio do teste o quão resiliente a aplicação é com relação a caminhos e parâmetros é uma excelente forma de detectar erros críticos, como deixar a aplicação indisponível, e débitos técnicos.

4. Qual é a importância de validar requisições sem token ou com token aleatório no contexto da segurança do sistema?

   R: Testar os *endpoints* enviando requisições sem autenticação é extremamente importante, e podem revelar falhas gravíssimas.

Principalmente quando parte da aplicação pode ser acessada sem autenticação, e parte da aplicação o usuário deve estar autenticado para que possa ver as informações, como no caso da Empresa XYZ.

5.2. Segunda Entrevista - Apresentação inicial do Software Fuzzer

A segunda entrevista com o engenheiro de software sênior teve como propósito reunir feedback sobre a relevância dos testes escritos para a aplicação de *marketplace* da Empresa XYZ e a apresentação dos resultados do software Fuzzer. As respostas foram as seguintes:

1. Os resultados obtidos são considerados relevantes para os objetivos do projeto de *backend*?

   R: São em partes. Após analisar os resultados gerados pelo relatório, pude notar que poderia incluir algumas validações um pouco diferentes com relação a parâmetros aleatórios e caminhos aleatórios. Basicamente, existem algumas respostas diferentes dependendo do caso, mas que são aceitas, como:
   - Resposta de sucesso (200): é aceita quando foi implementada uma estratégia de ignorar qualquer parâmetro ou caminho invalido.
   - Resposta de não encontrado (404): é aceita quando o caminho ou parâmetro é tratado corretamente e não é encontrado.
   - Resposta de *bad request* (400): é aceita quando existem parâmetros obrigatórios serem passados para que o *endpoint* funcione corretamente.

   À medida que escalamos os times da Empresa XYZ, mais importante é manter a consistência, qualidade e integridade da nossa API, então esse tipo de teste é essencial.

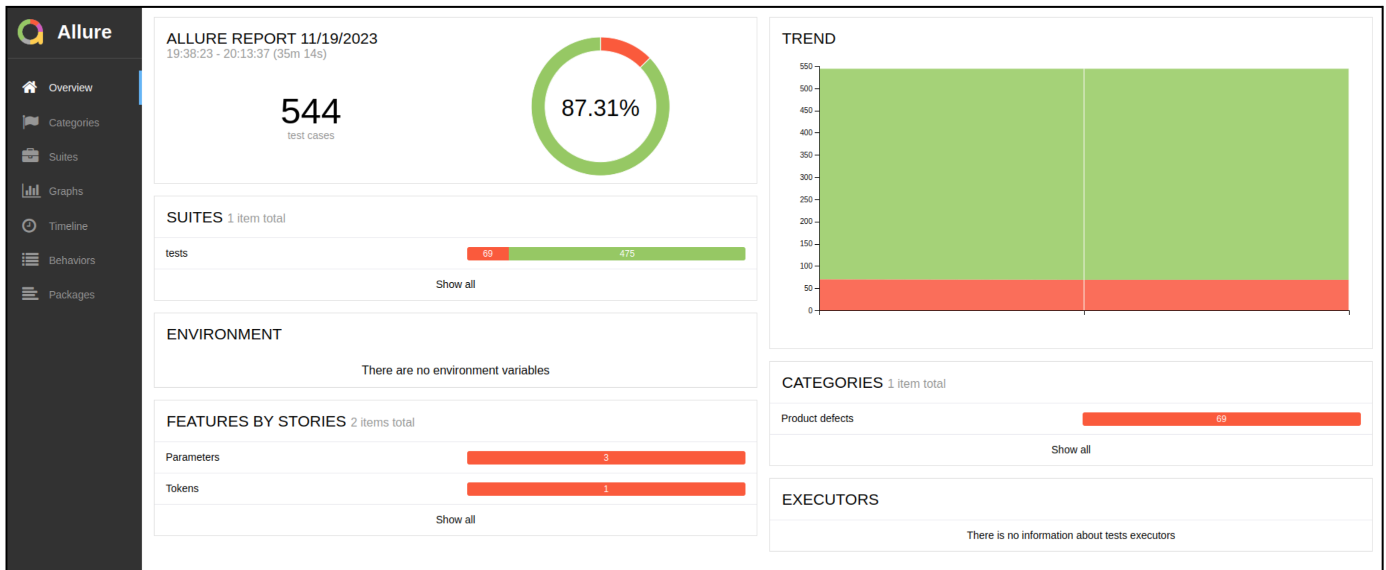
2. Quais estratégias seriam mais apropriadas para alimentar o software com os *endpoints* a serem testados de maneira abrangente e eficaz?

