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

Ailton da Silva dos Santos Filho

Uma Abordagem para a Detecção de Ataques a Vulnerabilidades Lógicas de Adulteração de Parâmetros e Controles de Acesso em APIs Web REST utilizando Redes de Petri Coloridas

Manaus
Agosto de 2025

Ailton da Silva dos Santos Filho

Uma Abordagem para a Detecção de Ataques a Vulnerabilidades Lógicas de Adulteração de Parâmetros e Controles de Acesso em APIs Web REST utilizando Redes de Petri Coloridas

Tese apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas como requisito parcial para obtenção do grau de Doutor em Informática.

Orientador: Prof. Eduardo Luzeiro Feitosa

Manaus
Agosto de 2025

Ficha Catalográfica

Elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

S237a Santos Filho, Ailton da Silva dos
Uma abordagem para a detecção de ataques a vulnerabilidades lógicas de adulteração de parâmetros e controles de acesso em apis web rest utilizando redes de petri coloridas / Ailton da Silva dos Santos Filho. - 2025.
98 f. : il., color. ; 31 cm.

Orientador(a): Eduardo Luzeiro Feitosa.
Tese (doutorado) - Universidade Federal do Amazonas, Programa de Pós-Graduação em Informática, Manaus, 2025.

1. API Web. 2. Vulnerabilidade Lógica. 3. detecção de ataques. 4. OpenAPI. 5. OWASP. I. Feitosa, Eduardo Luzeiro. II. Universidade Federal do Amazonas. Programa de Pós-Graduação em Informática. III. Título

---

Ministério da Educação
Universidade Federal do Amazonas
Coordenação do Programa de Pós-Graduação em Informática

FOLHA DE APROVAÇÃO

"UMA ABORDAGEM PARA A DETECÇÃO DE ATAQUES A VULNERABILIDADES LÓGICAS DE ADULTERAÇÃO DE PARÂMETROS E CONTROLES DE ACESSO EM APIS WEB REST UTILIZANDO REDES DE PETRI COLORIDAS"

AILTON DA SILVA DOS SANTOS FILHO

Tese de Doutorado defendida e aprovada pela banca examinadora constituída pelos professores:

Prof. Dr. Eduardo Luzeiro Feitosa - **Presidente**

Prof. Dr. Raimundo da Silva Barreto - **Membro Interno**

Prof. Dr. Diego Luiz Kreutz - **Membro Externo**

Prof. Dr. José Luiz de Souza Pio - **Membro Externo**

Prof. Dr. Gilbert Breves Martins - **Membro Externo**

Manaus, 27 de agosto de 2025.

Documento assinado eletronicamente por **Eduardo Luzeiro Feitosa**, **Professor do Magistério Superior**, em 27/08/2025, às 11:07, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015 .

Documento assinado eletronicamente por **Raimundo da Silva Barreto**, **Professor do Magistério Superior**, em 27/08/2025, às 11:21, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015 .

Documento assinado eletronicamente por **Jose Luiz de Souza Pio**, **Professor do Magistério Superior**, em 27/08/2025, às 15:44, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015 .

Documento assinado eletronicamente por **Gilbert Breves Martins**, **Usuário Externo**, em 04/09/2025, às 08:55, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Diego Luis Kreutz**, **Usuário Externo**, em 10/09/2025, às 17:46, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://sei.ufam.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2754212** e o código CRC **5D732102**.

Avenida General Rodrigo Octávio, 6200 - Bairro Coroado I Campus Universitário Senador Arthur Virgílio Filho, Setor Norte - Telefone: (92) 3305-1181 / Ramal 1193
CEP 69080-900, Manaus/AM, coordenadorppgi@icomp.ufam.edu.br

Referência: Processo nº 23105.037107/2025-05 SEI nº 2754212

Resumo

As APIs Web REST são amplamente utilizadas para integração entre sistemas, mas seu crescimento trouxe desafios de segurança, como vulnerabilidades lógicas, que são difíceis de detectar com métodos automatizados tradicionais. Entre essas falhas, destaca-se a Broken Object Level Authorization (BOLA), que permite que usuários não autorizados acessem ou modifiquem dados restritos. Esta tese propõe uma abordagem semi-automatizada baseada em Redes de Petri Coloridas para detectar ataques a Vulnerabilidades Lógicas de Adulteração de Parâmetros e Controles de Acesso em APIs REST, transformando especificações OpenAPI em modelos formais e analisando logs do servidor para identificar divergências entre o comportamento esperado e o observado.

Para validar a abordagem, foi desenvolvida a ferramenta `Links2CPN`, que realiza automaticamente a transformação do modelo e a detecção de potenciais ataques. Em três cenários de aplicações vulneráveis, a solução demonstrou mais de 95% de acurácia e precisão na identificação de tentativas de ataques BOLA. Os resultados mostram que a combinação de modelagem formal e análise de logs é capaz de detectar vulnerabilidades lógicas, oferecendo uma alternativa viável para melhorar a segurança de APIs REST.

Palavras-chave: API Web, Vulnerabilidade Lógica, detecção de ataques, OpenAPI, OWASP.

