UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Marcelo de Castro Loebens

Detectando ataques Broken Object Level Authorization em APIs REST usando dependência Produtor-Consumidor

Manaus
Fevereiro de 2022

Marcelo de Castro Loebens

Detectando ataques Broken Object Level Authorization em APIs REST usando dependência Produtor-Consumidor

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas como requisito para obtenção do grau de Mestre em Informática.

Orientador: Prof. Eduardo Luzeiro Feitosa

Manaus
2022

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Loebens, Marcelo de Castro

L825d   Detectando ataques Broken Object Level Authorization em APIs REST usando dependência Produtor-Consumidor / Marcelo de Castro Loebens . 2022

54 f.: il. color; 31 cm.

Orientador: Eduardo Luzeiro Feitosa

Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. API - Application Programming Interfaces. 2. BOLA - Broken Object Level Authorization. 3. Detecção. 4. OpenAPI. 5. vulnerabilidade. I. Feitosa, Eduardo Luzeiro. II. Universidade Federal do Amazonas III. Título

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
INSTITUTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

FOLHA DE APROVAÇÃO

"Detectando ataques Broken Object Level Authorization em APIs REST usando dependência Produtor-Consumidor "

MARCELO DE CASTRO LOEBENS

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Eduardo Luzeiro Feitosa - PRESIDENTE

Prof. Horácio Antônio Braga Fernandes de Oliveira - MEMBRO INTERNO

Diego Luis Kreutz:93504420006 Digitally signed by Diego Luis Kreutz:93504420006 Date: 2022.02.25 11:40:40 -03'00'

Prof. Diego Luis Kreutz - MEMBRO EXTERNO

Manaus, 25 de Fevereiro de 2022

Resumo

As Web APIs (*Application Programming Interfaces*) vêm se tornando ubíquas em aplicações que demandam algum tipo de comunicação entre cliente e servidor, sendo desenvolvidas majoritariamente utilizando o estilo de arquitetura REST *(Representational State Transfer*) e largamente empregadas no desenvolvimento de aplicações Web e móveis. Esse aumento de popularidade, no entanto, trouxe novos desafios de segurança, como a difusão de vulnerabilidades derivadas principalmente da ausência de controles de estado das aplicações clientes, reflexo do requisito de *statelessness* do design REST. Dentre essas vulnerabilidades destaca-se a *Broken Object Level Authorization* (ou BOLA), um tipo específico de quebra de controle de acesso. Ela foi elencada na primeira posição do OWASP *API Security Top 10*, sendo considerada a vulnerabilidade de maior prevalência em aplicações do mundo real, uma vez que pode levar ao acesso não autorizado a dados sensíveis. Este trabalho visa fornecer uma abordagem de detecção de tentativas de exploração dessa vulnerabilidade em tempo de execução, a partir da identificação de relacionamentos produtor-consumidor entre os *endpoints*, extraídos a partir de especificações da API no padrão OpenAPI (OAS). Essa solução será avaliada utilizando-se APIs de aplicações vulneráveis a BOLA, visando validar a sua capacidade de detecção de ataques.

Palavras-chave: API Web, BOLA, detecção, OpenAPI, OWASP, vulnerabilidade.

Abstract

Web APIs (Application Programming Interfaces) have become ubiquitous in applications that require some type of communication between client and server, being developed mainly using the REST (Representational State Transfer) style of architecture and widely used in the development of Web and mobile applications. This increase in popularity, however, brought new security challenges, such as the spread of vulnerabilities derived mainly from the absence of state controls on client applications, reflecting the statelessness requirement of REST design. Among these vulnerabilities, the Broken Object Level Authorization (or BOLA), a specific type of access control breach, stands out. It's listed in the first position of the OWASP API Security Top 10 and it's considered the most prevalent vulnerability in real-world applications, leading to unauthorized access to sensitive data in successful attacks. This work aims to provide an approach to detect attempts to exploit this vulnerability at runtime, using the identification of producer-consumer relationships between endpoints extracted from API specifications compliant to the standard OpenAPI (OAS). This solution will be evaluated using APIs of applications vulnerable to BOLA, in order to validade its capacity to detect attacks.

Keywords: Web API, BOLA, detection, OpenAPI, OWASP, vulnerability.

Lista de Figuras

2.1 Exemplo de aplicação vulnerável a BOLA. Fonte: Elaborada pelo autor. 10

3.1 Resultado da Aplicação dos Critérios. Fonte: Elaborada pelo autor. 15
3.2 Compilado dos métodos de detecção/prevenção citados. Fonte: Elaborada pelo autor. 18

4.1 Exemplo de requisição HTTP para um *endpoint* produtor de um recurso. Fonte: Elaborada pelo autor. 22
4.2 Exemplo de requisição HTTP para um *endpoint* consumidor de um recurso. Fonte: Elaborada pelo autor. 23
4.3 Método proposto para detecção de ataque BOLA. Fonte: Elaborada pelo autor. 24
4.4 Extração (*parsing*) da informação relevante da especificação OpenAPI. Fonte: Elaborada pelo autor. 26
4.5 Análise de requisições pelo *proxy*, considerando extração da Figura 4.4. Fonte: Elaborada pelo autor. 27

5.1 crAPI - Interface de navegação. Fonte: Capturada pelo autor. 31
5.2 crAPI - Resultado da requisição ao *endpoint* '*GET community/api/v2/community/posts/recent*'. Fonte: O autor. 32
5.3 crAPI - Exploração do *endpoint* vulnerável '*GET /identity/api/v2/vehicle/{car_id}/location*'. Fonte: O autor. 33
5.4 crAPI - Resultado da requisição ao *endpoint* '*POST /workshop/api/merchant/contact_mechanic*'. Fonte: O autor. 33
5.5 crAPI - Exploração do *endpoint* vulnerável '*GET /workshop/api/mechanic/mechanic_report*'. Fonte: O autor. 34
5.6 vAPI - Exploração do *endpoint* vulnerável '*GET /vapi/api1/user/{api1_id}*'. Fonte: O autor. 35
5.7 vAPI - Exploração do *endpoint* vulnerável '*PUT /vapi/api1/user/{api1_id}*'. Fonte: O autor. 36
5.8 PIXI - Interface de navegação. Fonte: Capturada pelo autor. 37
5.9 PIXI - Erro no servidor revela nome de arquivo de configuração. Fonte: Capturado pelo Autor. 38
5.10 PIXI - Arquivo de configuração contendo segredo é exposto ao método http GET. Fonte: O autor. 38
5.11 PIXI - JWT expõe diversos dados de usuário e informação que pode ser usada para engenharia reversa do servidor. Fonte: Capturado pelo autor. 39
5.12 PIXI - JWT pode ser modificado de forma válida utilizando o segredo. Fonte: Capturado pelo autor. 40

5.13 PIXI - *Endpoint GET /api/user/info* retorna informação a respeito de qualquer usuário, desde que o JWT incluído no *header* ‘x-access-token’ seja válido. Fonte: O autor. 40
5.14 crAPI - Link inserido na especificação OAS entre operações que produzem e consomem o identificador ‘car_id’. Fonte: O autor. 41
5.15 crAPI - Link inserido na especificação OAS entre operações que produzem e consomem o identificador ‘report_id’. Fonte: O autor. 42
5.16 crAPI - Detecção de tentativa de exploração da vulnerabilidade pela implementação da metodologia sugerida. Fonte: O autor. 42
5.17 crAPI - Requisições onde tentativas de ataque BOLA foram detectadas pela metodologia sugerida são respondidas pelo *proxy* com ‘401 - *Unauthorized*’ sem chegar ao servidor. Fonte: O autor. 43
5.18 vAPI - Link inserido na especificação OAS entre operações que produzem e consomem o identificador ‘api1_id’. Fonte: O autor. 44
5.19 vAPI - Detecção de tentativa de exploração da vulnerabilidade pela implementação da metodologia sugerida. Fonte: O autor. 45
5.20 vAPI - Requisições onde tentativas de ataque BOLA foram detectadas pela metodologia sugerida são respondidas pelo *proxy* com ‘401 - *Unauthorized*’ sem chegar ao servidor. Fonte: O autor. 45
5.21 PIXI - Todos os valores do parâmetro ‘picture_id’ são retornados pelo endpoint *GET /api/pictures*. Em uma requisição futura a um *endpoint* consumidor do parâmetro com qualquer dos valores retornados, o *proxy* verifica que o valor se encontra na *Allowlist* e não bloqueia a requisição. Fonte: O autor. 46

Sumário

1 Introdução 2

1.1 Motivação 3

1.2 Objetivos 4

1.3 Contribuições Esperadas 4

1.4 Estrutura do Documento 4

2 Conceitos Básicos 5

2.1 API Web 5

2.1.1 REST 5

2.1.2 OpenAPI 6

2.2 OWASP 8

2.2.1 *API Security Top 10* 2019 e BOLA 8

2.3 Considerações sobre o Capítulo 10

3 Trabalhos Relacionados 11

3.1 Revisão Inicial da Literatura 11

3.1.1 Trabalhos identificados por *Snowball Sampling* 12

3.1.1.1 RESTalk 12

3.1.1.2 RESTler 12

3.1.1.3 Detecção de APIs Scrapers através do fluxo de requisições 13

3.1.1.4 Suporte a Navegação via HATEOAS 13

3.2 Revisão Sistemática da Literatura 13

3.2.1 Protocolo de revisão sistemática 14

3.2.2 Trabalhos identificados na Revisão Sistemática 15

3.2.3 Respostas as Questões de Pesquisa 18

3.3 Discussão 20

4 Método Proposto 22

4.1 Definições 22

4.2 Método 23

4.3 Implementação 24

4.3.1 Validador 25

4.3.2 Classificador 25

4.3.3 *Allowlist* 26

4.4 Considerações sobre o Capítulo 28

5 Resultados **30**
5.1 Condições de Experimentação 30
5.2 Aplicações e Vulnerabilidades 31
5.2.1 crAPI 31
5.2.2 vAPI 33
5.2.3 PIXI 37
5.2.4 VAmPI 41
5.3 Resultados 41
5.3.1 crAPI 41
5.3.2 vAPI 44
5.3.3 PIXI 46
5.3.4 VAmPI 47
5.4 Discussão 47

6 Conclusão **49**
6.1 Dificuldades Encontradas 50
6.2 Trabalhos Futuros 50

Referências Bibliográficas **52**

Capítulo 1

Introdução

O avanço na disponibilização de aplicações que fornecem serviços e informação sob demanda na Internet potencializou o aumento da distribuição e uso de Web APIs (*Application Programming Interfaces*). São várias as vantagens do uso de Web APIs como a redução no tempo de desenvolvimento, a facilidade de integração multiplataforma e entre serviços de terceiros, a reusabilidade de código, entre outras. O relatório *State of The Internet / Security* Vol. 5, da provedora de segurança Akamai, descreve que, em 2018, 83% do tráfego monitorado por eles era proveniente de Web APIs (Akamai, 2019).

Essas APIs são frequentemente disponibilizadas seguindo o design REST (*Representational State Transfer*), por facilitar a compreensão e implementação no lado cliente (Petrillo et al., 2016). Além disso, REST possui seis princípios (arquitetura cliente-servidor, *statelessness* - ausência de controles de estado no servidor, *cacheability*, comunicação em camadas, interface uniforme e código sob demanda - opcional)(Fielding, 2000) que, se seguidos, potencializam os benefícios para Web APIs.

Contudo, o crescimento no uso de APIs REST foi acompanhado por um aumento no número de vulnerabilidades, que colocam usuários e companhias em risco. Muitas delas acabam negligenciadas pelas desenvolvedoras de software até que um grande problema venha à tona (Macy, 2018). Para se ter uma ideia, a entidade sem fins lucrativos OWASP (*Open Web Application Security Project*), na última versão de seu *Top 10 Most Critical Web Application Security Risks* (OWASP, 2017) - uma lista montada a partir da contribuição de profissionais da área contendo as maiores vulnerabilidades encontradas pelas organizações em aplicações web - mostrou que nove das dez vulnerabilidades descritas continham algum subcomponente relacionado a APIs. Isso fez com que a OWASP propusesse um novo projeto somente voltado para segurança de API's, batizado de *API Security Top 10* 2019 (OWASP, 2019).

Dentre as 10 vulnerabilidades, a listada no topo da lista, *Broken Object Level Authorization* ou simplesmente BOLA, enfatiza a tendência das Web APIs em expor informações sobre suas funcionalidades (*endpoints*). Atacantes manipulam referências a objetos enviados na requisição ao servidor, obtendo acesso indevido a recursos caso não existam contramedidas. Vale ressaltar que BOLA tem destaque na lista pela facilidade em explorar a vulnerabilidade, somada à dificuldade de adotar contramedidas organizadas que protejam todo o fluxo da comunicação.

1.1 Motivação

O aumento do oferecimento de serviços via Web baseados em API implica em preocupações de segurança, especialmente quanto as vulnerabilidades específicas desse tipo de arquitetura. A OWASP, em seu projeto API Security Top 10, listou a vulnerabilidade *Broken Object Level Authorization* (BOLA) como primeiro lugar dentre as mais relevantes em APIs. A vulnerabilidade, que é uma refatoração da conhecida IDOR (*Insecure Direct Object Reference*)(Yalon and Shkedy, 2019), já foi explorada em serviços de empresas como Facebook[1], Uber[2], T-Mobile[3], Apple[4], entre outras.

O problema com *Broken Object Level Authorization* é justamente com a autorização no nível do objeto, ou seja, quem pode acessar quais dados. Uma vez que objetos podem ser várias coisas que um usuário pode acessar (por exemplo, o conteúdo de sua cesta de compras, pedidos anteriores, endereço atual, cartões de crédito, entre outros), é de se esperar que, uma vez logado, apenas o usuário deva acessar seus próprios objetos. Mas, se houver um problema com a autorização, outros usuários também poderão acessá-los.

As atuais contramedidas incluem o uso de identificadores aleatórios para os recursos, a implementação de controles de autenticação mais robustos e testes extensivos (OWASP, 2019). Contudo, essas contramedidas possuem problemas, e são focadas em encontrar vulnerabilidades em ambiente de desenvolvimento, não na detecção em tempo de execução. Vários desses problemas derivam do próprio design REST, que estimula o uso de diversos identificadores, o que gera a necessidade de implementar controles de autenticação para cada um deles.

Os controles de autenticação são dependentes da implementação no lado do servidor, sendo difíceis de generalizar (Prokhorenko et al., 2016). Além disso, a implementação é uma tarefa trabalhosa, o que acaba por facilitar o aparecimento de falhas, uma vez que depende inteiramente da capacidade do desenvolvedor (Li and Xue, 2011). Estes devem se preocupar com políticas de acesso (que podem ser complexas e com diferentes níveis de autorização) que regulem cada recurso (os quais usualmente existem em quantidade elevada). Na literatura, os testes usualmente são implementados utilizando técnicas de análise estática ou de *fuzzing*. Essas técnicas, ainda que usualmente eficientes, dependem de código fonte (não genéricas), e tem dificuldade de lidar com código não nativo e entender fluxos de estado (Mendoza and Gu, 2018).

Recentemente, o trabalho de Atlidakis et al. (2019) utilizando *fuzzers* para detectar defeitos em uma API utilizou a extração das relações de dependência do tipo produtor-consumidor, diretamente da especificação da API, observando que para modelar o acesso a um determinado recurso é necessário que requisições sejam executadas em uma determinada ordem de precedência.

É com base nessa relação que esta proposta visa determinar se um identificador inserido em uma requisição foi informado ao cliente pelo servidor em uma requisição anterior. Para obter o identificador, o cliente deve ter executado uma requisição a um *endpoint* produtor de determinado recurso para que possa informá-lo posteriormente em uma requisição a um *endpoint* consumidor deste recurso. Essa abordagem, empregando a especificação das APIs, é capaz de mitigar um ataque do tipo BOLA, cujo cenário envolve alteração dos identificadores para obter recursos que estão fora do escopo de autorização do cliente. A abordagem difere dos

[1] Disponível em: <https://medium.com/bugbountywriteup/disclose-private-attachments-in-facebook-messenger-infrastructure-15-000-ae13602aa486>, acesso em 01/12/2020.

[2] Disponível em: <https://www.forbes.com/sites/daveywinder/2019/09/12/uber-confirms-account-takeover-vulnerability-found-by-forbes-30-under-30-honoree/?sh=370328679b87>, 01/12/2020.

[3] Disponível em: <https://arstechnica.com/information-technology/2017/10/t-mobile-website-bug-apparently-exploited-to-mine-sensitive-account-data/>, acesso em 01/12/2020.

[4] Disponível em: <https://thehackernews.com/2020/05/sign-in-with-apple-hacking.html>, acesso em 01/12/2020.

métodos descritos na literatura por ser agnóstica quanto a implementação no lado do servidor (a abordagem utiliza a descrição da API) ao mesmo tempo que propicia a detecção de tentativas de ataque em tempo de execução, não em ambiente de teste, podendo evitar que falhas nos controles de autenticação possam ser exploradas para esse tipo de ataque.

1.2 Objetivos

O objetivo geral desta proposta é elaborar um método para detecção de ataques *Broken Object Level Authorization* a APIs Web, através da identificação de relações produtor-consumidor a partir da especificação desta API, possibilitando a validação de requisições contra esse tipo de ataque em tempo de execução.

Como objetivos específicos, pretende-se:

- Elaborar uma solução para identificação e extração de relações produtor-consumidor, através da analise da especificação (no formato *OpenAPI Specification*) de uma API Web.
- Propor uma arquitetura capaz de utilizar essas relações para estabelecer uma sequência de causalidade capaz de validar a requisição atual de acordo com as requisições e respostas anteriores, em tempo de execução.

1.3 Contribuições Esperadas

Espera-se que, ao final do trabalho, as seguintes contribuições sejam alcançadas:

- Um método de detecção de ataques BOLA que possa ser integrado em ferramentas de validação de requisições a partir de especificações OpenAPI;
- A capacidade de detecção do método se some aos benefícios da adoção de padronização para APIs Web, aumentando o estímulo para que os fornecedores invistam na especificação da API como parte importante do processo de desenvolvimento do serviço.

1.4 Estrutura do Documento

Este documento está organizado em 6 capítulos (incluindo este). No Capítulo 2 são apresentados os conceitos necessários para a compreensão dessa pesquisa. O Capítulo 3 descreve os trabalhos relacionados a temática de detecção de ataques BOLA em APIs Web. No Capítulo 4 é apresentada a metodologia proposta para solução e sua implementação, enquanto o Capítulo 5 descreve os resultados obtidos com a experimentação do método em APIs vulneráveis. Por fim, o Capítulo 6 é reservado para a conclusão do trabalho e expectativas para trabalhos futuros.

Capítulo 2

Conceitos Básicos

Neste capítulo são apresentados os principais conceitos relacionados ao tema de detecção de ataques BOLA a APIs Web. São abordadas as partes envolvidas no projeto dessas APIs e definições de vulnerabilidade aplicadas ao contexto, visando auxiliar o melhor entendimento sobre o desenvolvimento do trabalho.

2.1 API Web

Uma API (*Application Programming Interface*) é uma interface utilizada como ferramenta para que diferentes componentes de uma arquitetura possam interagir programaticamente. Em específico, uma API Web é uma interface que possibilita que um sistema Web se comunique com outros sistemas através de requisições. APIs Web são frequentemente implementadas utilizando o padrão de design REST via requisições HTTP (*Hypertext Transfer Protocol*), devido a praticidade e desempenho oferecido pelas características do design REST somado ao uso difundido do protocolo HTTP como meio de comunicação na Web.