   R: Acredito que a estratégia que mais faria sentido, para evitar o processo manual de adicionar os *endpoints*, seria utilizar a API do Swagger para alimentar o software de teste. Publicamos na interface do Swagger todas as informações dos nossos *endpoints* como caminho, resposta de sucesso, exemplo de corpo de retorno, etc. Essa seria a melhor alternativa para tirarmos o máximo proveito do Software Fuzzer.

5.3. Apresentação e Análise do Relatório Final Gerado

A análise do relatório gerado pelo Allure Report revelou *insights* sobre a eficácia dos testes do Software Fuzzer aplicados aos *endpoints* do método HTTP GET. Um total de 136 *endpoints* foram submetidos aos 4 testes elaborados, resultando em 544 casos de teste. Desses, 69 testes falharam, representando uma taxa de falha de 12,68%, conforme apresentado na Figura 9.

Figura 9 - Visão geral dos resultados obtidos



Fonte: Elaborado pela autora

A categorização, ilustrada na Figura 10, das falhas foi crucial para compreender a natureza e a gravidade dos problemas identificados. O time de desenvolvimento *backend* classificou 14 dessas falhas como críticas. Esses casos críticos referiam-se a *endpoints* não públicos que, de maneira inesperada, respondiam com sucesso a requisições sem autenticação. Esta descoberta apontou para uma potencial vulnerabilidade de segurança, destacando a importância dos testes na identificação de brechas não evidentes.

As 55 falhas restantes foram considerados débitos técnicos. Esses débitos foram interpretados como oportunidades para fortalecer a aplicação, especialmente em relação à capacidade de lidar com parâmetros desconhecidos ou valores de parâmetros inválidos. Essa abordagem proativa visa melhorar a robustez do sistema, prevenindo potenciais problemas decorrentes de entradas inesperadas.

Figura 10 - Categorização dos testes

Fonte: Elaborado pela autora

Como resultado dessa análise, foram criadas 6 tarefas no quadro de tarefas do time de desenvolvimento. Essas tarefas visam endereçar e corrigir os problemas identificados durante os testes, garantindo que a aplicação seja aprimorada e reforçada contra possíveis falhas e vulnerabilidades.

Essa análise do relatório destaca não apenas os problemas encontrados, mas também a utilidade prática dos testes no processo de desenvolvimento. A identificação proativa de falhas críticas e débitos técnicos demonstra o valor desses testes como parte integrante das práticas de garantia de qualidade do sistema.

5.4. Comparação com Trabalhos Relacionados

O presente projeto representa uma contribuição valiosa no âmbito do teste de APIs em uma aplicação de *marketplace*, diferenciando-se por vários aspectos distintivos. A integração com o Swagger, a oferta de uma interface gráfica intuitiva e a ênfase em testes de regressão são características que fortalecem sua proposta, posicionando-se como uma solução prática e acessível.

Enquanto o SonarQube direciona seus esforços para a análise estática de código, visando identificar problemas, padrões de codificação e vulnerabilidades, e o FreeFuzz se destaca no teste automatizado de *fuzzing* em nível de API para bibliotecas de aprendizado profundo, ambos compartilham o objetivo comum de aprimorar a qualidade e a segurança do software. Essa busca por excelência reflete o comprometimento geral da comunidade de desenvolvimento em elevar os padrões no ciclo de vida do software.

Em síntese, os trabalhos apresentam abordagens complementares, contribuindo para o aprimoramento do ciclo de desenvolvimento de software. Cada um oferece uma perspectiva única e valiosa, demonstrando a diversidade de ferramentas e técnicas disponíveis para atender às demandas variadas do desenvolvimento de software contemporâneo.

6. CONCLUSÃO

O objetivo central do projeto consistiu em desenvolver o software Fuzzer e integrá-lo de maneira efetiva ao processo de desenvolvimento e teste de uma aplicação de *marketplace*, com ênfase na avaliação da qualidade e segurança do código.

A eficácia da metodologia empregada foi claramente comprovada através do levantamento de requisitos detalhados e das entrevistas conduzidas junto à equipe de desenvolvimento *backend*. Essas estratégias se mostraram cruciais para a compreensão aprofundada das necessidades e expectativas dos *stakeholders*, proporcionando uma base sólida para o desenvolvimento do software Fuzzer.

A análise minuciosa do relatório gerado pelo Allure Report revelou dados importantes sobre a eficácia dos testes realizados. Identificou-se um número significativo de falhas, incluindo aquelas classificadas como críticas, destacando vulnerabilidades que, de outra forma, poderiam passar despercebidas. Essas descobertas enfatizam a importância desses testes na identificação proativa de potenciais riscos de segurança, reforçando o papel fundamental do Software Fuzzer no aprimoramento da robustez e segurança do sistema.

A validação da versão final do software viabilizou a integração efetiva dos testes do Software Fuzzer nos procedimentos regulares de teste do *backend* da empresa estudada. Essa incorporação não apenas validou a eficácia da solução, mas proporcionou a criação de uma camada adicional de garantia de qualidade e uma abordagem proativa na identificação de vulnerabilidades.

Levando-se em consideração o potencial do software desenvolvido, os trabalhos futuros para este projeto visam iniciativas para aprimorar a abrangência e eficácia do sistema, como:

- Implantar o sistema em um servidor corporativo, a fim de disponibilizar as funcionalidades e relatórios via Web para os times;
- Implementação de agendamento automático para a execução dos testes, a fim de proporcionar uma abordagem mais eficiente e programada para a execução dos testes, facilitando a gestão e o monitoramento contínuo, e;
- Incluir testes para outros métodos HTTP, como POST.

Essas iniciativas representam uma visão abrangente para impulsionar o projeto em direção a uma implementação mais robusta e automatizada.

7. REFERÊNCIAS

1. ABCOMM. **Marketplaces: crescimento exponencial ao longo da pandemia.** ABCOMM, 2021. Disponível em: <https://abcomm.org/noticias/marketplaces-crescimento-exponencial-ao-longo-da-pandemia/> . Acesso em: 10 de maio de 2023.

2. ALLURE (2023). **Allure Report Documentation.** Disponível em: <https://allurereport.org/docs/> . Acesso em: 02 de dezembro de 2023.

3. BEIZER, Boris. **Software testing techniques**. Dreamtech Press, 2003.

4. BOGDAN, R. S.; BIKEN, S. **Investigação qualitativa em educação: uma introdução à teoria e aos métodos.** 12.ed. Porto: Porto, 2003.

5. CERVO, A. L. BERVIAN, P. A. **Metodologia científica**. 5.ed. São Paulo: Prentice Hall, 2002.

6. DEFEHR, J. **Investigación acción dialógica: El fenómeno de agencia democrática y transformativa de la habilidad de respuesta**. Universidad de Winnipeg, 2015.

7. FLASK (2023). **About Flask** . Disponível em: <https://flask.palletsprojects.com/en/3.0.x/> . Acesso em: 02 de dezembro de 2023.