2.1.1 REST

REST (*Representational State Transfer* ou Tranferência Representacional de Estado) é um estilo de arquitetura de software descrito inicialmente na tese de doutorado de Fielding (2000), que define requisitos para que aplicações web respondam solicitações de serviços para seus clientes, manipulando representações textuais de recursos usando um conjunto uniforme de operações independentes de estado (*stateless*). O estilo de design REST foi derivado das seguintes seis considerações, que portanto são requisitos para ele:

- **Arquitetura cliente-servidor**: Clientes e servidores devem ser mantidos e desenvolvidos de maneira independente, se comunicando exclusivamente através da interface, e devem poder ser modificados independentemente;

- ***Statelessness***: Clientes devem adicionar às requisições todos os dados necessários para subsidiar a resposta, de forma que não existem controles de estado e sincronia da comunicação no servidor. Cada requisição é, portanto, tratada pelo servidor como uma nova requisição;

- ***Cacheability***: Determinados recursos podem ser sinalizados no servidor como cacheáveis, diminuindo o volume de tráfego e aumentando a performance tanto do servidor como do cliente;
- **Comunicação em camadas**: A arquitetura do servidor e deve ser transparente para o cliente, de forma que estes podem lidar com armazenamento, autenticação, comunicação e etc, em entidades separadas, não sendo distinguível para o cliente se este está lidando diretamente com o servidor ou através de intermediários;
- **Interface uniforme**: A interface uniforme é essencial para o design REST, uma vez que esta permite o desacoplamento da arquitetura, simplificando o desenvolvimento de cada parte individual. Este requisito é dividido em outros quatro sub requisitos:
  - **Identificação de recursos nas requisições**: Os recursos são identificados nas requisições e separados das representações retornadas para os clientes. Ou seja, o mesmo recurso pode ser retornado em representações diferentes de acordo com a requisição.
  - **Manipulação de recursos por representações**: De porte de qualquer representação de um recurso, um cliente tem informação suficiente para efetuar requisições posteriores utilizando o mesmo recurso;
  - **Mensagens auto descritivas**: Cada mensagem deve possuir informação suficiente para explicitar como esta deve ser processada;
  - **Hipermídia como motor do estado da aplicação** (HATEOAS, do inglês *Hypermedia as the engine of the aplication state*): a partir do acesso a um dos *endpoints*, o cliente pode navegar dinamicamente entre os recursos acessáveis a partir do recurso anterior, sendo essas informações, usualmente URIs, retornadas em conjunto com o recurso;
- **Código sob demanda** (opcional): O servidor pode estender as funcionalidades do cliente, enviando a este o código necessário para a execução de um determinado recurso para o qual a aplicação cliente não possui suporte nativo.

As APIs são denominadas RESTful se desenvolvidas obedecendo todas recomendações de design de arquitetura REST (Richardson and Ruby, 2008). A utilização desses requisitos se mostra adequada para diminuir a complexidade de implementação no cliente, uma vez que o programador familiarizado com uma API RESTful tem maior facilidade em entender outras APIs que seguem o design, em muito devido ao requisito de interface uniforme, bem como a escalabilidade, uma vez que o desenvolvimento de cada parte é suficientemente desacoplado das demais. Partes de código podem ser enviadas pelo servidor e executadas no cliente, sob demanda, estendendo as funcionalidades deste, e recursos podem ser sinalizados como cacheáveis. Esse recurso também diminui os custos na ponta do servidor, diminuindo a carga sobre este, o que em conjunto com a ausência de controle dos estados da aplicação pelo servidor (*statelessness*) favorece o aumento do desempenho (Feng et al., 2009).

2.1.2 OpenAPI

A especificação OpenAPI, também conhecida como OAS (*OpenAPI Specification*), é um padrão de descrição de APIs REST derivado da especificação Swagger, a qual foi doada para a Linux Foundation pela SmartBear Software como parte da OpenAPI Initiative, uma iniciativa que visa difundir os benefícios da padronização na descrição de APIs. Essa iniciativa é formada por

diversos membros, entre eles grandes empresas de software, como Google, IBM, Ebay, Oracle, dentre outras[1].

Com a padronização, um cliente pode entender e interagir com um servidor remoto com o mínimo de implementação, sendo um dos objetivos que esta seja facilmente interpretável tanto para humanos como para máquinas, sem ambiguidades. Assim, a capacidade e lógica do funcionamento de um servidor através de uma API REST pode ser entendida sem acesso a documentação e implementação. Uma descrição da API que adere a OAS, conhecida como OAS *Document*, pode ser usada para automatizar a geração de servidores e clientes nas mais diversas linguagens, além de servir como base para implementação de ferramentas de teste, análise, entre outras.

A especificação descreve informações sobre a API, sobre o fornecedor, termos de uso, endereço dos servidores e caminho para acesso aos recursos, formatos e requisitos para requisições e respostas, dentre outras. Um OAS *Document* pode ser descrito nos formatos JSON (*JavaScript Object Notation*) ou YAML (*YAML Ain't Markup Language*, um acrônimo recursivo), sendo ele mesmo em estrutura um objeto JSON, ou seja, seguindo a notação de objetos da linguagem JavaScript.

Os terminais onde cada serviço pode ser requisitado pelo cliente são determinados *endpoints* e são usualmente representados por URLs (do inglês *Uniforme Resource Locators*) (Jin et al., 2018). As nomenclaturas utilizadas na OAS para os métodos aceitos por um *endpoint* são as mesmas utilizadas na padronização do HTTP, que define 9 tipos de métodos de requisição[2,3]:

- GET: Transfere uma representação atual do recurso alvo;
- HEAD: Transfere o status e cabeçalho do recurso alvo;
- POST: Realiza um processamento no recurso alvo utilizando o enviado no *payload* da requisição;
- PUT: Substitui todas as representações do recurso alvo pelas descritas no *payload* da requisição;
- DELETE: Remove todas as representações do recurso alvo;
- CONNECT: Estabelece uma conexão com o servidor, identificada pelo recurso alvo;
- OPTIONS: Descreve as opções disponíveis para o recurso alvo;
- TRACE: Realiza um teste enviando uma mensagem por todo caminho entre o cliente e o recurso alvo (útil para *debug*);
- PATCH: Realiza uma modificação parcial no recurso alvo.

A especificação OAS manteve os valores de versionamento da especificação Swagger, que estava na versão 2.0 (e passou a se chamar OAS 2.0). Atualmente, a especificação encontra-se na versão 3.1.0.

A partir da versão 3.0 foram incluídos como parte da especificação *Links* (ou *Link Objects*). Com links, podem ser modeladas as relações entre valores retornados por uma operação e sua utilização em operações futuras. *Links* são essenciais para descrição do funcionamento considerando o item HATEOAS do requisito de Interface Uniforme do design REST, de posse

[1] A lista completa de empresas que fazem parte da iniciativa está disponível em: <https://www.openapis.org/membership/members>, acesso em 26/02/2022.

[2] RFC 7231 - Disponível em: <https://tools.ietf.org/html/rfc7231>.

[3] RFC 5789, seção 2 - Disponível em: <https://tools.ietf.org/html/rfc5789>.

dos identificadores de recursos retornados por um *endpoint*, um cliente deve ser capaz de navegar dinamicamente para outros *endpoints* acessáveis a partir deste, utilizando os identificadores retornados por esse *endpoint*.

2.2 OWASP

A fundação OWASP (Open Web Application Security Project) é uma entidade sem fins lucrativos cujo objetivo é trabalhar em melhorias para segurança de softwares e da Web. A entidade é conhecida pelo seus projetos Top 10, onde são elencadas por ordem de classificação as dez maiores vulnerabilidades de determinado segmento, baseando-se em um conjunto de dados coletados de ocorrências de explorações dessas vulnerabilidades e em surveys realizados junto a comunidade de profissionais da área.

Em 2017, a entidade divulgou a segunda versão do *Top 10 Most Critical Web Application Security Risks* (OWASP, 2017), contendo as maiores vulnerabilidades encontradas em aplicações web. Destas, nove continham algum subcomponente relacionado a APIs. Isso fez com que a instituição propusesse um novo projeto somente voltado para segurança de API's, batizado de *API Security Top 10*, lançado em 2019 (OWASP, 2019).

Em 2021, a entidade divulgou a última versão até esta data do *Top 10 Most Critical Web Application Security Risks* (OWASP, 2021). Nessa, a classe de vulnerabilidades *Broken Access Control* subiu da 5ª para a 1ª posição no ranking, sendo mais prevalente em aplicações que qualquer uma das outras categorias.

2.2.1 *API Security Top 10* 2019 e BOLA

Foram incluídas na lista de mais relevantes as seguintes vulnerabilidades encontradas em APIs, considerando o ano de 2019:

- API1:2019 -***Broken Object Level Authorization*** (Quebra de autenticação em nível de objeto): APIs possuem essa vulnerabilidade quando falham em verificar se um usuário possui as permissões necessárias para acessar o recurso solicitado;
- API2:2019 - ***Broken User Authentication*** (Quebra de autenticação em nível de usuário): Descreve falhas no mecanismo de autenticação de APIs, como ausência de encriptação, adoção de chaves criptográficas fracas e ausência de mecanismos para evitar ataques de força bruta;
- API3:2019 - ***Excessive Data Exposure*** (Exposição excessiva de informação): APIs possuem essa vulnerabilidade quando expõem dados sensíveis que não serão utilizados pela aplicação, ou que não deveriam ser retornados para os usuários;
- API4:2019 - ***Lack of Resources and Rate Limiting*** (Falta de limitação de recursos e taxa de acesso): Descreve APIs que não empregam mecanismos de limitação de consumo de recursos, como limites na taxa de requisição e no tamanho das requisições recebidas;
- API5:2019 - ***Broken Function Level Authorization*** (Quebra de autenticação em nível de função): Quando APIs não possuem um claro isolamento entre operações administrativas e regulares, elas tendem a possuir falhas de autorização quanto a verificação do nível de hierarquia do usuário na realização dessas operações;
- API6:2019 - ***Mass Assignment*** (Atribuição em massa): Se a API não aplica corretamente filtragens sobre os dados recebidos e enviados, atacantes podem ser capazes de modificar propriedades de recursos que não deveriam;

- API7:2019 -***Security Misconfiguration*** (Erro em configurações de segurança): Como resultado de configurações impróprias ou incompletas, mensagens de erro podem expor dados sensíveis da aplicação e serviços usados por APIs armazenamento de dados na nuvem, por exemplo) podem ficar expostos, levando ao comprometimento desses serviços;
- API8:2019 - ***Injection*** (Injeção de código): Se as entradas de dados das APIs não forem corretamente sanitizadas, elas podem receber dados maliciosos, que podem ser interpretados como código, ao invés de dados, pelo servidor;
- API9:2019 - ***Improper Assets Management*** (Administração imprópria de recursos): Em ecossistemas de APIs, a ausência de um inventário das APIs (documentação) pode levar versões antigas ou descontinuadas, potencialmente com problemas de segurança, a permanecerem acessíveis, resultando eventualmente no vazamento de dados sensíveis;
- API10:2019 - ***Insufficient Logging and Monitoring*** (Registros e monitoração insuficientes): A ausência de logs, de um nível apropriado de detalhes e de monitoração constante desses registros permite que atividades maliciosas permaneçam indetectáveis por longos períodos de tempo.

A OWASP utiliza uma metodologia própria de avaliação de risco para classificar as vulnerabilidades em mais ou menos relevantes. Cada uma das vulnerabilidades recebeu notas de 1 a 3 de acordo com o grau de risco para os seguintes critérios: Explorabilidade (mais difícil a mais fácil de explorar), prevalência (menos a amplamente presente), detectabilidade (mais fácil a mais difícil de detectar) e impacto (menos a mais severo).

Nesse trabalho, focaremos na vulnerabilidade que se encontra no topo da lista, ***Broken Object Level Authorization*** (ou BOLA). A vulnerabilidade foi classificada com nota 3 em explorabilidade, prevalência e impacto, e nota 2 em detectabilidade, ou seja, é facilmente explorável, aparece frequentemente, tem impacto técnico severo e tem dificuldade de detecção mediana.

Essa vulnerabilidade ocorre quando uma aplicação garante acesso a determinado grupo de recursos baseada apenas em entradas de dados do usuário, não validando se o recurso está no escopo de autenticação deste usuário. Nesse cenário, um atacante manipula identificadores que se referem a um determinado recurso, obtendo assim acesso a outros recursos ao qual este não deveria ter acesso. Isso acontece pelo fato de servidores desenhados pelas diretivas de design REST, por definição, não possuírem um controle rígido de estados para os clientes (requisito de *statelessness*), se baseando nos identificadores para controlar o acesso ao recurso que são enviados pelo cliente (requisito de Interface Uniforme).

A fim de facilitar o entendimento, considere como exemplo de aplicação vulnerável a BOLA uma aplicação web onde um usuário pode fazer seu login e visualizar suas informações pessoais, ilustrada na Figura 2.1.

Na Figura 2.1, a aplicação cliente inicialmente realiza o login, envia uma requisição *POST* com as credenciais do usuário para o *endpoint* `/login`, recebendo como resposta um número identificador do usuário (um ID), o *ID01*, que deverá ser utilizado nas requisições seguintes. Em seguida, de posse do identificador do usuário, a aplicação realiza uma requisição *GET* ao *endpoint* `/accounts`, obtendo como resposta as informações do usuário, cujo ID é (*ID01*) *User1*. Em seguida, o ataque se dá com uma alteração (ou *tampering*) na requisição, na pela qual o usuário malicioso, substituindo o ID original por outro, consegue acesso às informações de outros usuários. Essa alteração pode se dar de duas formas:

1. O atacante pode interceptar e alterar a requisição em tempo de execução, através de um proxy;

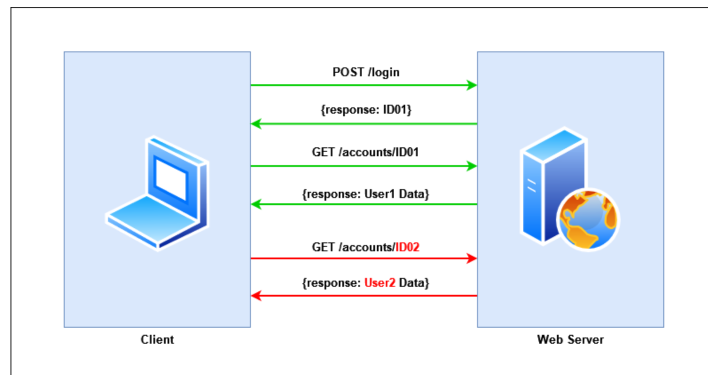

Figura 2.1: Exemplo de aplicação vulnerável a BOLA. Fonte: Elaborada pelo autor.

2. O atacante pode criar uma requisição idêntica a legítima, alterar os atributos desejados e enviá-la ao servidor.

A vulnerabilidade, apesar de simples, é bastante comum, pois ainda que haja implementação de infraestruturas robustas de segurança, essas muitas vezes não são aplicadas individualmente aos recursos pelos desenvolvedores a cada acesso, e a gravidade de um ataque bem sucedido pode variar de acesso, modificação ou destruição de informação sensível até controle completo sobre a conta de outros usuários.

A vulnerabilidade BOLA é uma refatoração da vulnerabilidade IDOR (*Insecure Direct Object Reference*), presente na lista original de vulnerabilidades críticas a aplicações web publicada pela OWASP em 2013 (OWASP, 2013). A vulnerabilidade foi reconceituada pelo fato do nome anterior não definir especificamente o problema central da vulnerabilidade, que não é o acesso direto aos objetos através das referências e sim sobre a falha na autorização para acesso a estes através das referências.

2.3 Considerações sobre o Capítulo

Neste Capítulo foram vistos os conceitos necessários para a compreensão desta proposta. Primeiro foi descrito o conceito de API Web, seguido pelo padrão de design REST e seus requisitos, sendo esse predominante no processo de desenvolvimento desse tipo de API, seguido da iniciativa OpenAPI e sua especificação de APIs REST. A seguir, foi apresentada e fundação OWASP, uma entidade sem fins lucrativos que trabalha em melhoreis de segurança para a Web, e seus projetos Top 10, que elencam as principais vulnerabilidades em determinado segmento. As vulnerabilidades que formam o *API Security Top 10* foram brevemente descritas. Em seguida, foi conceituada e exemplificada a vulnerabilidade que se encontra em primeiro lugar na lista, Broken Object Level Authorization (BOLA), cuja detecção é alvo deste trabalho.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são apresentados trabalhos consideradas relevantes para o tema, incluindo abordagens de modelagem de fluxos de comunicação em APIs RESTful, detecção de comunicação anômala e análise de especificações OAS para extração de informação que permita navegação através de HATEOAS.

Os trabalhos relacionados foram inicialmente obtidos através da metodologia de *snowball sampling*. Após isso, para obter trabalhos relacionados ao tema de detecção de ataques BOLA de maneira mais rigorosa, foi realizada uma revisão sistemática da literatura.

3.1 Revisão Inicial da Literatura

Devido ao caráter recente da preocupação sobre a segurança de APIs Web, parte do material disponível para pesquisa tem caráter prático e vem de entidades formadas por profissionais, em contraste com trabalhos realizados por instituições acadêmicas. Assim, apenas uma revisão sistemática recente e vaga, relacionada ao tema, foi encontrada. Em outras palavras, não foram encontradas revisões sistemáticas diretamente relacionadas a temática de segurança em APIs Web, sendo a quantidade de trabalhos acadêmicos relacionados pequena. Dessa forma, foi adotada a metodologia de revisão de trabalhos em bola de neve (*snowball sampling*), a partir de artigos cuja base pudesse ser adaptada a detecção de ataques BOLA em APIs Web.

Além disso, foram consideradas definições de vulnerabilidades estabelecidas em recursos reconhecidos (Por exemplo, o OWASP Top 10 como um classificador confiável de ataques mais relevantes a APIs Web), mas que ainda não foram propriamente formalizadas em torno de trabalhos teóricos, algo similar ao sugerido por Sureda Riera et al. (2020) em sua revisão sistemática sobre detecção de anomalias para prevenção de ataques. Considerando que a vulnerabilidade BOLA é uma refatoração da vulnerabilidade IDOR, já identificada há alguns anos, foram pesquisadas também técnicas de detecção desta vulnerabilidade.

Inicialmente, não foram encontrados trabalhos no contexto de tentativa de exploração de APIs Web. Foram encontradas diretivas de prevenção a vulnerabilidade em nível de desenvolvimento, dentre as quais se destaca o trabalho de KumarShrestha et al. (2015), que elencou as principais medidas existentes. Essas medidas envolvem a manipulação mais robusta dos identificadores e da autenticação para acesso a um recurso a nível de desenvolvimento. Isso denota que os avanços na detecção desse tipo de ataque em tempo de execução têm sido lentos, sendo os esforços focados no processo de desenvolvimento.

3.1.1 Trabalhos identificados por *Snowball Sampling*

3.1.1.1 RESTalk

Ivanchikj (2016) propôs uma Linguagem de Domínio Específico (ou DSL, acrônimo em inglês para *Domain Specific Language*) para modelar um fluxo de mensagens entre cliente e servidor, considerando que esses obedecem aos requisitos do padrão de design REST. O principal objetivo era a obtenção de uma ferramenta que permitisse a compreensão do comportamento dinâmico de um fluxo de comunicação entre cliente e servidor, considerando que a obtenção de recursos envolve todo um processo de comunicação e não apenas uma requisição. Essa linguagem foi batizada de **RESTalk**.

Posteriormente, foi desenvolvida uma forma de visualizar essa "conversa" entre cliente e servidor (Ivanchikj et al., 2018b), possibilitando uma maior capacidade de compreensão. Os autores consideraram que a comunicação pode ser guiada pelo fluxo de hipermídia (ou de *links*) e que pode ser difícil para o cliente determinar o fluxo de requisições necessárias para a obtenção de um recurso específico apenas olhando a documentação do funcionamento estático da API. Assim, o uso da linguagem para modelar a conversa poderia passar a fazer parte do processo de desenvolvimento, auxiliando na representação de fluxos de comunicação em diferentes cenários de uso e clarificando o caminho necessário para a obtenção e cada recurso. Nesse trabalho, a interface foi avaliada por projetistas de APIs REST, os quais deram seus *feedbacks* aos autores sobre quais cenários eram atendidos pela ferramenta e quais não eram.

Em seguida, foi desenvolvida uma ferramenta para detecção de padrões nessas conversas, através do uso de logs (Ivanchikj et al., 2018a), o que possibilitou investigações mais profundas sobre o comportamento de uma API, e permitiu buscas por padrões conhecidos ou extração automática de padrões dos registros.

Todos os trabalhos para o desenvolvimento da ferramenta **RESTalk** demonstram que a análise de uma entrada isolada não é suficiente para modelar o comportamento da comunicação, ainda que pelos requisitos de design REST uma requisição deva conter toda informação necessária para que o servidor a responda satisfatoriamente. O processo de comunicação não é atômico, ainda que não existam controles de estado do lado do servidor. Assim, a comunicação deve seguir um determinado fluxo para ser bem sucedida e esse fluxo pode ser modelado e visualizado.

3.1.1.2 RESTler

Em Atlidakis et al. (2019) e Atlidakis et al. (2020) foi introduzida a primeira ferramenta de *fuzzer stateful* para APIs REST, o **RESTler**. Essa ferramenta permite testar, de forma automatizada, APIs REST a partir de especificações no formato *Swagger* (OAS 2.X e inferiores), a fim de detectar erros de funcionamento, vulnerabilidades e garantir determinadas propriedades de segurança.

A **RESTler** foi desenvolvida para se contrapor à ferramentas de teste que se baseiam no armazenamento das requisições e respostas em uma sessão, que posteriormente são reproduzidas para a realização dos testes, ou seja, o fluxo de comunicação é armazenado e utilizado para recriar os cenários de teste. Contudo, esse fluxo explora cenários restritos, uma vez que se baseia no funcionamento regular da API, o que limita o escopo de erros que esses testes podem encontrar.

Em alguns cenários de funcionamento, as cadeias de requisições de teste são geradas utilizando dependências produtor-consumidor extraídas de descrições de APIs (*Swagger*). Uma das abordagens dessa ferramenta consiste exatamente em "*inferir que uma requisição B precisa ser executada depois da requisição A, pois a requisição B tem como entrada um recurso x produzido*

por A" (Atlidakis et al., 2019). Assim, a ferramenta pode automatizar um teste que necessite de um determinado fluxo de comunicação entre cliente e servidor.

Embora os artigos não deixem claro como a extração das dependências é feita, recentemente o projeto tornou-se de código aberto, permitindo análises visando a extração de pontos importantes à nossa implementação. Contudo, é preciso destacar que a ferramenta é voltada para detecção de problemas no funcionamento de APIs através de técnicas de *fuzzing*, não tendo pretensão de detecção de problemas de segurança em específico.

3.1.1.3 Detecção de APIs Scrapers através do fluxo de requisições

No trabalho de Filho and Feitosa (2019) foi demonstrado que, utilizando a diagramação **RESTalk**, o comportamento típico de *API Scrapers* (*bots* com o objetivo de extrair informações de uma API) é fundamentalmente diferente de um cliente legítimo da API, permitindo a identificação desses bots. Enquanto a sequência de requisições de um cliente legítimo obedece a dependência entre os *endpoints* da API e o fluxo de uso da aplicação, os *API Scrapers* requisitam recursos diretamente, explorando o conhecimento do funcionamento da API em si, ou empregando algoritmos de força bruta, sem necessariamente seguir a dependência estabelecida na aplicação, de forma semelhante a um ataque BOLA.

No entanto, o processo de construção diagrama **RESTalk** é manual, se tornando assim mais suscetível a erros. Além disso, toda mudança na API demanda a revisão do documento por um especialista, o que dificulta sua utilização para automatização da detecção de ataques.

3.1.1.4 Suporte a Navegação via HATEOAS

O trabalho de Kotstein and Decker (2020) se propôs a utilizar descrições de APIs Web para dar suporte a navegação via HATEOAS, uma vez que os fornecedores de serviço usualmente fornecem descrições do funcionamento ao invés de retornar diretamente *hyperlinks* às requisições.

Para isso, os autores levantaram os principais recursos disponibilizados por ferramentas de descrição de APIs, considerando que essas ferramentas não são eficazes em descrever a relação de conexão entre os *endpoints*, essenciais para uma navegação entre os recursos da API baseada no item HATEOAS do requisito de Interface Uniforme do padrão REST. Nesse levantamento, os autores detectaram que apenas duas ferramentas de descrição possuíam algum suporte a descrição do fluxo de hipermídia (WADL e OpenAPI), sendo OpenAPI a única relevante. Os autores destacaram que o recurso oferecido pela especificação OpenAPI (descrição de *Links*) é pouco utilizado em descrições de APIs, sendo encontrado em somente 3 de 1611 especificações analisadas.

A técnica utilizada pelos autores para injetar os *links* entre as requisições apresentou muitos falsos positivos, por se basear em comparações de strings utilizadas nos identificadores dos recursos. Quando esses identificadores são genéricos (ex.: 'id') para vários recursos, esses levaram a ambiguidades no processo de determinação das relações.

3.2 Revisão Sistemática da Literatura

A revisão sistemática da literatura (RSL) foi realizada seguindo o protocolo sugerido por Kitchenham (2004) e motivada pela necessidade de levantar uma maior quantidade de literatura relacionada, de maneira mais rigorosa e de modo a dar embasamento e consistência ao trabalho. O protocolo de revisão sistemática utilizado será descrito a seguir.

3.2.1 Protocolo de revisão sistemática

Para realização da revisão sistemática, conforme orientado por Kitchenham (2004), é necessário definir o protocolo de revisão que será executado a priori. Esse protocolo é importante por alguns motivos, como definição dos objetivos, escopo e para que a revisão possa ser reproduzida posteriormente.

Inicialmente, temos como enunciado do **objetivo** da revisão sistêmica: "Encontrar métodos de detecção e prevenção de ataques BOLA (Broken Object Level Authorization ou Quebra de Autenticação em Nível de Objeto) a APIs Web". Esse objetivo nos leva a **questão primária** da revisão: "Quais os métodos adotados na literatura para detecção e prevenção desse tipo de ataque (ou similares)?" e a **questão de secundária** da revisão: "Quais são suas vantagens e desvantagens?".

Foi escolhida como base de conhecimento para a revisão **a base Scopus, da Elsevier** e como idioma a **língua inglesa**, idioma dos artigos indexados pela base. A partir daí, ainda seguindo a metodologia proposta por Kitchenham (2004), foi construída e refinada a string de busca utilizada para recuperar os artigos na base. Essa string foi construída conforme a abordagem sugerida, adotando a escolha de palavras chaves através da separação por "População", "Intervenção" e "Resultado" (*Outcome*).

- **População**: APIs REST Web;
- **Intervenção**: Broken Object Level Authorization (BOLA), ou Insecure Direct Object Reference (IDOR);
- **Outcome**: Detecção, Prevenção, Combate e similares.

A string foi então refinada para utilizar palavras mais abrangentes e retornar resultados mais específicos na área de Computação. Em sua versão final adaptada para consulta a base Scopus:

("API" OR "APIs" OR "api" OR "programming interface")
AND
("web" OR "WEB" OR "REST" OR "RESTful" OR "Restful")
AND
("Broken Object Level Authorization" OR "BOLA" OR "Insecure Direct Object Reference" OR "IDOR" OR "tampering" OR "broken authorization")
AND
("Analysis" OR "Detection" OR "Detecting" OR "prevent" OR "Prevention" OR "Preventing" OR "Mitigation" OR "Mitigating" OR "fighting" OR "test" OR "testing" OR "review")
AND
(SUBJAREA(COMP) OR SUBJAREA(ENGI))

Essa versão da string retornou **111 documentos** que serviram de base inicial para filtragem para extração de dados. Para selecionar os mais relevantes para esse levantamento, foram adotados os seguintes critérios:

- **Inclusão**:
  - **I1** – Incluir artigos cujos título e resumo reflitam relação com a temática de segurança no processo de comunicação entre clientes web e servidores. Em caso de dúvida, decisão será inclusão.

- **Exclusão**:
  - **E1** – Excluir artigos não disponíveis para leitura (artigos pagos não incluídos pelo sistema CAFe da CAPES ou não localizados);
  - **E2** – Excluir artigos cuja leitura da introdução não indique relação com ataques de quebra de autenticação a nível de objeto;
  - **E3** – Detectado conteúdo coincidente de um mesmo autor entre dois artigos, excluir um desses artigos, dando preferencia (para permanência) ao artigo mais abrangente ou mais novo em caso de duplicação.

Os resultados da aplicação desses critérios estão elencados no quadro da Figura 3.1.

Documentos inicialmente retornados pela ***string* de busca**	Após aplicação do filtro de **Inclusão – I1**	Após aplicação do filtro de **Exclusão – E1**	Após aplicação do filtro de **Exclusão – E2**	Após aplicação do filtro de **Exclusão – E3**	Documentos selecionados para **extração de dados**
111	40	36	11	10	**10**

Figura 3.1: Resultado da Aplicação dos Critérios. Fonte: Elaborada pelo autor.

Ao final da extração, foram identificadas as técnicas relacionadas a detecção de ataques do tipo BOLA (ou similares) e suas vantagens e desvantagens. O compilado dos resultados encontrados será descrito na próxima seção.

3.2.2 Trabalhos identificados na Revisão Sistemática

O trabalho de Prokhorenko et al. (2016) é uma revisão da literatura que tenta classificar as técnicas utilizadas para proteção de APIs web. Para isso, os autores buscam levantar traços comuns entre os tipos de abordagem e os tipos de ataques que estas visam detectar ou prevenir. Os autores classificam os tipos de técnicas utilizando dois critérios: (1) **Base de decisão** (estatística, política ou intenção) e (2) **Sujeito observado** (Entradas, Aplicação ou Saídas).

Os autores detalham diversas categorias de métodos de prevenção que são gerados a partir dos critérios, como abordagens de caixa preta ou caixa branca, proteções de injeção, proteções de ataques XSS, dentre outros. Além disso, levantam como um problema recorrente de segurança a existência de código legado, desenvolvido utilizando tecnologias menos seguras ou desatualizados, devido ao alto custo para desenvolver novamente as mesmas funcionalidades, o que leva a um aumento na dificuldade de detecção e maior prevalência de vulnerabilidades.

Dentro do contexto relacionado a ataques BOLA, os autores citam como técnicas comumente utilizadas para proteção de APIs Web o uso de controles de autenticação. Para validação de parâmetros, os autores apontam técnicas de sanitização e o uso de alarmes para requisições anômalas.

Sobre o uso de controles de autenticação, os autores apontam como limitação que este é difícil de generalizar entre diferentes aplicações, uma vez é dependente de implementação, ficando a cargo do desenvolvedor a responsabilidade de fazer cumpri-los. Para as técnicas de sanitização dos parâmetros e detecção e alarme de requisições anômalas, os autores apontam que é difícil selecionar as características corretas a serem observadas para as detecções. Assim, estas comumente

são implementadas se baseando na experiência do desenvolvedor, e não em metodologias formais.

O trabalho de Fungy et al. (2014) propõe uma melhoria em técnicas de testes em API baseados em *fuzzing*. Os autores destacam a dificuldade de ferramentas de testes baseadas em *fuzzing* para detecção de vulnerabilidades quanto a modificação de parâmetros em requisições, e propõem uma ferramenta sensível a sequência de estados entre as requisições para validação destes parâmetros.

As vulnerabilidades referentes a modificação de parâmetros normalmente aparecem quando, durante a implementação, estes são considerados imutáveis devido a regras de negócio na aplicação cliente, desconsiderando a possibilidade de intervenções nas requisições. A ferramenta proposta atua identificando quais parâmetros são reinseridos entre as requisições, sinalizando estas como dependentes destes parâmetros. Além disso, a ferramenta tenta identificar, ao refazer certas requisições, quais parâmetros são *tokens* utilizados somente uma vez por requisição. Dessa forma, a ferramenta consegue realizar testes automatizados executando requisições em uma sequencia válida, o que é capaz de estender a quantidade de rotas verificada com sucesso.

Os autores afirmam que a ferramenta foi capaz de identificar, em dois bancos diferentes, sequencias de requisições que levaram a exploração de vulnerabilidades.

Monshizadeh et al. (2014) propuseram a ferramenta MACE, que utiliza análise estática para verificar, no código da aplicação web, a existência de controles que façam cumprir as checagens de autenticação, visando mitigar, ataques que se baseiam em escalada de privilégios.

As escaladas de privilégio podem ser de dois tipos: **horizontal**, quando o atacante tenta acessar dados de outros usuários com o mesmo nível de autenticação, ou **vertical**, quando o atacante tenta obter privilégios superiores aos que possui. A ideia é verificar se, para um dado papel (com um determinado nível de permissão), as permissões verificadas ao longo do código são consistentes entre si. A premissa é que, caso estas sejam consistentes, isso é um forte indicativo de que os mecanismos de segurança evitam escalada de privilégios.

Os autores informam que executaram sua ferramenta em 7 códigos *open source*, e que destes, 5 apresentaram vulnerabilidades identificadas.

Mendoza and Gu (2018) propõem uma ferramenta (WarDroid) para detecção de inconsistência na validação de entradas entre a aplicação e APIs em dispositivos móveis. A manutenção de consistência na validação dos dados inseridos na aplicação cliente deve ser replicada do lado do servidor, caso contrário, vulnerabilidades podem existir e serem exploradas através da análise das requisições feitas pela API.

A técnica utilizada pela ferramenta realiza análise estática no código da aplicação para um dispositivo Android (APK), de forma a obter as partes do código que realizam requisições. Depois, utilizam essas partes de código para gerar requisições HTTP(S) que podem vir a ser utilizadas para explorar a API. Como vantagens da análise estática, os autores destacam a eficiência. Como desvantagens, destacam a possibilidade de lidar com código ofuscado (o que dificulta a utilização), bem como dificuldades de lidar com controles de estado (que dependem da dinâmica da aplicação) e analisar código sob demanda (não nativo).

Os autores afirmam ter testado a ferramenta em 10,000 aplicações da loja (Google Play). Dessas 4,562 foram sinalizadas como potencialmente vulneráveis. Os autores então montaram um subconjunto com 1,000 requisições que deveriam ser inválidas, escolhidas aleatoriamente dentre as detectadas para essas aplicações. Dentre essas, 884 foram aceitas pela API, confirmando a vulnerabilidade. Os autores citam ainda exemplos de aplicações dentre esse conjunto onde é possível explorar as vulnerabilidades existentes e obter dados sem autorização, fazer compras de graça, fazer injeção de conteúdo, injeção de código SQL, negação de serviço e até

transferências bancárias.

A ferramenta BLOCK, proposta por Li and Xue (2011), se baseia no conceito de caixa preta para identificar requisições que utilizem estados inválidos para explorar vulnerabilidades no servidor. A ferramenta foi desenvolvida dando suporte a aplicações PHP.

Os autores indicam três tipos de vulnerabilidades que podem ser inseridas na sessão, em tempo de desenvolvimento: variáveis insuficientes que determinem o estado da sessão, checagem insuficientes das variáveis de sessão ao longo do código e erros na checagem dessas variáveis que permitam que estas checagens sejam evitadas.

A técnica proposta pelos autores na ferramenta consiste em definir e identificar, observando as requisições, parâmetros invariantes dentro de algumas regras definidas. A partir daí, esses parâmetros são utilizados para formar *templates* para as requisições. Essas requisições são executadas, e conforme as páginas recebidas como resposta, são inferidas entre válidas ou inválidas.

Os autores citam como comparação a utilização de controles autenticação, já presentes na maior parte *frameworks* de desenvolvimento. Os autores citam como desvantagens dos controles de autenticação a necessidade de serem utilizados pelos desenvolvedores, que normalmente tem como prioridade a entrega de funcionalidades. Essa seria uma vantagem da abordagem utilizada, a possibilidade de utilização para diferentes linguagens e *frameworks*. Como desvantagem, os autores citam a dificuldade de generalização da forma de modelagem de estados, uma vez que são dependentes da implementação.

Duas ferramenta baseada tanto na análise estática de código fonte quanto em requisições foram propostas por Bisht et al. (2014). Uma das ferramentas, com análise em caixa preta (NoTamper), utilizando código fonte do cliente e a análise das requisições. A outra (WAPTEC), com análise em caixa branca, utilizando além do código fonte do cliente e as requisições, o código fonte do servidor.

Os autores ressaltam que um dos motivos para aparecimento de vulnerabilidades é o uso de verificações sobre os parâmetros mais fortes do lado do cliente que do lado do servidor. Esse desnível é utilizado pelos autores para identificar vulnerabilidades na aplicação. Os autores incluíram no texto quatro casos de ataques a aplicações do mundo real envolvendo modificações de parâmetros nas requisições. Esses ataques foram feitos como provas de conceito e os responsáveis notificados para que as vulnerabilidades fossem corrigidas.

A abordagem utilizada utiliza o código do cliente como uma forma de inferir o funcionamento das implementações no lado do servidor para o caso de caixa preta, enquanto que o caso de caixa branca utiliza também o código do servidor. A abordagem de caixa branca foi considerada superior na maior parte dos critérios utilizados pelos autores, o que é esperado considerando o fato desta possuir bem mais informação sobre as validações adotadas do lado do servidor. Contudo, a abordagem de caixa preta é mais facilmente adaptável a diferentes tipos de tecnologias e servidores e seu uso pode ser generalizado mais facilmente.

O trabalho de Viriya and Muliono (2021) focou na demonstração de que 3 métodos comumente adotados por desenvolvedores de aplicações para dispositivos móveis (detecção de *Jailbreaking*, detecção de *root* e SSL *pinning*) não são suficientes para garantir a segurança das aplicações, através da apresentação de casos de *ethical hacking* como exemplos. Os autores descrevem 3 casos de ataques BOLA bem sucedidos a aplicações mobile, duas em aplicações de *e-commerce* e uma em aplicação de *e-banking*.

O artigo demonstra identificação de vulnerabilidades em APIs que utilizam como controles de autenticação verificações simplificadas, baseadas em presunções que não podem ser garantidas,

contudo, em um grupo de testes pequenos, o que dificulta a confiabilidade.

3.2.3 Respostas as Questões de Pesquisa

Após a realização da Revisão Sistemática da Literatura, observando as técnicas de detecção e prevenção para responder a questão principal de pesquisa **"Quais os métodos adotados na literatura para detecção e prevenção desse tipo de ataque (ou similares)?"**, compilamos os dados no quadro da Figura 3.2.

Artigos	Métodos de detecção / prevenção citados				
	Controles de Autenticação	Validação de parâmetros	*Fuzzing*	Análise Estática	Análise de Requisições
Viriya and Muliono (2021)	X				
Mendoza and Gu (2018)				X	
Prokhorenko et al. (2016)	X	X			
Monshizadeh et al. (2014)				X	
Fungy et al. (2014)			X		
Bisht et al. (2014)				X	X
Li and Xue (2011)	X				X

Figura 3.2: Compilado dos métodos de detecção/prevenção citados. Fonte: Elaborada pelo autor.

É possível verificar que os métodos de **controles de autenticação** e **análise estática** foram os mais citados dentre os artigos considerados mais relevantes no levantamento sistemático.