8. GIL, A. C. **Métodos e técnicas de pesquisa social**. 5.ed. São Paulo: Atlas, 1999.

9. GIL, A. C. **Métodos e técnicas de pesquisa social.** 6. ed. São Paulo: Atlas, 2008.

10. GODEFROID, P., KIEZUN, A., & LEVIN, M. Y. **Grammar-based whitebox fuzzing**. In Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation (pp. 206-215). 2008. Disponível em: <https://patricegodefroid.github.io/public_psfiles/pldi2008.pdf> Acesso em: 29 de maio de 2023.

11. IEEE. IEEE Standard 610-1990: **IEEE Standard Glossary of Software Engineering Terminology, IEEE Press**. Disponivel em: <https://www.informatik.htw-dresden.de/~hauptman/SEI/IEEE_Standard_Glossary_of_Software_Engineering_Terminology%20.pdf> . Acesso em: 18 de

junho de 2023.

12. KHAN, M. E. **Different forms of software testing techniques for finding errors.** International Journal of Computer Science Issues (IJCSI), 7(3), 24. 2010. Disponível em: <https://www.researchgate.net/profile/A-Zaidan/publication/46093547_Towards_Corrosion_Detection_System/links/549239a60cf2484a3f3e0b22/Towards-Corrosion-Detection-System.pdf#page=19> Acesso em: 08 de junho de 2023.

13. KHAN, M. E.; KHAN, F. **Importance of Software Testing in Software Development Life Cycle.** International Journal of Computer Science Issues, Vol. 11, nº 2, 2014. Disponível em: <https://www.proquest.com/openview/a938358c4025fb1fc1551f31dbf81eaa/1?pq-origsite=gscholar&cbl=55228> Acesso em: 23 de junho 2023.

14. LAKATOS, E. M.; MARCONI, M. de A. **Fundamentos de metodologia científica**. 6. ed. 5. reimp. São Paulo: Atlas, 2007.

15. LI, Y., et al. V-fuzz: **Vulnerability-oriented evolutionary fuzzing.** 2019. arXiv preprint arXiv:1901.01142. Disponível em: <https://arxiv.org/pdf/1901.01142.pdf> Acesso em: 11 de junho de 2023.

16. MCNAMEE, S. Research as a relational practice. In Simon, G; Chard, A. (Eds.). **Systemic inquiry: innovations in reflexive practice research**. London: Everything is connected Press, 74-94. 2014

17. MILLER, B. et al. **Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services.** Computer Sciences Department University of Wisconsin. 1995. Disponível em: <https://www.paradyn.org/papers/fuzz-revisited.pdf> Acesso em: 05 de junho de 2023.

18. MILLER, B. et al. **Study of the Reliability of UNIX utilities** - Communications of the ACM, Vol. 3, nº 12, 1990. Disponível em: <https://dl.acm.org/doi/pdf/10.1145/96267.96279> Acesso em: 03 de junho de 2023.

19. MILLER, B. et al. **The Relevance of Classic Fuzz Testing:Have We Solved This One?** IEEE Transactions on Software Engineering, Vol. XX, nº. YY, Fevereiro, 2021. Disponível em: <https://arxiv.org/pdf/2008.06537.pdf> Acesso em: 05 de junho de 2023.

20. MILLER, Barton P.; ZHANG, Mengxiao; HEYMANN, Elisa R. **The relevance of classic fuzz testing: Have we solved this one?.** IEEE Transactions on

Software Engineering, v. 48, n. 6, p. 2028-2039, 2020.

21. MILLER, Barton. **Foreword for Fuzz Testing Book.** Madison, Wisconsin, April 2008. Disponível em: <https://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html> Acesso em: 03 de junho de 2023.

22. MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom. **The art of software testing**. John Wiley & Sons, 2011.

23. NETO, A. C. D. **Introdução a Teste de Software.** Engenharia de Software Magazine. 2007. Disponível em: <https://www.researchgate.net/profile/Arilo-Neto/publication/266356473_Introducao_a_Teste_de_Software/links/5554ee6408ae6fd2d821ba3a/Introducao-a-Teste-de-Software.pdf> Acesso em: 25 de maio de 2023.

24. PAN, F. et al. **Efficient Model-based Fuzz Testing Using Higher-order Attribute Grammars**. J. Softw., v. 8, n. 3, p. 645-651, 2013. Disponível em: <http://www.jsoftware.us/vol8/jsw0803-16.pdf> Acesso em: 08 de junho de 2023.

25. POTTER, B.; McGRAW, G. **Software security testing.** IEEE Security & Privacy, 2(5), 81-85. 2004. Disponível em: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=63b02c70e2e19fe0ede9a6c3e8c30558d042151c> Acesso em: 08 de junho 2023.

26. PRESSMAN, R. S., **“Software Engineering: A Practitioner’s Approach”**, McGraw-Hill, 6th ed, Nova York, NY, 2005.

27. PRODANOV, C. C.; FREITAS, E.C. **Metodologia do trabalho científico [recurso eletrônico] : métodos e técnicas da pesquisa e do trabalho acadêmico**. 2. ed. Novo Hamburgo: Feevale, 2013.
28. PYTEST (2023). **pytest: helps you write better programs.** Disponível em: <https://docs.pytest.org/en/7.4.x/> . Acesso em: 02 de dezembro de 2023.

29. RICHARDSON, R. J. **Pesquisa social: métodos e técnicas.** São Paulo: Editora Atlas, 1999.

30. SCHIEFERDECKER, I., GROSSMANN, J., & SCHNEIDER, M. **Model-based security testing**. Workshop on Model-Based Testing 2012 (MBT 2012). arXiv preprint arXiv:1202.6118. Disponível em: <https://arxiv.org/pdf/1202.6118.pdf> Acesso em: 11 de junho de 2023.

31. SHARIF, Md Haris Uddin; MOHAMMED, Mehmood Ali. **A literature review of financial losses statistics for cyber security and future trend**. World Journal of Advanced Research and Reviews, v. 15, n. 01, p. 138-156, 2022. Disponível em: <https://wjarr.com/sites/default/files/WJARR-2022-0573.pdf> . Acesso em: 10 de maio de 2023.

32. SOMMERVILE, Ian; **Engenharia de Software / Ian Sommerville** ; tradução Ivan Bosnic e Kalinka G. de O. Gonçalves ; revisão técnica Kechi Hirama. — 9. ed. — São Paulo : Pearson Prentice Hall, 2011. Disponível em: <https://www.facom.ufu.br/~william/Disciplinas%202018-2/BSI-GSI030-EngenhariaSoftware/Livro/engenhariaSoftwareSommerville.pdf> . Acesso em: 5 de novembro de 2023.

33. SONARQUBE (2023). **SonarQube 10.3 Documentation.** Disponível em: <https://docs.sonarsource.com/sonarqube/latest/> . Acesso em: 02 de dezembro de 2023.

34. SWAGGER (2023). **OpenAPI Specification**. Disponível em: <https://swagger.io/specification/> . Acesso em: 01 dezembro de 2023.

35. TOLEDO, Luciano Augusto; DE FARIAS SHIAISHI, Guilherme. **Estudo de caso em pesquisas exploratórias qualitativas: um ensaio para a proposta de protocolo do estudo de caso**. Revista da FAE, v. 12, n. 1, 2009.

36. WEI, Anjiang et al. **Free lunch for testing: Fuzzing deep-learning libraries from open source**. In: Proceedings of the 44th International Conference on Software Engineering. 2022. p. 995-1007. Disponível em: <https://dl.acm.org/doi/abs/10.1145/3510003.3510041> . Acesso em: 02 de dezembro de 2023.