Para a resposta da questão de pesquisa secundária **"Quais são suas vantagens e desvantagens?"** podemos utilizar a Tabela 3.1, que elenca os valores das respostas para cada artigo relevante:

Para os métodos mais utilizados, podemos elencar:

- **Controles de autenticação**:
  - **Vantagens**: Como estes foram citados na maior parte para comparação, não foram descritas vantagens além destes já estarem presentes nos *frameworks* de desenvolvimento mais comuns (Li and Xue, 2011).
  - **Desvantagens**: Dependentes de implementação, dificultando a generalização (Prokhorenko et al., 2016), além de depender inteiramente do desenvolvedor (Li and Xue, 2011).
- **Análise estática**:
  - **Vantagens**: Eficiência, uma vez que a aplicação não precisa estar rodando para os testes (Mendoza and Gu, 2018).
  - **Desvantagens**: Dependente da disponibilidade do código-fonte, dificuldade de lidar com código sob demanda (não nativo) ou controle de estados (Mendoza and Gu, 2018).

Tabela 3.1: Contribuição dos artigos as questões de pesquisa

Trabalho	**Questão Primária**	**Questão Secundária**
Viriya and Muliono (2021)	O artigo demonstra identificação de vulnerabilidades em APIs que utilizam como controles de autenticação verificações simplificadas, baseadas em presunções que não podem ser garantidas.	Não contemplada.
Mendoza and Gu (2018)	O artigo demonstra a utilização de análise estática para detecção de vulnerabilidades de modificação de requisições.	Como vantagens da análise estática, os autores destacam a eficiência. Como desvantagens, os autores destacam a possibilidade de lidar com código ofuscado (o que dificulta a utilização), bem como dificuldades de lidar com controles de estado (que dependem da dinâmica da aplicação) e analisar código sob demanda (não nativo).
Prokhorenko et al. (2016)	Como técnicas comumente utilizadas para proteção de APIs Web os autores apontam o uso de controles de autenticação. Para validação de parâmetros, os autores apontam técnicas de sanitização e o uso de alarmes para requisições anômalas.	Os autores não apresentaram vantagens para as técnicas apontadas, apenas as limitações. Sobre o uso de controles de autenticação, este é difícil de generalizar entre diferentes aplicações, uma vez é dependente de implementação, ficando a cargo do desenvolvedor a responsabilidade de fazer cumpri-los. Para as técnicas de sanitização dos parâmetros e detecção e alarme de requisições anômalas, os autores apontam que é difícil selecionar as características corretas a serem observadas para as detecções. Assim, estas comumente são implementadas se baseando na experiência do desenvolvedor, e não em metodologias formais.
Monshizadeh et al. (2014)	A ferramenta utiliza técnicas de análise estática do código, verificando se ao longo do código, para um dado perfil de autenticação, se as verificações dessa autenticação são feitas de maneira homogênea. A premissa dos autores é que se estas verificações são feitas de maneira homogênea, é provável que este evite ataques baseados em escalada de privilégios.	Não contemplada.
Fungy et al. (2014)	Os autores apresentam uma tentativa de melhoria de técnicas de *fuzzing* para detecção de vulnerabilidades nas requisições efetuadas a APIs.	As técnicas de *fuzzing* tem como vantagens a possibilidade de automatizar testes dentro e fora do ambiente de produção. Como limitações, estas têm dificuldades para seguir os diferentes estados que definem uma requisição como válida, ou seja, considerando a sequência de requisições adotadas e parâmetros envolvidos nessas requisições. Essas limitações são o alvo dos autores, que visam detectar parâmetros necessários entre as requisições e a evitar a reutilização de tokens desenhados para serem utilizados somente uma vez.
Bisht et al. (2014)	Os autores utilizam dois tipos de análise estática do código (caixa preta e caixa branca) combinados com a análise das requisições. Abordagens de caixa preta levam em consideração somente a análise do código no lado cliente, enquanto que abordagens de caixa branca analisam tanto o código do cliente quanto do servidor.	Dentro da análise estática do código, os autores citam como vantagem das análises de caixa preta que estas são agnósticas quanto a linguagem do lado do servidor, o que a torna preferível para automatização em diferentes tecnologias. Abordagens de caixa branca, por sua vez, tendem a obter resultados mais precisos, uma vez que possuem mais informação.
Li and Xue (2011)	Os autores citam como comparação a utilização de controles autenticação, já presentes na maior parte frameworks de desenvolvimento. A ferramenta proposta utiliza a análise das requisições como forma de detecção em caixa preta. A detecção se baseia em encontrar parâmetros invariantes nas requisições para detecção de estados, que são utilizados para gerar *templates* de requisições para testes.	Os autores citam como desvantagens dos controles de autenticação a necessidade de serem utilizados pelos desenvolvedores, que normalmente tem como prioridade a entrega de funcionalidades. Os autores citam como vantagem da abordagem utilizada a possibilidade de utilização para diferentes linguagens e frameworks. Como desvantagem, os autores citam a dificuldade de generalização da forma de modelagem de estados, uma vez que são dependentes da implementação.

3.3 Discussão

O levantamento sistemático demonstrou que os métodos presentes nos artigos mais relevantes dentre os levantados tem como métodos de detecção e prevenção de ataques similares a BOLA mais citados Controles de Autenticação e Análise Estática de código-fonte. As vantagens e desvantagens encontradas para esses métodos foram elencadas na sessão anterior.

Contudo, trabalhos relacionados ao contexto específico de detecção de ataques BOLA ou similares a APIs WEB RESTful são escassos na literatura, aparecendo em pouca quantidade mesmo no levantamento sistemático. Os trabalhos encontrados inicialmente pelo levantamento de *snowball sampling* dão indícios de que APIs RESTful possuem determinadas características que podem ser utilizadas para a elaboração de ferramentas de detecção de ataque baseadas na compreensão da dinâmica da aplicação.

Os trabalhos de desenvolvimento da ferramenta RESTalk demonstram que a obtenção de um recurso não é direta, mesmo que no design REST o controle de estados esteja ausente do lado do servidor. Assim, para que o cliente obtenha o recurso de seu interesse, ele executa uma série de requisições, trilhando o caminho para a obtenção do recurso. Essa ideia foi identificada na revisão sistemática nos trabalhos de Fungy et al. (2014) e Li and Xue (2011), que verificaram que para executar requisições válidas nos cenários de testes e detecção, é necessário identificar parâmetros invariantes entre requisições, bem como parâmetros que não devem ser reutilizados.

Também de maneira similar a ideia do trabalho de Fungy et al. (2014), porém para o contexto de APIs REST, a ferramenta de testes automatizados para APIs REST em serviços de nuvem (RESTler) mostrou que ao considerar que requisições são executadas de maneira ordenada, é possível automatizar cenários de teste diretamente da descrição da API. A ordem de execução das requisições pode ser modelada como uma arquitetura produtor-consumidor, onde um *endpoint* produz um identificador primeiro, para posteriormente outro consumi-lo.

Esses trabalhos, quando trazidos para o contexto da detecção de ataques BOLA, servem como ponto de partida para a concepção de uma solução. Em cenários comuns, recursos não são acessados diretamente, ou seja, a aplicação legítima segue um fluxo de requisições que permitem ao cliente possuir as informações necessárias para executar uma requisição bem sucedida que retorne o recurso. Além disso, para diferenciar um fluxo legítimo de um anômalo, precisamos entender quais *endpoints* precisam ser navegados primeiro para que o cliente obtenha o recurso de interesse.

Isso é confirmado nas observações de Filho and Feitosa (2019), que utilizaram o RESTalk para demonstrar que o comportamento de API *Scrapers* é diferente do comportamento regular de um cliente legítimo. Isso porque os primeiros tentam executar requisições diretamente, sem seguir o fluxo de comunicação que seria seguido por um cliente legítimo. Essa observação pode ser estendida para mitigar ataques do tipo BOLA, que, nesse ponto, tem comportamento similar ao de API *Scrapers*.

Para determinar a ordem em que requisições são executadas, ou seja, classificá-las entre produtoras e consumidoras dentro do contexto de APIs Web, é necessário modelar o fluxo de navegação por hipermídia, ou seja, determinar conexões de sucessão entre os *endpoints*. Para isso, a especificação OAS a partir da versão 3.0 fornece suporte aos *Links*.

Essa abordagem tem a vantagem de poder funcionar em caixa preta, ou seja, sem depender da implementação do cliente ou do servidor, utilizando apenas o fluxo de comunicação através da API e uma especificação contratual desta.

No trabalho de Kotstein and Decker (2020) foi evidenciado que essa funcionalidade é pouco utilizada pelos fornecedores das APIs e que somente comparações entre os diversos recursos, sem considerar relações entre os *endpoints*, tem elevado risco de ambiguidade. Contudo, uma ferramenta eficaz para detecção de ataques poderia estimular a utilização maior dos *Links* na

descrição da API, que teria custo inferior ao custo de desenvolvimento para diminuir a incidência do ataque.

É neste contexto que a solução proposta irá atuar.

Capítulo 4

Método Proposto

Este Capítulo descreve o método proposto para detecção da vulnerabilidade BOLA como uma arquitetura capaz de tratar requisições em tempo de execução e realizar a detecção. Antes de apresentá-lo de fato, é preciso que inicialmente algumas definições, necessárias para o funcionamento da proposta, sejam apresentadas ao leitor.

4.1 Definições

Ainda que a ideia geral de modelar um processo entre elementos produtores e consumidores de um determinado recurso seja amplamente utilizada na literatura científica, para entendimento do método a ser proposto é preciso descrever o que são as relações de produtor-consumidor no contexto deste trabalho. Baseando-se na ideia utilizada por Atlidakis et al. (2019), para melhorias de testes automáticos em APIs, podemos conceituar as relações de produtor-consumidor utilizando três definições:

1. Um *endpoint* é **produtor** de um recurso quando, ao receber uma requisição, o servidor a processa e retorna um identificador válido (caso a requisição seja bem sucedida) para esse recurso. Do ponto de vista prático, o servidor pode ter criado um recurso novo e respondido com o identificador, ou pode ter acessado um recurso já existente e respondido com o identificador correspondente. Na Figura 4.1, o *endpoint1* é produtor do recurso;

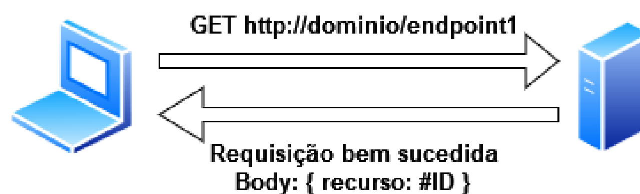

Figura 4.1: Exemplo de requisição HTTP para um *endpoint* produtor de um recurso. Fonte: Elaborada pelo autor.

2. Um *endpoint* é **consumidor** de um recurso quando requer um identificador desse recurso para que a requisição seja considerada válida. Na Figura 4.2, o *endpoint2* é consumidor do recurso;

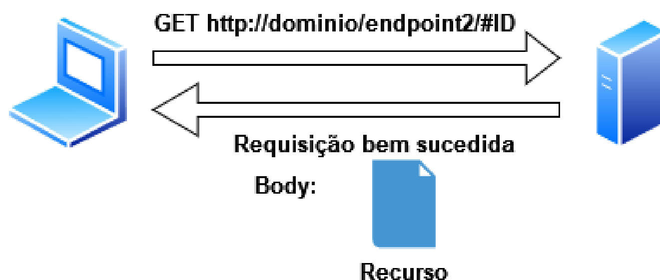

Figura 4.2: Exemplo de requisição HTTP para um *endpoint* consumidor de um recurso. Fonte: Elaborada pelo autor.

3. Uma requisição a um *endpoint* consumidor só poderá conter identificadores já remetidos ao cliente em requisições anteriores a *endpoints* produtores. Requisições que não atendem esse critério sinalizam uma tentativa de ataque à API via BOLA.

4.2 Método

A visão geral do método proposto para detecção dos ataques BOLA é ilustrada na Figura 4.3. Vale destacar que o método proposto é visto como uma solução para ser implementada na forma de um *proxy* ou um WAF (*Web Application Firewall*), filtrando o trafego HTTP entre cliente e servidor.

O método proposto utiliza como entrada as requisições HTTP do(s) cliente(s), que serão analisadas antes do processamento da requisição pelo servidor. Para detectar uma tentativa de ataque BOLA, é proposto o armazenamento dos identificadores de recursos enviados pelo servidor. Desta forma, futuras requisições que visem esses recursos serão validadas quanto ao critério de identificadores, ou seja, só podem ser aceitas requisições com identificadores que tenham sido recebidos pelo cliente em requisições anteriores.

Para isso, uma *allowlist* será utilizada para o controle do consumo de recursos, sendo válida somente dentro de uma determinada sessão. A *allowlist* será criada quando uma autenticação válida for criada para um cliente e será invalidada em conjunto com a finalização da sessão desse cliente.

No que tange o funcionamento, ao receber uma requisição HTTP, esta será primeiramente validada quanto ao seu formato por um validador (ou *data validator*), a fim de evitar o processamento de requisições não conformantes com o especificado para a API (na especificação OAS). Caso o formato da requisição não seja válido, o cliente será notificado através do código de erro 400 (*Bad request*).

Caso o formato da requisição seja válido, o *endpoint* alvo da requisição deverá ser classificado como produtor ou consumidor. Para isso, as informações necessárias serão extraídas da descrição dos *Links* na especificação OAS (presente nas versões 3.0 e superiores) da API, através do uso de um *parser*. A partir dessa classificação:

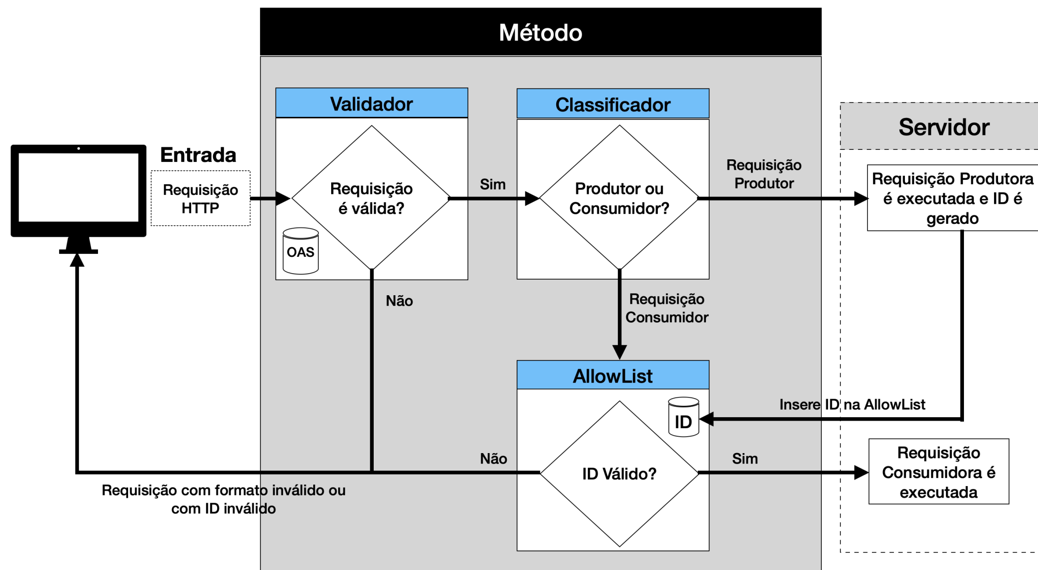

Figura 4.3: Método proposto para detecção de ataque BOLA. Fonte: Elaborada pelo autor.

- Caso o *endpoint* seja **produtor**, a requisição é processada pelo servidor. Caso seja bem sucedida, os identificadores retornados na resposta do servidor são inseridos na *allowlist* da sessão do cliente;
- Caso o *endpoint* seja **consumidor**, os identificadores inseridos na requisição serão comparados com os já presentes na *allowlist*. Caso os identificadores estejam presentes, a requisição é considerada válida e remetida ao servidor para processamento. Caso não, a requisição é considerada inválida (uma tentativa de ataque a vulnerabilidade BOLA), ficando a cargo da implementação estipular se e como esta será respondida. Como exemplos, pode ser retornado para o cliente o código HTTP 401 (*Unauthorized*), ou uma resposta mais complexa pode ser customizada através do uso de um *callback*, a depender da implementação.

Na prática, a adição desse um *proxy* possui impactos que não podem ser desprezados. Em termos de desempenho, a necessidade de filtrar operações, processar dados e acessar bases de dados (*Allowlist*) leva a um aumento no tempo de resposta das requisições, bem como a um aumento no consumo de memória. Do ponto de vista de segurança, o comprometimento do *proxy* por um atacante pode revelar dados de sessão dos clientes, de forma que a segurança deste também deve ser levada em consideração. Esses pontos, embora relevantes, não serão detalhados no escopo deste trabalho.

4.3 Implementação

O método proposto foi implementado para realização de experimentos de validação, utilizando-se componentes *open source* já disponíveis onde possível. Nessa seção, discutiremos os detalhes dessa implementação.

Os componentes inseridos no método proposto foram utilizados como base para construção de um *proxy* que intermedia o envio de requisições do cliente e das respostas dos servidores, com

a exceção do validador. Em caso de detecção positiva, o *proxy* pode manter o envio da requisição para o servidor ou bloquear a requisição sem necessariamente remetê-la ao servidor. Para o desenvolvimento deste *proxy*, foi utilizada a ferramenta *open source* Mitmproxy (Cortesi et al., 2010), que dá suporte a customização de extensões programáveis (*Addons*) utilizando a linguagem Python. O código-fonte utilizado na implementação do *proxy* encontra-se disponível em *<https://github.com/mcloebens/antiBOLA/tree/64a968b3b9b7e088f9540270ce0b0377a859abc4>*.

4.3.1 Validador

Embora para esta implementação o componente validador não tenha sido utilizado, ele poderia ter sido implementado utilizando diversas ferramentas já existentes. Dentre as opções, a ferramenta OpenAPI-Core (Maciag and contributors, 2021) seria a escolha que mais facilmente se adequaria a implementação interna do Mitmproxy (ambos implementados em Python). Outras ferramentas disponíveis para outras linguagens (por exemplo, Express-OpenAPI-Validator (DiMascio and contributors, 2021), em JavaScript) poderiam também ser utilizadas com o devido esforço de implementação.

Contudo, esses validadores se tornam muito restritivos para realização de testes para várias APIs, uma vez que as especificações dessas APIs precisam ser bem detalhadas ou o validador irá recusar a requisição. Nem todas as APIs testadas no escopo desse trabalho possuíam sua especificação OAS disponibilizada e nenhuma delas fornecia a especificação dos *Links* entre as operações, o que demandou a construção dessas especificações a partir de testes.

Assim, substituímos a implementação deste por uma lógica interna menos restritiva, onde verificamos apenas se a operação alvo da requisição (Método + *Path*) está descrita na API para fazermos o processo de detecção. Caso não esteja, a requisição é enviada diretamente para o servidor. Essa lógica assume que as requisições de interesse para validação do método são bem formadas com relação aos requisitos da API, ou seja, possuem todos os parâmetros necessários, o que é suficiente para realização dos experimentos no escopo desse trabalho.

4.3.2 Classificador

O componente Classificador foi implementado através da utilização de um *parser* para extrair a informação incluída na especificação OpenAPI do servidor de interesse da ferramenta. Para isso, foi utilizada a ferramenta Swagger Parser (Messinger and contributors, 2021), desenvolvida para a linguagem JavaScript. Essa ferramenta, além de interpretar a especificação como um objeto, permitindo a interação programática com esta, é capaz de dereferenciar referências inseridas, facilitando o acesso a pontos conectados logicamente dentro da especificação.

Um *script* foi desenvolvido para retornar a informação considerada pertinente da especificação da API em um objeto JSON, evitando a necessidade da interface de comunicação contínua entre um processo Javascript e o *proxy* (Python). Para os propósitos dessa implementação, foram consideradas relevantes as especificações que definem uma requisição no escopo da especificação OpenAPI (considerada uma operação): *Path*, métodos e seus parâmetros, respostas e seus parâmetros e, principalmente, *Links* para outras operações inseridos nas respostas.

Esse *script* (Figura 4.4) é executado uma única vez assim que o *proxy* é iniciado, e as informações retornadas por ele são utilizadas para definir quais operações possuem *Links* declarados entre si, de onde inferimos a informação de uma relação de sucessão entre uma operação e outra (produtor-consumidor). Assim que uma requisição é recebida, verificamos se esta se destina a um método consumidor e utilizamos a *Allowlist* para validar seus parâmetros. Para uma resposta recebida, podemos verificar se esta é resultado de uma requisição bem

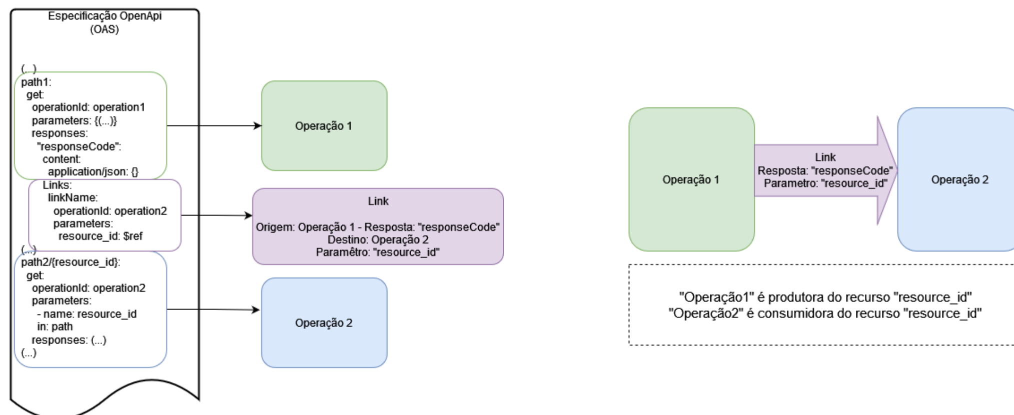

Figura 4.4: Extração (*parsing*) da informação relevante da especificação OpenAPI. Fonte: Elaborada pelo autor.

sucedida a um *endpoint* produtor e inserir os parâmetros retornados na *Allowlist* para validar requisições futuras. Esse fluxo está detalhado na Figura 4.5.

4.3.3 *Allowlist*

A *Allowlist* foi implementada como um dicionário que relaciona um *token* de identificação do cliente aos dados da sessão deste. Esta sessão é descrita como um grafo, onde os parâmetros retornados para um dado *Link* são associados a um vértice que liga as operações vinculadas (produtora e consumidora). Para facilitar essa implementação, foi utilizada a biblioteca *open source* NetworkX (Hagberg et al., 2008), ainda que não sejam necessárias operações robustas para navegação em grafos.

Quando uma requisição é feita para um método produtor, extraímos os valores dos parâmetros inseridos na resposta desta pelo servidor. Assim, são inseridos como dados do vértice apenas os valores referentes a um parâmetro de interesse (descrito para utilização em outras requisições no *Link*). Esses parâmetros podem estar inseridos em diversos pontos da requisição, como no caminho (*path*), consulta (*query*), cabeçalho (*header*) ou corpo (*body)*. Também podem ser inserido de diversas formas, como números, texto, parâmetros internos de um objeto (ou objeto contendo objetos, e assim sucessivamente). Dessa forma, é preciso processar corretamente a resposta, considerando a descrição desta no documento OAS.

Havendo execução de um método consumidor, podemos consultar se existe o vértice entre a respectiva operação e a operação que a deveria ter precedido. Caso exista, podemos recuperar os dados associados a esse vértice, que são, efetivamente, o histórico de parâmetros retornados para o cliente pelo *endpoint* produtor do recurso. De maneira mais rigorosa, seria possível também verificar se existe um histórico válido de requisições para este cliente (cliente visitou pelo menos um nó produtor para cada nó consumidor visitado), ao invés de verificar apenas a requisição imediatamente anterior. Esses dados são então comparados com os parâmetros na requisição, que de maneira similar a descrita para as respostas de requisições para métodos produtores, podem estar inseridos em diversos pontos e diversas formas, e devem ser extraídos de acordo com a especificação OAS da API.

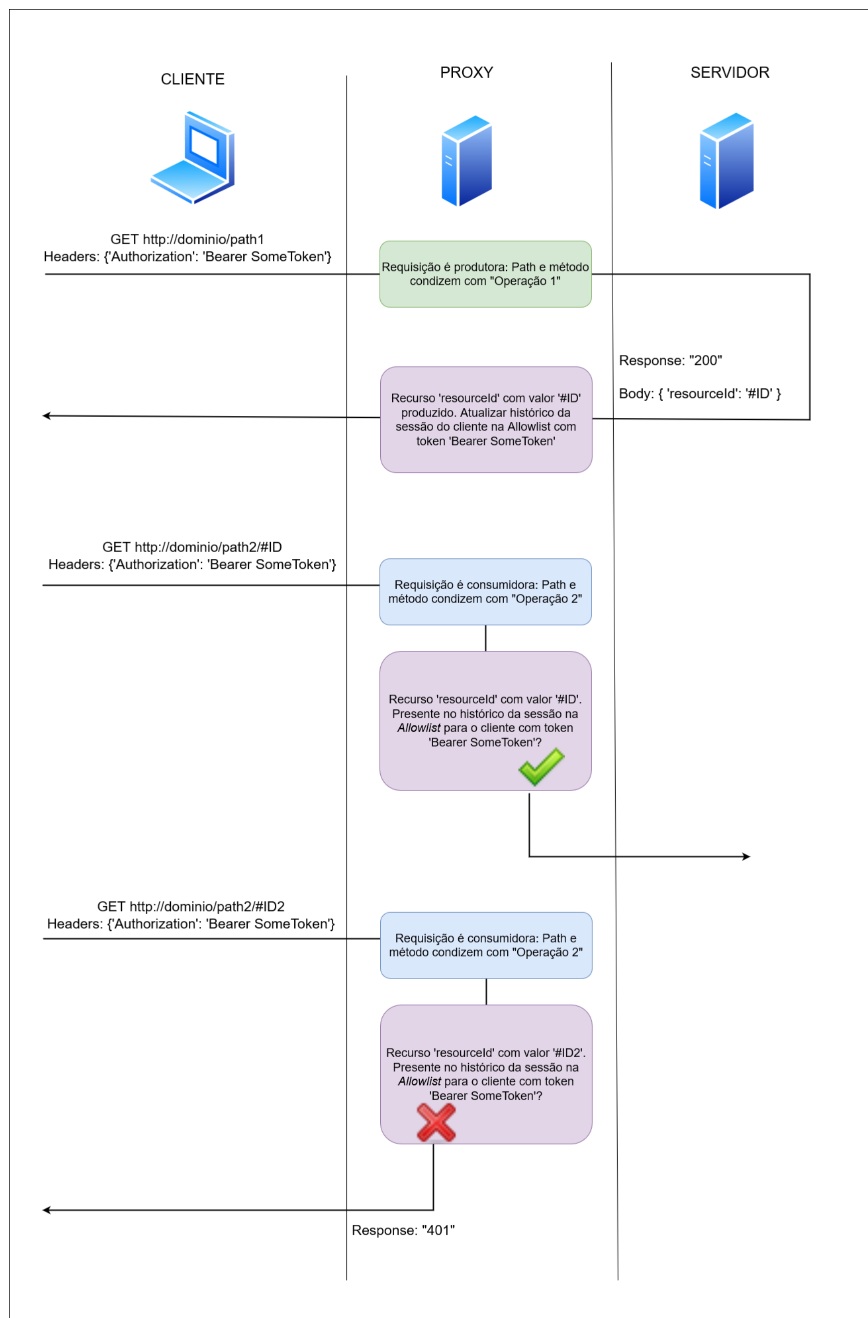

Figura 4.5: Análise de requisições pelo *proxy*, considerando extração da Figura 4.4. Fonte: Elaborada pelo autor.

4.4 Considerações sobre o Capítulo

Neste capítulo foi descrito o método proposto para detecção de um ataque BOLA. Ele é baseado na interceptação, análise e validação de requisições HTTP. Embora sua estrutura não seja uma novidade e já tenha sido utilizada em diferentes contextos e cenários, até onde se sabe esta é a primeira instanciação ou especialização para detecção de ataques BOLA, podendo ser considerada uma contribuição.

O método considera que *endpoints* podem ser classificados como produtores ou consumidores de um determinado recurso, baseando-se em uma especificação da API, em formato OAS 3.0 ou superior, contendo os *Links* entre os *endpoints*. Assim, uma requisição a um *endpoint* produtor só pode conter identificadores de recursos que já tenham sido remetidos ao cliente em respostas a requisições anteriores a *endpoints* produtores.

A aplicação desta regra mitiga precisamente o cenário de um ataque à API via BOLA, no qual o cliente inclui na requisição de um recurso um identificador que este não recebeu do servidor, que sugere portanto conhecimento externo ao fluxo de comunicação (que pode ter sido resultado da exploração bem sucedida de outra vulnerabilidade, ou simplesmente que este "chutou" o identificador, baseando-se no formato de um identificador legítimo (identificadores fora do formato esperado podem ser invalidados diretamente pelo *data validator*).

Dessa forma, o método efetivamente irá inserir, do lado do servidor, uma representação de estado da comunicação entre este e o cliente, não implementada por este atendendo a diretiva de design REST. Contudo, como esse controle de estado é independente do processo de desenvolvimento, ele não prejudica os benefícios de redução de custo de implementação do *design*. De toda forma, uma vez que existem controles de autenticação implementados, o conceito de estado já está presente na comunicação.

Essa abordagem possibilita a detecção do ataque em tempo de execução, diferenciando-se das abordagens atualmente empregadas que tem foco na robustez do controle de segurança, em tempo de desenvolvimento, evitando que falhas nessa fase levem a um ataque bem sucedido.

A princípio, o principal requisito para que a solução tenha capacidade de detecção satisfatória é que a descrição OAS da API seja precisa na descrição das relações entre os *endpoints* através dos *Links*, o que não tem sido frequente no mundo real (Kotstein and Decker, 2020). Contudo, esse tipo de descrição tem custo ínfimo de desenvolvimento se comparada ao volume de testes necessários para garantir a implementação de controles de segurança robustos para cada recurso. Futuramente, é possível que essa descrição de relações possa ser obtida através da análise de registros de comunicação legítima com técnicas de *machine learning*, o que viria a complementar essa solução.

Já sobre a implementação, devido ao grande número de possibilidades para implementação de APIs somadas a descrição da especificação destas no formato OpenAPI, algumas considerações foram feitas para possibilitar a implementação de um protótipo funcional. Algumas delas possuem relação com as características de implementação da API não cobertas pela especificação OpenAPI, outras foram aplicadas para simplificar a implementação do protótipo. Discutiremos algumas delas e as razões envolvidas.

Devido a necessidade de identificação do cliente, foram utilizados parâmetros inseridos nas requisições e respostas contendo a sequencia de caracteres "token" ou "auth" (sem diferenciar maiúsculas de minúsculas) para diferenciação entre clientes. Esse formato é funcional para propósitos de desenvolvimento, porém em uma aplicação para uso geral deve ser substituído pela implementação da autenticação da API alvo.

Foi considerado também que a implementação da API é tal que os parâmetros retornados por um *endpoint* produtor não são alterados pelo cliente para envio a um *endpoint* consumidor, ou seja, se um parâmetro é inserido como parte de um *Link* após determinada resposta, o

valor inserido na requisição sucessora é o mesmo recebido na requisição ao *endpoint* produtor. Utilizando somente a especificação, não é possível ter informações a respeito de lógicas internas de manipulação dos parâmetros retornados. Além disso, a implementação atual utiliza somente parâmetros inseridos em objetos JSON incluídos no corpo da resposta, cabeçalho (*headers*) ou no campo *query* da URL.

Capítulo 5

Resultados

Esse capítulo expõe as condições de experimentação e os resultados obtidos com a implementação do método proposto para detecção de ataques BOLA, utilizando projetos de aplicações disponíveis online para fins experimentais e educativos.

5.1 Condições de Experimentação

Ainda que existam vários casos reais noticiados de ataques BOLA em APIs, como os ataques envolvendo o Facebook, Uber e outros elencados na sessão 1.1, as vulnerabilidades expostas são corrigidas imediatamente, evitando maiores prejuízos relacionados. Isso torna o uso de vulnerabilidades reais para experimentação difícil, uma vez que haveria a necessidade de encontrar, através de testes, uma API real cuja vulnerabilidade fosse desconhecida até esse momento.

Outra opção seria a utilização de um histórico de requisições e respostas em um *replay* da execução destas. Contudo, não foram localizadas bases de testes que incluíssem todo o conteúdo de um histórico de requisições e respostas para uma aplicação com API documentada. Dessa forma, esse trabalho utilizou, para validação do método proposto, aplicações desenvolvidas para testes de segurança e aprendizado relacionados a segurança de APIs.

Essas aplicações expõem, de maneira deliberada, vulnerabilidades através de suas APIs, imitando cenários do mundo real de forma que seus usuários possam testar diferentes formas de ataque para explorar essas vulnerabilidades e aprender com isso sobre os detalhes envolvidos. Algumas delas possuem roteiros e desafios para que o usuário encontre e explore as vulnerabilidades presentes. No contexto dessa dissertação, utilizamos as vulnerabilidades BOLA encontradas nas APIs de quatro destas aplicações.

As vulnerabilidades BOLA conhecidas foram exploradas em testes e, após isso, os *Links* entre os *endpoints* produtores e consumidores foram descritos nas especificações OpenAPI das aplicações. Para as aplicações que não forneceram a especificação, esta foi criada com parâmetros básicos e apenas os *Paths* e métodos para os quais os *Links* foram descritos, permitindo que a ferramenta os analisasse.

As aplicações *proxy* e cliente foram instanciadas em diferentes portas na mesma máquina. Sendo assim, todas as requisições são feitas no domínio "*localhost*". As aplicações foram instanciadas utilizando *containers* da ferramenta Docker (algumas implementam mais de um serviço, utilizando assim mais de uma porta). Para o envio de requisições, foi utilizada a ferramenta *Postman*, configurada para enviar todas as suas requisições através do *proxy* implementado

(Mitmproxy) com o método de detecção. Esse ambiente foi instanciado em uma máquina virtual Linux com as seguintes configurações:

- Sistema Operacional: Ubuntu 20.04.2 LTS (Focal) 64-bit;
- Memória RAM: 8 Gb;
- CPU: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, 6 núcleos.

5.2 Aplicações e Vulnerabilidades

5.2.1 crAPI

A crAPI (*completely ridiculous API*) (Piyush et al., 2021) é uma aplicação desenvolvida pela OWASP para demonstração e exercícios referentes a vulnerabilidades listadas em seu Top 10 de vulnerabilidades em APIs (OWASP, 2019) e possui dois desafios referentes a vulnerabilidade BOLA. A aplicação vulnerável é um modelo de serviço de facilitação entre usuários e profissionais (Figura 5.1), que coloca em contato mecânicos e donos de carros que precisam de seus serviços. Nesta, o usuário pode fazer uma conta associada a um veículo e solicitar a realização de serviços por um mecânico. O usuário pode também comprar peças e outros itens de vendedores e interagir com outros usuários em um fórum. A API possui alguns *endpoints* vulneráveis a ataques BOLA, listados a seguir.

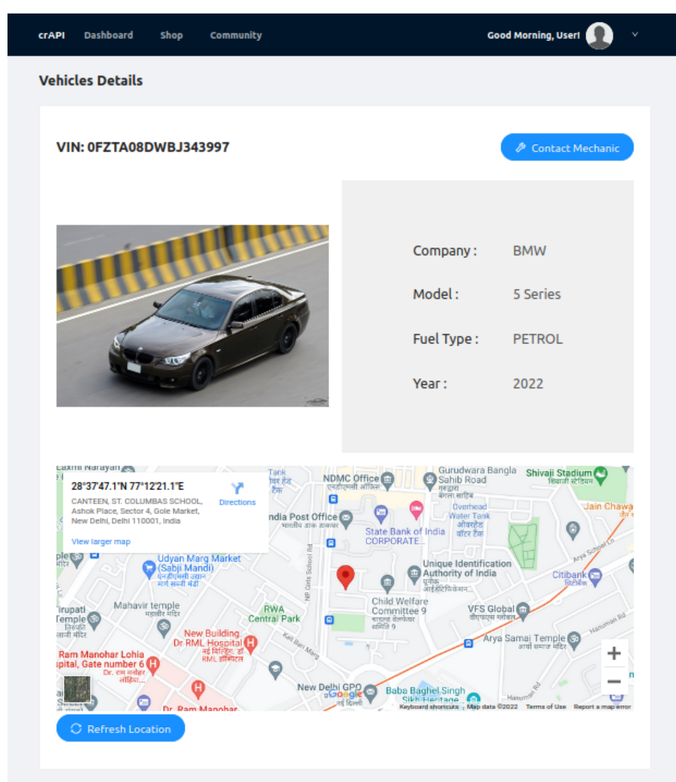

Figura 5.1: crAPI - Interface de navegação. Fonte: Capturada pelo autor.

GET /identity/api/v2/vehicle/{car_id}/location

Esse *endpoint* é consultado ao se clicar no botão para atualizar a localização do veículo (botão *Refresh Location* na Figura 5.1). Retorna, entre outros dados, a informação de localização de qualquer veículo na base de dados, desde que o identificador *car_id* seja fornecido corretamente e o usuário que fez a requisição possua um *token* válido de autenticação.

O *car_id* é um Identificador Único Universal (UUID) e pode ser obtido para veículos de outros usuários em outro *endpoint* (*GET /community/api/v2/community/posts/recent*), parte da funcionalidade de fórum da ferramenta, que expõe muita informação (outra das vulnerabilidades mais comuns em APIs web presentes no Top 10 OWASP de 2019) sobre os usuários. As Figuras 5.2 e 5.3 ilustram a exploração da vulnerabilidade.

Na Figura 5.2, podemos ver a resposta a uma requisição feita para este *endpoint*. Ela expõe, de maneira demasiada, dados dos usuários e identificadores que podem ser usados para tentativas de engenharia reversa do funcionamento da aplicação. Entre eles, um UUID que identifica o veículo. Esse identificador pode ser utilizado como o parâmetro *car_id* para exploração da vulnerabilidade BOLA, expondo outros dados sensíveis (localização do veículo), o que pode ser visto na 5.3.

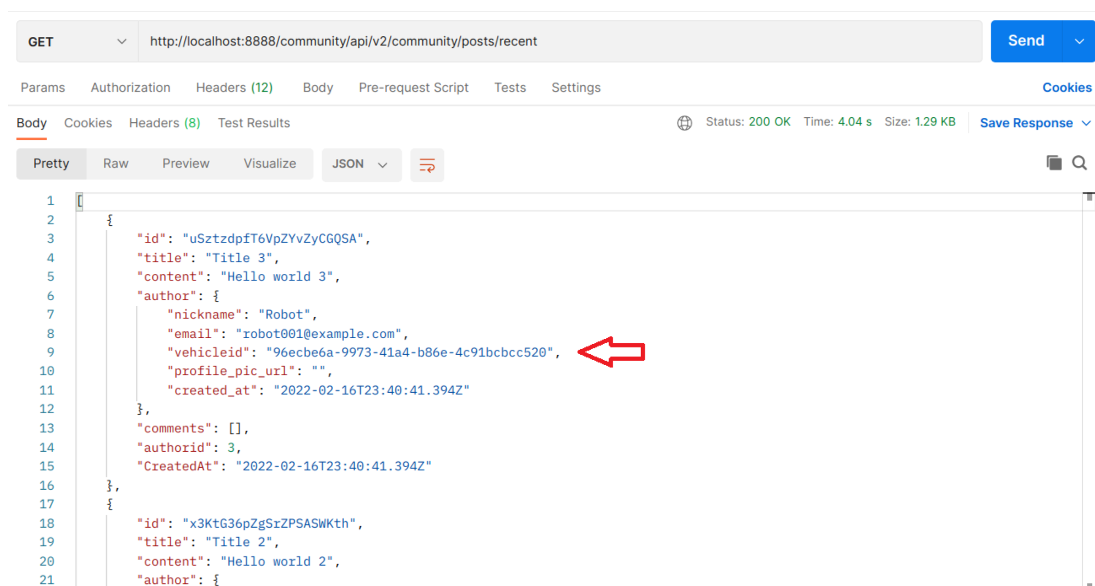

Figura 5.2: crAPI - Resultado da requisição ao *endpoint* '*GET community/api/v2/community/posts/recent*'. Fonte: O autor.

GET /workshop/api/mechanic/mechanic_report

Este *endpoint* retorna o relatório do mecânico sobre o serviço solicitado no *endpoint POST /workshop/api/merchant/contact_mechanic*. A requisição anterior retorna em seu corpo o parâmetro *id* que deve ser inserido na *query* da requisição como parâmetro *report_id* (Figura 5.4).

Esse identificador pode ser enumerado e possui problemas nos controles de autenticação, retornando qualquer relatório existente na base de dados (por exemplo o referenciado pelo valor *report_id* igual a 1, como visto na Figura 5.5). Esse relatório traz informações sensíveis de outro usuário da plataforma.

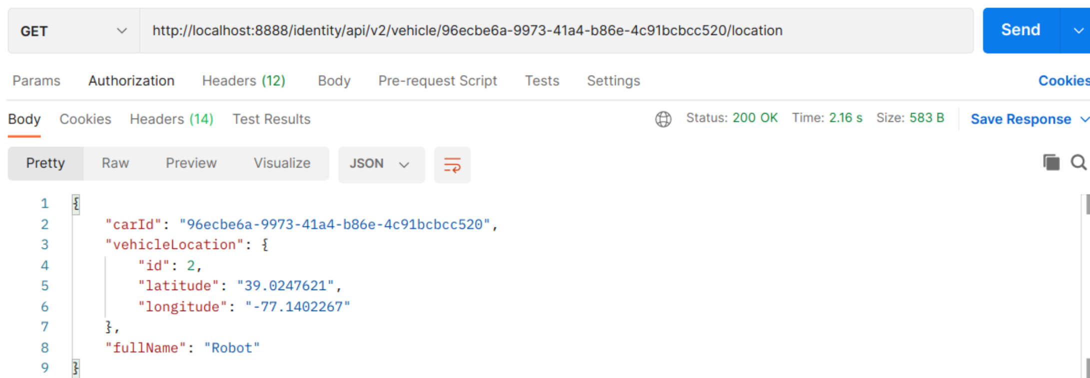

Figura 5.3: crAPI - Exploração do *endpoint* vulnerável '*GET /identity/api/v2/vehicle/{car_id}/location*'. Fonte: O autor.

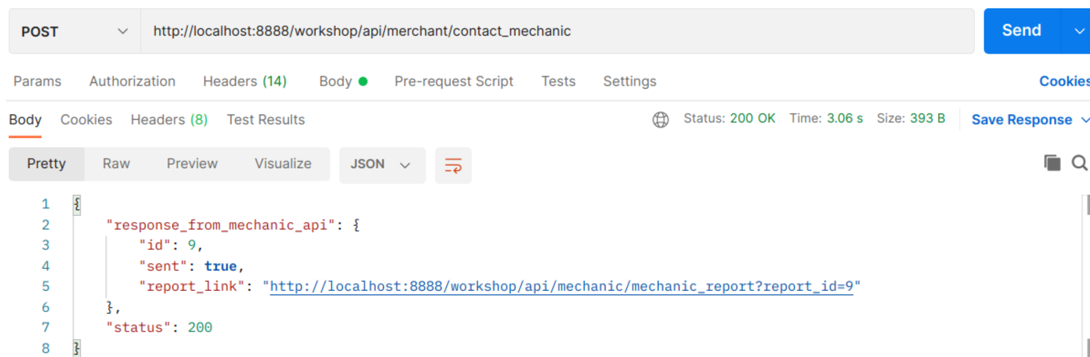

Figura 5.4: crAPI - Resultado da requisição ao *endpoint* '*POST /workshop/api/merchant/contact_mechanic*'. Fonte: O autor.

5.2.2 vAPI

A vAPI (*vulnerable Adversely Programmed Interface*) (Kulkarni et al., 2021) é uma aplicação vulnerável com exercícios relacionados ao Top 10 OWASP de vulnerabilidades em APIs (OWASP, 2019). A aplicação possui interface navegável apenas para acesso à sua documentação e é focada em vulnerabilidades no processo de autenticação, onde suas operações permitem a criação e autenticação de usuários e recuperação e atualização de suas informações. Para cada vulnerabilidade descrita no Top 10 OWASP, a API descreve um conjunto de operações vulneráveis para serem testadas pelos usuários. A API possui os seguintes *endpoints* vulneráveis:

GET /vapi/api1/user/{api1_id}

Este *endpoint* é consultado para se obter as informações do usuário. Qualquer usuário com um *token* válido obtido pode executar o método, que retorna os dados do usuário desde que o parâmetro *api1_id* exista na base. Além do problema de autenticação, os identificadores são enumeráveis, o que expõe mais informação sobre o funcionamento da aplicação no servidor. A Figura 5.6 mostra a obtenção dos dados de outro usuário, referenciados pelo parâmetro *api1_id* com valor '1'.

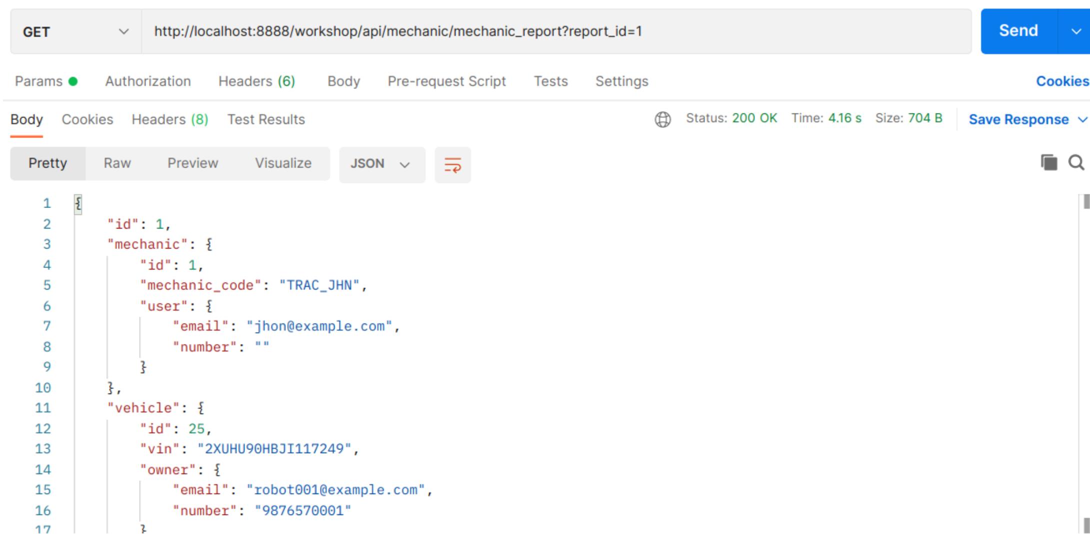

Figura 5.5: crAPI - Exploração do *endpoint* vulnerável '*GET /workshop/api/mechanic/mechanic_report*'. Fonte: O autor.

PUT /vapi/api1/user/{api1_id}

Mesmo *path* vulnerável (*/vapi/api1/user/*), porém com outra operação (agora PUT ao invés de GET). Neste *endpoint* é possível modificar os dados de qualquer usuário *api1_id*, desde que o usuário que executa o método possua um *token* válido. A Figura 5.7 mostra a alteração de dados de usuário referenciados pelo parâmetro *api1_id* com valor '1', como *username* e *name* que posteriormente são retornados utilizando o método GET, demonstrando que foram realmente alterados para esse usuário.

POST http://localhost/vapi/api1/user Send

Params Authorization Headers (11) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
{
    "username": "marcelo125",
    "name": "marcelo",
    "course": "123",
    "password": "12345"
}
```

Body Cookies (3) Headers (7) Test Results Status: 201 Created Time: 1620 ms Size: 289 B Save Response

Pretty Raw Preview Visualize JSON

```
{
    "username": "marcelo125",
    "name": "marcelo",
    "course": "123",
    "id": 8
}
```

GET http://localhost/vapi/api1/user/1 Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Body Cookies (3) Headers (7) Test Results Status: 200 OK Time: 2.03 s Size: 316 B Save Response

Pretty Raw Preview Visualize JSON

```
{
    "id": 1,
    "username": "michaels",
    "name": "Michael Scott",
    "course": "flag{api1_d0cd9be2324cc237235b}"
}
```

Figura 5.6: vAPI - Exploração do *endpoint* vulnerável '*GET /vapi/api1/user/{api1_id}*'. Fonte: O autor.

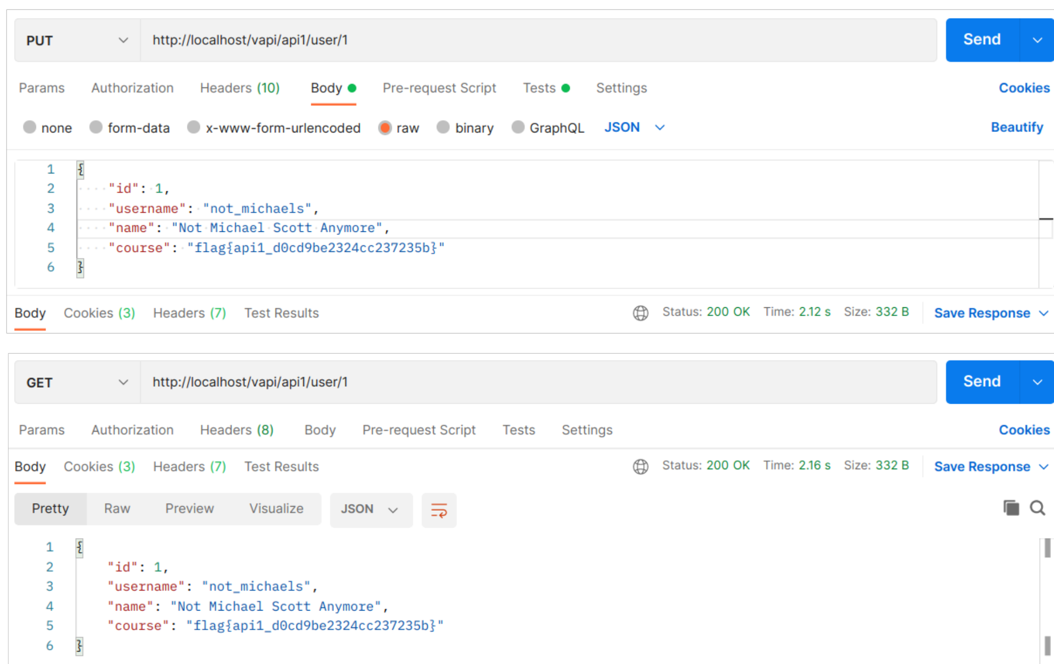

Figura 5.7: vAPI - Exploração do *endpoint* vulnerável '*PUT /vapi/api1/user/{api1_id}*'. Fonte: O autor.

5.2.3 PIXI

A PIXI *photo sharing* API (Becher and contributors, 2020) é uma aplicação vulnerável desenvolvida para exercícios e desafios relacionados às vulnerabilidades do Top 10 OWASP de vulnerabilidades em aplicações web (OWASP, 2017). A aplicação é um modelo de ferramenta de publicação de imagens (Figura 5.8), onde os criadores recebem moedas digitais de acordo com *feedbacks* positivos (ou "loves") dos usuários da aplicação. Embora a aplicação foque nas vulnerabilidades para aplicações web e não para APIs, esta possui alguns *endpoints* vulneráveis a BOLA que podem ser utilizados para os propósitos desse trabalho.

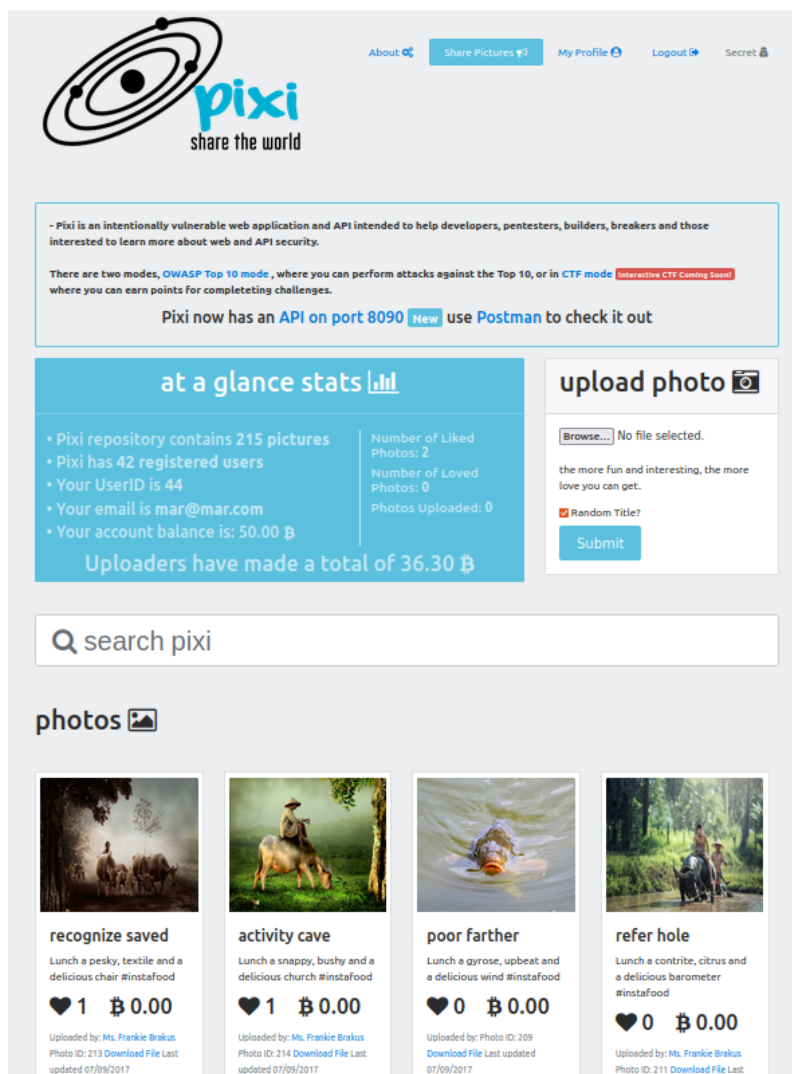

Figura 5.8: PIXI - Interface de navegação. Fonte: Capturada pelo autor.

Para poder executar requisições na API, é necessário primeiro obter os *cookies* de sessão retornados pelo acesso a interface de navegação em um *browser*. A API possui os seguintes *endpoints* vulneráveis:

GET /api/user/info

Este *endpoint* retorna as informações de usuário. Para acessá-lo, é necessário o envio de um *token* transparente (JWT) obtido na realização do login. Contudo, esse *token* pode ser editado, permitindo a um atacante obter as informações de qualquer usuário ao enviar o *token* modificado. Isso é feito obtendo-se o segredo utilizado para assinatura do *token*, que aparece *hardcoded* em um arquivo de configuração exposto pela aplicação (Figuras 5.9 e 5.10).

```
TypeError: "server.conf" is not a function
    at /usr/src/pixi/web/server.js:1309:15
    at Layer.handle [as handle_request] (/usr/src/pixi/web/node_modules/express/lib/router/layer.js:95:5)
    at next (/usr/src/pixi/web/node_modules/express/lib/router/route.js:137:13)
    at Route.dispatch (/usr/src/pixi/web/node_modules/express/lib/router/route.js:112:3)
    at Layer.handle [as handle_request] (/usr/src/pixi/web/node_modules/express/lib/router/layer.js:95:5)
    at /usr/src/pixi/web/node_modules/express/lib/router/index.js:281:22
    at Function.process_params (/usr/src/pixi/web/node_modules/express/lib/router/index.js:335:12)
    at next (/usr/src/pixi/web/node_modules/express/lib/router/index.js:275:10)
    at /usr/src/pixi/web/node_modules/express-unless/index.js:44:14
    at Layer.handle [as handle_request] (/usr/src/pixi/web/node_modules/express/lib/router/layer.js:95:5)
```

Figura 5.9: PIXI - Erro no servidor revela nome de arquivo de configuração. Fonte: Capturado pelo Autor.

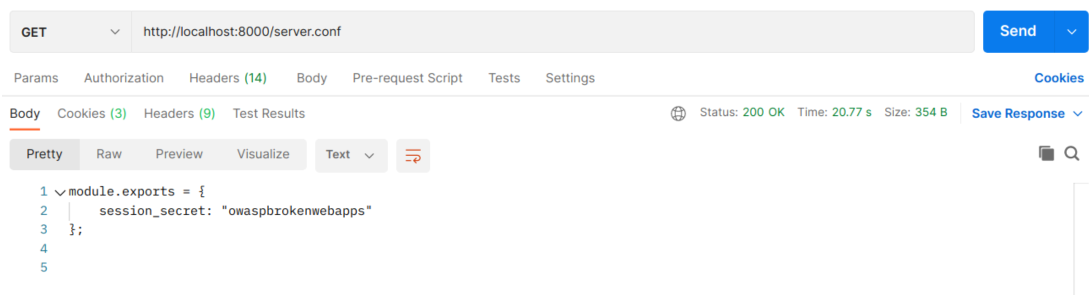

Figura 5.10: PIXI - Arquivo de configuração contendo segredo é exposto ao método http GET. Fonte: O autor.

Após realizado o login utilizando o *endpoint POST /api/login*, o token JWT recebido pode ser modificado usando o segredo encontrado. O processo[1] é ilustrado nas Figuras 5.11 e 5.12, mantido pela companhia Auth0, focada em serviços de autenticação. Após a obtenção do token modificado, a vulnerabilidade BOLA pode ser explorada, como pode ser visto na Figura 5.13.

Usualmente, autenticações que requerem o uso de *tokens* do tipo JWT possuem métodos de atualização do *token* após um determinado intervalo de tempo. Contudo, a aplicação não realiza essas atualizações, permitindo que tratemos, no escopo desse trabalho, o *token* como um parâmetro.

GET /api/picture/{picture_id}

O *endpoint* retorna as informações de uma imagem específica. O parâmetro *picture_id* pode ser enumerado, permitindo que se obtenha qualquer imagem da base. Da mesma maneira, os

[1] Foi utilizada, para manipulação do JWT, uma ferramenta online presente no site *JSON Web Tokens* <jwt.io>, acesso em 18/02/2022.

Encoded PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjp7Il9pZCI6NDQsImVtYWlsIjoibWFyQG1hci5jb20iLCJwYXNzd29yZCI6Im1hcjEyMzQ1NiIsIm5hbWUiOiJ3cmFwcGVkZnJvZyIsInBpYyI6Imh0dHBzOi8vczMuYW1hem9uYXdzLmNvbS91aWZhY2VzL2ZhY2VzL3R3aXR0ZXIvcmRiYW5ub24vMTI4LmpwZyIsImlzX2FkbWluIjpmYWxzZSwiYWNjb3VudF9iYWxhbmNlIjo0OS45MDAwMDAwMDAwMDAwMDAwMDYsImFsbF9waWN0dXJlcyI6W119LCJpYXQiOjE2NDUzNjY1MzR9.0QlY0ZU5yjtILoSfG2rltYwkNmXovCvFXe8ZrzNKm4A

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

{
  "alg": "HS256",
  "typ": "JWT"
}

PAYLOAD: DATA

{
  "user": {
    "_id": 44,
    "email": "mar@mar.com",
    "password": "mar123456",
    "name": "wrappedfrog",
    "pic": "https://s3.amazonaws.com/uifaces/faces
/twitter/rdbannon/128.jpg",
    "is_admin": false,
    "account_balance": 49.900000000000006,
    "all_pictures": []
  },
  "iat": 1645366534
}

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Figura 5.11: PIXI - JWT expõe diversos dados de usuário e informação que pode ser usada para engenharia reversa do servidor. Fonte: Capturado pelo autor.

endpoints GET /api/picture/{picture_id}/loves e *GET /api/picture/{picture_id}/likes* também retornam informações de qualquer imagem da base (*loves* e *likes*, respectivamente). No caso desses *endpoints*, é possível que a intenção fosse expor os dados de todas as imagens da base, uma vez que na interface navegável o usuário pode ver as imagens de outros usuários, ou seja, os *endpoints* não são necessariamente vulneráveis a BOLA, mas no escopo desse trabalho serão avaliados como sendo para fins de análise da metodologia proposta. O *endpoint GET /api/pictures* retorna todos os identificadores *picture_id* na base. A exploração desse *endpoint* é similar as ilustradas para as aplicações anteriores.

DELETE api/picture/delete

O *endpoint* deleta qualquer imagem especificada pelo parâmetro *picture_id* desde que o usuário tenha um *token* JWT válido. Esse parâmetro pode ser enumerado, o que permite que qualquer imagem na base seja deletada. A exploração desse *endpoint* é similar as ilustradas para as aplicações anteriores.

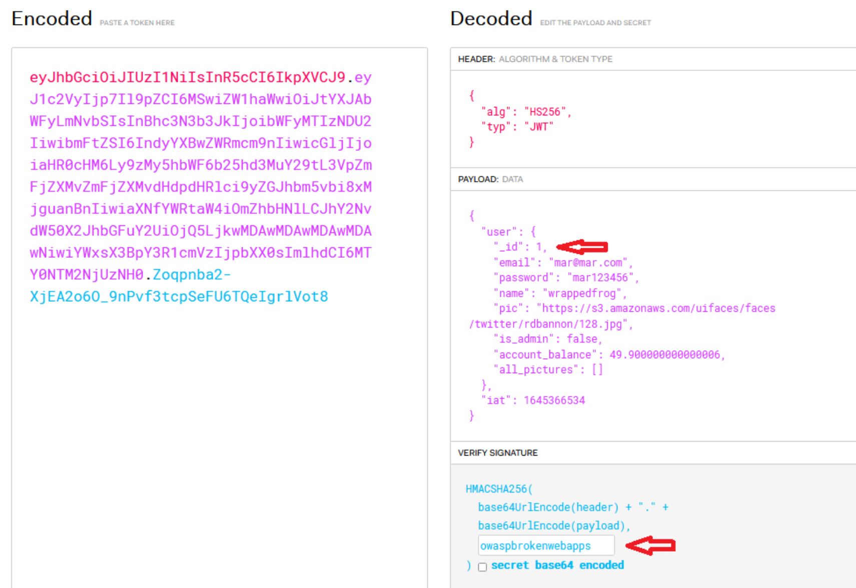

Figura 5.12: PIXI - JWT pode ser modificado de forma válida utilizando o segredo. Fonte: Capturado pelo autor.

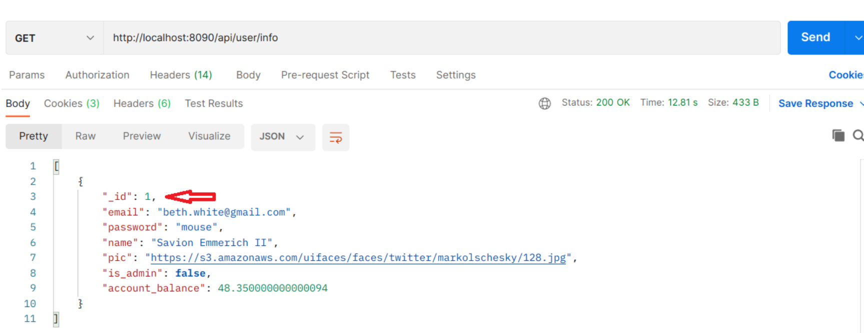

Figura 5.13: PIXI - *Endpoint GET /api/user/info* retorna informação a respeito de qualquer usuário, desde que o JWT incluído no *header* ‘x-access-token’ seja válido. Fonte: O autor.

5.2.4 VAmPI

A VAmPI (*vulnerable API*) (Anonymous, 2021) é uma aplicação vulnerável com vulnerabilidades do Top 10 OWASP de vulnerabilidades em APIs (OWASP, 2019) inseridas para teste de ferramentas de detecção de vulnerabilidades. A aplicação é um modelo de ferramenta de cadastro de livros, onde os usuários se autenticam e inserem seus livros e um segredo, dados que posteriormente pode ser recuperado pelo usuário. Essa aplicação não possui interface navegável. A API possui o seguinte *endpoint* vulnerável:

GET /books/v1/{book}

Este *endpoint* é consultado para se obter o segredo de um livro a partir de seu nome. Qualquer usuário com um *token* válido pode executar a operação, obtendo assim o segredo de qualquer livro desde que esse possua o identificador *book*. Esse identificador é retornado quando o usuário insere seu livro e seu segredo na base de dados, e pode ser obtido em outro ponto da API, no método *GET /books/v1*, que lista todos os livros da aplicação.

5.3 Resultados

5.3.1 crAPI

Para detecção de tentativas de ataque aos *endpoints* vulneráveis, foram inseridos os seguintes *Links* na especificação OpenAPI (Tabela 5.1 e Figuras 5.14 e 5.15):

Tabela 5.1: crAPI - *Links* Inseridos na Especificação OAS

Origem (Método Path)	Resposta	Parametro	Destino (Método Path)
GET /identity/api/v2/vehicle/vehicles	200	car_id	GET /identity/api/v2/vehicle/{car_id}/location
POST /workshop/api/merchant/contact_mechanic	200	report_id	GET /workshop/api/mechanic/mechanic_report

```
/identity/api/v2/vehicle/vehicles:
  get:
    operationId: getVehicleInfo
    summmary: Retrieve vehicle info
    parameters:
    - name: Content-Type⋯
    - name: Accept⋯
    responses:
      "200":
        description: Report received
        content:
          application/json: {}
        links:
          getmechanicalreport:
            operationId: getVehicleLocation
            parameters:
              car_id: $response.body#/uuid
      "400":⋯
      "401":⋯
```

```
/identity/api/v2/vehicle/{car_id}/location:
  get:
    operationId: getVehicleLocation
    summary: Get vehicle location
    parameters:
    - name: Content-Type⋯
    - name: Accept⋯
    - name: car_id
      in: path
      required: true
      schema:
        type: string
    responses:
      "200":
        description: Ok
        content:
          application/json: {}
      "400":⋯
      "401":⋯
```

Figura 5.14: crAPI - Link inserido na especificação OAS entre operações que produzem e consomem o identificador ‘car_id’. Fonte: O autor.

Figura 5.15: crAPI - Link inserido na especificação OAS entre operações que produzem e consomem o identificador ‘report_id’. Fonte: O autor.

Os *Links* nas Figuras 5.14 e 5.15 sinalizam uma relação entre os parâmetros (campo *parameters*) incluídos na resposta das requisições listadas na coluna ‘Origem’ da 5.1, e as requisições onde esses parâmetros podem ser usados (campo *operationId*), listadas na coluna operações ‘Destino’. No contexto desse trabalho, as operações ‘Origem’ são vistas como produtoras dos identificadores incluídos no campo *parameters*, enquanto as operações ‘Destino’ são vistas como consumidoras destes recursos.

A partir da inserção dos *Links*, o método sugerido foi capaz de identificar tentativas de exploração nas vulnerabilidades, como pode ser visto nas Figuras 5.16 e 5.17. As requisições foram então bloqueadas antes de chegar ao servidor, sendo respondidas pelo *proxy* com ‘401 - *Unauthorized*’.

Figura 5.16: crAPI - Detecção de tentativa de exploração da vulnerabilidade pela implementação da metodologia sugerida. Fonte: O autor.

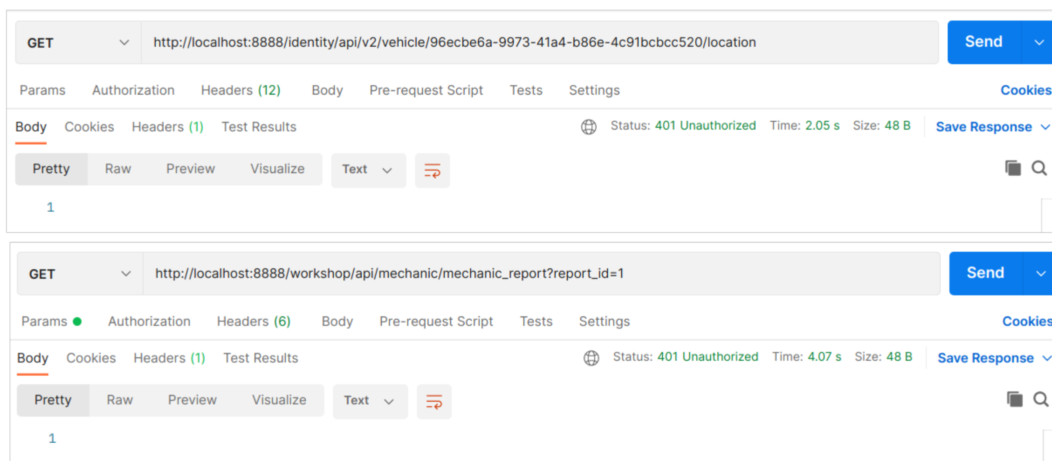

Figura 5.17: crAPI - Requisições onde tentativas de ataque BOLA foram detectadas pela metodologia sugerida são respondidas pelo *proxy* com '401 - *Unauthorized*' sem chegar ao servidor. Fonte: O autor.

5.3.2 vAPI

Para detecção de tentativas de ataque aos *endpoints* vulneráveis, foram inseridos os seguintes *Links* na especificação OpenAPI (Tabela 5.2 e Figura 5.18):

Tabela 5.2: vAPI - *Links* Inseridos na Especificação OAS

Origem (Método Path)	**Resposta**	**Parametro**	**Destino (Método Path)**
POST /vapi/api1/user	201	api1_id	GET /vapi/api1/user/api1_id
POST /vapi/api1/user	201	api1_id	PUT /vapi/api1/user/api1_id

```
/vapi/api1/user:
  post:
    operationId: createuser
    summary: Create User
    parameters:
    - name: Content-Type…
    - name: Accept…
    requestBody:…
    responses:
      "201":
        description: User created
        content:
          application/json: {}
        links:
          getuserinfo:
            operationId: getUserInfo
            parameters:
              api1_id: $response.body#/id
          updateuserinfo:
            operationId: updateUserInfo
            parameters:
              api1_id: $response.body#/id
```

```
/vapi/api1/user/{api1_id}:
  get:
    operationId: getUserInfo
    summary: Get User
    parameters:
    - name: Authorization-Token…
    - name: Content-Type…
    - name: api1_id
      in: path
      required: true
      style: simple
      explode: false
      schema:
        type: string
    responses:
      "200":…
  put:
    operationId: updateUserInfo
    summary: Update User
    parameters:
    - name: Authorization-Token…
    - name: Content-Type…
    - name: api1_id
      in: path
      required: true
      style: simple
      explode: false
      schema:
        type: string
    requestBody:…
    responses:
      "200":…
```

Figura 5.18: vAPI - Link inserido na especificação OAS entre operações que produzem e consomem o identificador ‘api1_id’. Fonte: O autor.

Os *Links* na Figura 5.18 sinalizam uma relação entre os parâmetros (campo *parameters*) incluídos na resposta das requisições listadas na coluna ‘Origem’ da 5.2, e requisições onde esses parâmetros podem ser usados (campo *operationId*), listadas na coluna operações ‘Destino’. Como já dito anteriormente, estes representam relações de produtor-consumidor de um recurso utilizadas pela metodologia sugerida para detecção das tentativas de exploração da

vulnerabilidade BOLA. Nesse caso, o mesmo parâmetro retornado pelo *endpoint* produtor pode ser usado em dois *endpoints* consumidores diferentes.

A partir da inserção dos *Links*, a metodologia sugerida foi capaz de identificar tentativas de exploração nas vulnerabilidades, como pode ser visto nas Figuras 5.19 e 5.20. As requisições foram então bloqueadas antes de chegar ao servidor, sendo respondidas pelo *proxy* com '401 - *Unauthorized*'

```
==================================================
REQUEST
GET /vapi/api1/user/1

- LINK FOUND - CONSUMER
- ORIGIN OP: createuser - RESPONSE: 201
- TARGET OP: getUserInfo
- NO DATA FOUND FOR THE LINK IN SESSION RECORDS

BOLA EXPLOIT ATTEMPT DETECTED - REQUEST BLOCKED

==================================================
RESPONSE
401 - Unauthorized
```

```
==================================================
REQUEST
PUT /vapi/api1/user/1

- LINK FOUND - CONSUMER
- ORIGIN OP: createuser - RESPONSE: 201
- TARGET OP: updateUserInfo
- NO DATA FOUND FOR THE LINK IN SESSION RECORDS

BOLA EXPLOIT ATTEMPT DETECTED - REQUEST BLOCKED

==================================================
RESPONSE
401 - Unauthorized
```

Figura 5.19: vAPI - Detecção de tentativa de exploração da vulnerabilidade pela implementação da metodologia sugerida. Fonte: O autor.

GET http://localhost/vapi/api1/user/1 Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies
Body Cookies (3) Headers (1) Test Results Status: 401 Unauthorized Time: 2.06 s Size: 48 B Save Response

PUT http://localhost/vapi/api1/user/1 Send
Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
{
    "id": 1,
    "username": "not_not_michaels",
    "name": "Not Not Michael Scott Anymore Anymore",
    "course": "flag{api1_d0cd9be2324cc237235b}"
}
```

Body Cookies (3) Headers (1) Test Results Status: 401 Unauthorized Time: 1980 ms Size: 48 B Save Response

Figura 5.20: vAPI - Requisições onde tentativas de ataque BOLA foram detectadas pela metodologia sugerida são respondidas pelo *proxy* com '401 - *Unauthorized*' sem chegar ao servidor. Fonte: O autor.

5.3.3 PIXI

Para detecção de tentativas de ataque aos *endpoints* vulneráveis, foram inseridos os seguintes *Links* na especificação OpenAPI (Tabela 5.3):

Tabela 5.3: PIXI - *Links* Inseridos na Especificação OAS

Origem (Método Path)	**Resposta**	**Parametro**	**Destino (Método Path)**
POST /api/login	200	x-access-token	GET /api/user/info
GET /api/user/pictures	200	picture_id	GET /api/picture/{picture_id}
GET /api/user/pictures	200	picture_id	GET /api/picture/{picture_id}/loves
GET /api/user/pictures	200	picture_id	GET /api/picture/{picture_id}/likes
GET /api/user/pictures	200	picture_id	DELETE api/picture/delete

Ainda que o parâmetro 'x-access-token' seja um *token* de autenticação, o Link que define as operações produtoras é consumidoras deste pode ser inserido, neste caso, da mesma forma que o de um parâmetro identificador comum, devido a implementação da metodologia sugerida adotar *cookies* de sessão como possíveis *tokens* de identificação do cliente. A partir da inserção desse e demais *Links*, a metodologia sugerida foi capaz de identificar tentativas de exploração das vulnerabilidades, de maneira similar as APIs descritas anteriormente.

Contudo, para o caso do parâmetro 'picture_id', caso sejam inseridos *Links* entre as operações consumidoras e o *endpoint GET /api/pictures* (que retorna informações de todas as figuras da base), a metodologia sugerida não é capaz de evitar a exploração da vulnerabilidade. Isso ocorre porque agora existe registro de envio, pelo servidor ao cliente, de todos os valores do parâmetro 'picture_id' que identificam recursos existentes, ou seja, todos os valores estão presentes na *Allowlist*. O comportamento do *proxy* é visto na Figura 5.21.

```
==================================================
REQUEST
GET /api/pictures


[::1]:34378: server connect localhost:8090 (127.0.0.1:8090)
==================================================
RESPONSE
200 - OK

LINK FOUND - PRODUCTOR
- ORIGIN OP: getAllPictures - RESPONSE: 200
- TARGET OP: useruploadedpictures
- DATA RECORDED: {'picture_id': [214, 209, 211, 208, 212, 207
, 206, 210, 205, 201, 204, 200, 202, 203, 185, 191, 196, 175,
 180, 197, 193, 178, 181, 189, 190, 195, 176, 182, 186, 187,
194, 199, 177, 183, 192, 198, 179, 184, 188, 171, 170, 174, 1
72, 173, 164, 165, 163, 169, 167, 161, 166, 162, 160, 168, 15
6, 152, 159, 151, 150, 157, 154, 158, 153, 155, 137, 144, 146
, 129, 132, 149, 143, 133, 139, 127, 147, 142, 136, 130, 126,
 148, 138, 141, 125, 131, 145, 140, 135, 134, 128, 121, 110,
119, 124, 109, 108, 113, 118, 122, 114, 115, 107, 123, 112, 1
17, 106, 120, 111, 116, 105, 104, 99, 91, 103, 90, 98, 95, 10
0, 101, 94, 93, 97, 96, 102, 92, 82, 87, 83, 86, 79, 85, 89,
80, 88, 81, 84, 77, 75, 76, 78, 74, 72, 71, 73, 69, 70, 66, 5
8, 60, 67, 57, 63, 68, 61, 56, 55, 62, 65, 64, 59, 54, 45, 52
, 40, 51, 44, 39, 53, 41, 47, 48, 49, 50, 42, 43, 46, 37, 38,
 20, 24, 26, 29, 35, 27, 31, 33, 22, 28, 32, 36, 23, 25, 30,
34, 21, 17, 12, 4, 6, 19, 9, 14, 3, 16, 11, 7, 13, 18, 2, 8,
1, 5, 10, 15, 0]}
```

```
==================================================
REQUEST
GET /api/picture/1

- LINK FOUND - CONSUMER
- ORIGIN OP: getAllPictures - RESPONSE: 200
- TARGET OP: useruploadedpictures
SEARCHING PARAMETER picture_id VALUE IN SESSION RECORDS
- PARAMETER picture_id : 1 FOUND IN SESSION RECORDS
- LINK FOUND - CONSUMER
- ORIGIN OP: getUserPictures - RESPONSE: 200
- TARGET OP: useruploadedpictures
- NO DATA FOUND FOR THE LINK IN SESSION RECORDS

[::1]:34378: server connect localhost:8090 (127.0.0.1:8090)
==================================================
RESPONSE
200 - OK
```

Figura 5.21: PIXI - Todos os valores do parâmetro 'picture_id' são retornados pelo endpoint *GET /api/pictures*. Em uma requisição futura a um *endpoint* consumidor do parâmetro com qualquer dos valores retornados, o *proxy* verifica que o valor se encontra na *Allowlist* e não bloqueia a requisição. Fonte: O autor.

5.3.4 VAmPI

Para detecção de tentativas de ataque ao *endpoint* vulnerável, foram inseridos os seguintes *Link*s na especificação OpenAPI (Tabela 5.4):

Tabela 5.4: VAmPI - *Link* Inseridos na Especificação OAS

Origem (Método Path)	**Resposta**	**Parametro**	**Destino (Método Path)**
POST /books/v1	200	book	GET /books/v1/{book}

A partir da inserção do *Link*, a metodologia sugerida foi capaz de identificar tentativas de exploração no *endpoint* vulnerável, de maneira similar as APIs descritas anteriormente. Contudo, de maneira similar ao parâmetro 'picture_id' na aplicação PIXI, se o *Link* entre a operação *GET /books/v1* com resposta '200' contendo o parâmetro 'book' e a operação *GET /books/v1/{book}* for adicionado, o método não é capaz de evitar o acesso a qualquer um dos livros, uma vez que todos os identificadores são retornados e inseridos na *Allowlist*.

5.4 Discussão

Considerando os cenários simulados pelas aplicações testadas, a implementação da metodologia mostrou-se capaz de detectar e evitar tentativas de exploração a vulnerabilidades BOLA conhecidas nessas aplicações através de suas APIs em formato caixa-preta em tempo real, ignorando os detalhes de implementação dos servidores e atendo-se somente as requisições e respostas para os casos onde os identificadores são desconhecidos pelo cliente.

Isso é possível na medida que o comportamento do servidor é corretamente modelado em sua especificação OpenAPI, contendo principalmente a informação de quais parâmetros são retornados pelo servidor após uma requisição e onde esses parâmetro podem ser utilizados (*Links*). Dessa forma, a comunicação pode ser vista como contendo estados, sacrificando-se em parte a simplicidade entre a diretiva de comunicação *stateless* entre cliente e servidor do padrão REST em troca da segurança dos dados no servidor, similarmente ao feito para implementação de controles de autenticação.

Contudo, como foi descrito para as APIs PIXI e VAmPI, se os parâmetros forem retornados por um *endpoint* presente nos *Links*, apenas os controles de autenticação são capazes de evitar o acesso a este. A metodologia sugerida não tem acesso a nenhuma informação sobre a autenticação a não ser o *token* que identifica a sessão do cliente, e dessa forma, não consegue evitar ataques BOLA onde a sequência temporal entre as requisições, conforme a especificação dos *Links*, está correta.

Dessa forma, vê-se que a descrição da API tem papel crucial no funcionamento desta solução, podendo levar a um comportamento indefinido e aparecimento de erros que não aconteceriam sem a existência do *proxy*, em casos onde a especificação OpenAPI não modelar corretamente o funcionamento do servidor, ou em casos onde a implementação do método apresentar falhas em interpretar o documento OAS.

Outro caso diferenciado observados nos testes ocorre também na aplicação PIXI, relativo às operações produtoras e consumidoras do *token* JWT inserido no parâmetro 'x-access-token'. Embora a implementação da metodologia sugerida pudesse tratar o *token* de autenticação como um parâmetro comum, não é provável que isso se reflita em aplicações reais devido à alguns fatores. O primeiro é que existe a necessidade de identificar corretamente o cliente, e se esta identificação é feita por *tokens* de autenticação, deve haver um *token* retornado previamente para identificar outro *token* (para esse caso, foram considerados os *cookies*). Outro motivo

é que a implementação usual de JWTs para autenticação exige que o *token* seja atualizado (em intervalos de tempo curtos, de preferência), ou seja, não é comum que JWTs tenham valores estáticos. Além disso, conceitualmente, *tokens* de autenticação não são identificadores de recursos, e portanto não estão cobertos pelo escopo desta solução.

O cenário de experimentos dessa pesquisa foi limitado a aplicações de testes, e, dessa forma, tem sua validade para o mundo real limitada aos cenários simulados por estas. Além disso, para cada API, foi necessário pelo menos descrever as informações dos *Links* em sua especificação OAS, que não foram fornecidas a priori pelos desenvolvedores dessas APIs. Para algumas delas, foi necessário escrever toda a especificação. Como as regras de negócio influenciam na forma que a API é descrita, estando ausentes outra formas de documentação, numa API vulnerável não é possível dizer com certeza que um *endpoint* retorna determinada informação por uma falha, ou se essa é a real intenção deste, ou se um *endpoint* aceita requisições contendo qualquer identificador por problemas nos controles de autenticação ou se esse é o comportamento esperado. Dessa forma, a ferramenta somente protegeu de maneira efetiva os *endpoints* para os quais esses *Links* estavam escritos, havendo a possibilidade de outros pontos de vulnerabilidade estarem presentes e passarem indetectados.

Para resultados com maior confiabilidade na detecção de vulnerabilidades desconhecidas, idealmente, a ferramenta deveria partir de descrições completas das APIs e ser instanciada em um cenário de uso real, ou com *replay* de um histórico de requisições completo, uma vez que testes automatizados também levam em conta requisições a pontos conhecidos.

Em comparação com a literatura encontrada, nosso método se diferencia por ser capaz de bloquear uma tentativa de ataque em tempo real, o que apenas o uso de controles robustos de autenticação se propõe a fazer. Além disso, o uso de controles de autenticação não pode ser evitado por nosso método, uma vez que ela depende do controle de sessões por clientes através de *tokens* de autenticação. Ele também não é capaz de bloquear ataques em casos onde houve de roubo de credenciais ou sequestro de sessão.

De fato, é porque existem controles de autenticação que a navegação entre os *endpoints* da API pode ser aprofundada pela solução de testes de *fuzzing* desenvolvida por Atlidakis et al. (2019), onde foi proposta a ideia de classificar *endpoints* entre produtores e consumidores no contexto de APIs web, utilizada como base nesse trabalho. O método detalhado nesse trabalho deve ser visto, portanto, como uma camada extra de segurança inserida sobre os controles de autenticação, evitando a exposição de dados em caso de falha pontual nos controles de autenticação sobre um objeto. Nosso método verifica se a sequência das requisições é legítima, não tendo informações se o usuário está autorizado a executar essas requisições.

Adicionar essa camada de segurança, conforme citado anteriormente, possui custo que sacrifica em parte benefícios do design REST. Nosso método adiciona, naturalmente, um *overhead* relacionado ao custo em espaço de armazenar históricos de envio de parâmetros por sessão do lado do servidor para cada requisição a um *endpoint* produtor, e um *overhead* relacionado ao custo no tempo de consultar essa base de informação a cada requisição destinada a um *endpoint* consumidor. Esse aumento de custo não pode ser negligenciado para implementações do método no mundo real.

Com tudo isso em consideração, os experimentos realizados no contexto desse trabalho demonstram que essa é uma alternativa promissora, sobretudo se considerarmos os avanços possíveis na descrição de APIs de maneira automatizada. Dessa forma, o custo de implementação do método se torna ínfimo, e, caso o custo para utilização desta em servidores de produção se mostre muito elevado, esta ainda pode ser aplicada em cenários de testes da API (como *replay* de requisições reais), podendo levar à detecção de ataques a posteriori e à identificação e correção de vulnerabilidades existentes.

Capítulo 6

Conclusão

Esta dissertação apresentou uma proposta de metodologia de detecção de ataques *Broken Object Level Authorization* (BOLA) a APIs desenvolvidas seguindo o padrão REST, que ocorrem quando um atacante obtém acesso a recursos de forma ilegítima devido a um problema nos controles de autenticação deste recurso. O método, que pode ser implementado para filtrar a comunicação em tempo real entre o cliente e o servidor, necessita como insumo da descrição da API no formato de especificação OpenAPI, incluindo a especificação dos *Links*, que relacionam uma resposta e os demais *endpoints* onde seu conteúdo pode ser utilizado pelo cliente.

Essa informação é suficiente para que se determine relações do tipo produtor-consumidor entre as operações realizadas na API, ou seja, dado um parâmetro que identifica um recurso, qual *endpoint* retorna essa informação ao cliente após uma requisição bem sucedida (produtor) e qual necessita que essa seja inserida na requisição para que esta seja bem sucedida (consumidor). De posse dessas relações, a detecção de uma tentativa de ataque pode ser feita na medida em que o cliente deve ter recebido um parâmetro identificador de um recurso do servidor antes de requisitar esse recurso ao servidor inserindo seu parâmetro identificador, ou seja, o parâmetro deve ter sido retornado após uma requisição a um *endpoint* produtor. Assim, armazenando-se um histórico dos identificadores retornados na sessão é possível verificar, para cada requisição a um *endpoint* consumidor, se o parâmetro inserido foi obtido de maneira legítima. Assumindo-se que a descrição da API modela as regras de negócio de maneira correta, caso não exista histórico de envio o cliente não deveria possuir a informação, sinalizando uma tentativa de ataque BOLA.

O método foi implementado, para validação, como um *proxy*, inicialmente interpretando uma especificação OpenAPI e após isso realizando as operações necessárias para o armazenamento dos parâmetros e validações das requisições. Como servidores, foram utilizadas quatro aplicações desenvolvidas para testes de vulnerabilidades em APIs contendo vulnerabilidades por *design*. De maneira geral, observando os cenários de ataque BOLA, a descrição dos *Links* foi suficiente para que o *proxy* recusasse as requisições e evitasse os ataques, desde que não houvesse exposição desses identificadores em um *Link* descrito. A exploração da vulnerabilidade só é possível para os casos em que o cliente obteve a informação dentro do fluxo de requisições esperado pela API. Nos casos em que o cliente enumera um identificador ou obtém esse em outro ponto da API não considerado parte do fluxo de comunicação esperado, a requisição que exploraria a vulnerabilidade é recusada.

6.1 Dificuldades Encontradas

Para utilização do método, é necessário que a descrição completa da especificação OpenAPI para a API que se deseja monitorar, incluindo os *Links* das respostas com as próximas requisições. Mesmo nos casos em que a descrição OpenAPI já existe, a descrição desses *Links* não é amplamente utilizada (o que já foi observado por Kotstein and Decker (2020)), de forma que para testar a metodologia, é necessário escrever a especificação completamente ou em parte, o que exige conhecimento detalhado das regras de negócio.

Além disso, a obtenção de bases reais para testes é difícil na medida em que a metodologia necessita das requisições e respostas completas, além da descrição da API. Testes utilizando aplicações reais necessitam que estas possuam vulnerabilidades conhecidas que não foram endereçadas, o que não é viável.

Como outras dificuldades, em menor nível, pode-se citar a necessidade de conhecimento técnico aplicado para implementação do método, que vai além do entendimento dos conceitos envolvidos. Para utilização de ferramentas já disponíveis para simplificar a implementação, é necessário se adaptar as linguagens de programação e demais tecnologias envolvidas.

6.2 Trabalhos Futuros

Considerando que a principal dificuldade para generalização do método consiste em obter especificações detalhadas da API, diversos avanços podem ser feitos na geração automática dessas especificações utilizando técnicas de aprendizagem de máquina, a partir de análise do código fonte ou de requisições e respostas utilizadas em um fluxo legítimo. Devem ser consideradas, contudo, as reais intenções dos fornecedores da API quanto a utilização desta, o que pode se mostrar difícil de generalizar utilizando somente dados.

Técnicas de aprendizagem de máquina também poderiam ser utilizadas para melhorar o funcionamento da arquitetura proposta, seja melhorando o processo de classificação ou de uso da *Allowlist*. É possível, inclusive, que a aplicação de técnicas de aprendizagem de máquina seja capaz de remodelar completamente a arquitetura, classificando diretamente requisições anômalas com base num modelo treinado, sem o armazenamento de um histórico das sessões dos usuários.

Com relação a autenticação, o processo de invalidação dos *tokens* e de sincronização dos detalhes de autenticação entre o servidor e o *proxy* não foi abordado neste trabalho. Melhorias no entendimento da autenticação no servidor poderiam ser usadas para melhorar o desempenho do *proxy*, bem como poderiam ser utilizadas para estender a detecção para outros problemas de segurança em APIs. Por exemplo, é possível que a BFLA (*Broken Function Level Authorization*), vulnerabilidade número 2 no Top 10 OWASP de 2019 (similar ao BOLA, mas envolve uma falha no controle de escopos de autorização entre usuários diferentes) possa ser detectada utilizando uma metodologia similar caso esta possua informação suficiente sobre o processo de autenticação no servidor para diferenciar as permissões de cada usuário.

Outro ponto não incluído no escopo dessa dissertação foi avaliar os impactos da inserção de um *proxy* para implementação do método de detecção no sistema como um todo. Esses impactos, a princípio, podem ser de desempenho ou de segurança. Com relação ao desempenho, a implementação pode levar a aumento no tempo de latência das requisições, aumento da carga de processamento no servidor, aumento no uso de memória (*Allowlist* armazena dados por requisição). Os impactos no desempenho e sua escalabilidade devem ser avaliados para garantir a viabilidade técnica em uma implementação do método. Para a segurança do sistema, sendo o *proxy* agora outro módulo exposto para o cliente e este contendo dados das sessões de todos

os clientes, vulnerabilidades na implementação deste podem causar a exposição do servidor a ataques que não ocorreriam caso o *proxy* não estivesse incluído no sistema.

Além disso, ainda com relação ao desempenho do sistema, uma das melhorias possíveis tem relação com a arquitetura distribuída dos sistemas web. A implementação do método considera que toda a informação retornada pela aplicação passa por um único nó da rede que existe entre o cliente e o servidor. Contudo, em aplicações reais, os dados provêm de diferentes servidores, de forma que a metodologia pode ser otimizada utilizando técnicas de sincronização em sistemas distribuídos, evitando a centralização do fluxo de requisições através de um único nó na rede.

Referências Bibliográficas

Akamai (2019). State of the internet / security vol. 5 issue 2: Retail attacks and api traffic. `https://www.akamai.com/content/dam/site/it/documents/state-of-the-internet/state-of-the-internet-security-retail-attacks-and-api-traffic-report-2019.pdf`. Acesso em 01/10/2021.

Anonymous (2021). Vampi. `https://github.com/erev0s/VAmPI`. [Version 0.0.1].

Atlidakis, V., Godefroid, P., and Polishchuk, M. (2019). Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE.

Atlidakis, V., Godefroid, P., and Polishchuk, M. (2020). Checking security properties of cloud service rest apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 387–397. IEEE.

Becher, N. and contributors (2020). Pixi photo sharing api. `https://github.com/DevSlop/Pixi`. [Version 1.0.0].

Bisht, P., Hinrichs, T., Skrupsky, N., and Venkatakrishnan, V. (2014). Automated detection of parameter tampering opportunities and vulnerabilities in web applications. *Journal of Computer Security*, 22(3):415–465.

Cortesi, A., Hils, M., Kriechbaumer, T., and contributors (2010). mitmproxy: A free and open source interactive HTTPS proxy. `https://mitmproxy.org/`. [Version 7.0].

DiMascio, C. and contributors (2021). An openapi validator for expressjs that automatically validates api requests and responses using an openapi 3 specification. `https://github.com/cdimascio/express-openapi-validator`. [Version 4.13.5].

Feng, X., Shen, J., and Fan, Y. (2009). Rest: An alternative to rpc for web services architecture. In *2009 First International Conference on Future Information Networks*, pages 7–10. IEEE.

Fielding, R. T. (2000). Rest: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California.*

Filho, A. S. and Feitosa, E. L. (2019). Detecção de api scrapers através do fluxo de hyperlinks. Anais do XIX Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais.

Fungy, A., Wangz, T., Cheungy, K., and Wongy, T. (2014). Scanning of real-world web applications for parameter tampering vulnerabilities. pages 341–352.

Hagberg, A. A., Schult, D. A., and Swart, P. J. (2008). Exploring network structure, dynamics, and function using networkx. In Varoquaux, G., Vaught, T., and Millman, J., editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA.

Ivanchikj, A. (2016). Restful conversation with restalk. In *International Conference on Web Engineering*, pages 583–587. Springer.

Ivanchikj, A., Gjorgjiev, I., and Pautasso, C. (2018a). Restalk miner: Mining restful conversations, pattern discovery and matching. In *International Conference on Service-Oriented Computing*, pages 470–475. Springer.

Ivanchikj, A., Pautasso, C., and Schreier, S. (2018b). Visual modeling of restful conversations with restalk. *Software & Systems Modeling*, 17(3):1031–1051.

Jin, B., Sahni, S., and Shevat, A. (2018). *Designing Web APIs: Building APIs That Developers Love*. O'Reilly Media, Inc.

Kitchenham, B. (2004). Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26.

Kotstein, S. and Decker, C. (2020). Navigational support for non hateoas-compliant web-based apis. In *Symposium and Summer School on Service-Oriented Computing*, pages 169–188. Springer.

Kulkarni, T., Deshpande, C., Silva, P. A., and Balu, M. (2021). vapi - vulnerable adversely programmed interface. `https://github.com/roottusk/vapi`. [Version 1.0.1].

KumarShrestha, A., Singh Maharjan, P., and Paudel, S. (2015). Identification and illustration of insecure direct object references and their countermeasures. In *IJCA*, volume 114, pages 39–44.

Li, X. and Xue, Y. (2011). Block: A black-box approach for detection of state violation attacks towards web applications. pages 247–256.

Maciag, A. and contributors (2021). Client-side and server-side support for the openapi specification v3. `https://github.com/p1c2u/openapi-core`. [Version 0.14.2].

Macy, J. (2018). How to build a secure api gateway. *Network Security*, 2018(6):12–14.

Mendoza, A. and Gu, G. (2018). Mobile application web api reconnaissance: Web-to-mobile inconsistencies vulnerabilities. volume 2018-May, pages 756–769.

Messinger, J. and contributors (2021). Swagger 2.0 and openapi 3.0 parser/validator. `https://apitools.dev/swagger-parser/`. [Version 10.0.3].

Monshizadeh, M., Naldurg, P., and Venkatakrishnan, V. (2014). Mace: Detecting privilege escalation vulnerabilities in web applications. pages 690–701.

OWASP (2013). Top 10-2013 the ten most critical web application security risks. `https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf`. Acesso em 01/10/2021.

OWASP (2017). Top 10-2017 the ten most critical web application security risks. `https://raw.githubusercontent.com/OWASP/Top10/master/2017/OWASP%20Top%2010-2017%20(en).pdf`. Acesso em 01/10/2021.

OWASP (2019). Top 10-2019 the ten most critical api security risks. `https://raw.githubusercontent.com/OWASP/API-Security/master/2019/en/dist/owasp-api-security-top-10.pdf`. Acesso em 01/10/2021.

OWASP (2021). Top 10-2021 the ten most critical web application security risks. `https://owasp.org/www-project-top-ten/`. Acesso em 01/10/2021.

Petrillo, F., Merle, P., Moha, N., and Guéhéneuc, Y.-G. (2016). Are rest apis for cloud computing well-designed? an exploratory study. In *International Conference on Service-Oriented Computing*, pages 157–170. Springer.

Piyush, R., Silva, P. A., and Lowe, J. D. (2021). completely ridiculous api (crapi). `https://github.com/OWASP/crAPI`. [Version 1.0.0].

Prokhorenko, V., Choo, K.-K., and Ashman, H. (2016). Web application protection techniques: A taxonomy. *Journal of Network and Computer Applications*, 60:95–112.

Richardson, L. and Ruby, S. (2008). *RESTful web services*. O'Reilly Media, Inc.

Sureda Riera, T., Bermejo Higuera, J.-R., Bermejo Higuera, J., Martínez Herraiz, J.-J., and Sicilia Montalvo, J.-A. (2020). Prevention and fighting against web attacks through anomaly detection technology. a systematic review. *Sustainability*, 12(12):4945.

Viriya, A. and Muliono, Y. (2021). Peeking and testing broken object level authorization vulnerability onto e-commerce and e-banking mobile applications. volume 179, pages 962–965.

Yalon, E. and Shkedy, I. (2019). API Security Project OWASP Projects' Showcase. Technical report, Global AppSec Amsterdam, Amsterdam.