Abstract

REST Web APIs are widely used for system integration, but their growth has introduced security challenges, such as logical vulnerabilities, which are difficult to detect with traditional automated methods. Among these flaws, Broken Object Level Authorization (BOLA) stands out, as it allows unauthorized users to access or modify restricted data. This thesis proposes an approach based on Colored Petri Nets to detect attacks on logical vulnerabilities in REST APIs by transforming OpenAPI specifications into formal models and analyzing server logs to identify discrepancies between expected and observed behavior. To validate the approach, the `Links2CPN` tool was developed, which automatically performs the model transformation and attack detection. In three vulnerable application scenarios, the solution demonstrated over 95% accuracy and precision in identifying BOLA attack attempts. The results show that the combination of formal modeling and log analysis is capable of detecting logical vulnerabilities, offering a viable alternative for improving the security of REST APIs.

Keywords: Web API, Business Logic Vulnerability, detection of attack, OpenAPI, OWASP.

Lista de Figuras

Nomenclatura

API Interface de Programação de Aplicações (*Application Programming Interface*)

BLVs Vulnerabilidades Lógicas

BOLA *Broken Object Level Authorization*

CLF *Common Logfile Format*

CPN *Colored Petri Net*

CTF *Capture The Flag*

CVE *Common Vulnerabilities and Exposures*

DFSM *Deterministic Finite State Machine*

DSL *Domain-Specific Language*

GPL *General-Purpose Languages*

HATEOAS *Hypermedia as the Engine of Application State*

HTTP *Hypertext Transfer Protocol*

IDOR *Insecure Direct Object Reference*

JSON *JavaScript Object Notation*

MCPN *Marked CPN*

OASESS *OpenAPI Specification Extended Security Scheme*

OAS *OpenAPI Specification*

OWASP *Open Web Application Security Project Foundation*

PII Informações de Identificação Pessoal (*Personally Identifiable Information*)

REST *REpresentation State Transfer*

RPC *Remote Procedure Calls*

SOAP *Simple Object Access Protocol*

SQLi *SQL injection*

UML *Unified Modeling Language*

URL *Uniforme Resource Locators*

XSS *Cross-Site Scripting*

YAML *YAML Ain't Markup Language*

Sumário

Capítulo 1

Introdução

A crescente digitalização de processos e serviços tem trazido benefícios significativos para empresas e usuários, mas também ampliado a superfície de ataque explorada por agentes maliciosos. Aplicações web e infraestruturas digitais tornaram-se alvos constantes de ataques cibernéticos, que vão desde injeções de código e sequestro de sessões até exploração de configurações inseguras e vazamento de dados sensíveis (Fortinet, 2025). Nesse cenário, a segurança de serviços web é um tema central para a segurança dessas aplicações e dos usuários.

As APIs (*Application Programming Interface*) Web vêm sendo consideradas o núcleo do desenvolvimento moderno de softwares e como a base da transformação digital pela qual grandes companhias e o mercado em si vêm passando (SmartBear, 2020; Google Cloud, 2021). Uma API é definida como uma interface utilizada por softwares para apresentar um recurso ou uma funcionalidade para outras aplicações, sistemas de computadores, humanos e até mesmo para a Internet (Hunter, 2017; Jin et al., 2018). As APIs surgiram como interfaces altamente acopladas entre sistemas de computador, evoluindo posteriormente para se tornarem fracamente acopladas, como é o caso das APIs Web.

As APIs Web permitem que aplicações clientes acessem e interajam com serviços de uma aplicação servidora, consumindo e criando dados de forma transparente. Por exemplo, existem plataformas que oferecem, através de APIs, serviços de compartilhamento de arquivos, pagamentos através de cartões de crédito, geolocalização, *e-commerce*, entre muitos outros. Utilizando APIs, uma aplicação terceira pode ter acesso a todos esses serviços sem precisar participar do desenvolvimento e manutenção de nenhum deles.

Dentre as vantagens de uso destacam-se: a redução no tempo de desenvolvimento, a facilidade de integração entre tecnologias heterogêneas, a reusabilidade de código e o desacoplamento entre cliente e servidor, que passam a poder ser desenvolvidos de forma totalmente independente (Hunter, 2017). Contudo, o au-

mento de popularidade e adoção das APIs Web, como comprovado em (Imperva, 2024; Salt Security, 2024), foi acompanhado por um aumento no número de vulnerabilidades e riscos de segurança. De acordo com a pesquisa da Salt Security (2025), 99% dos entrevistados tiveram problemas de segurança com APIs em ambiente de produção, enquanto 34% experienciaram vazamentos de dados.

Em uma das iniciativas com a finalidade de educar as pessoas envolvidas com o desenvolvimento e manutenção de APIs sobre os riscos de segurança existentes, a OWASP (*Open Web Application Security Project Foundation*) - entidade sem fins lucrativos, criou, em 2019, o *OWASP API Security Project*, um projeto que elenca as 10 maiores ameaças às APIs Web (OWASP, 2019). Em 2023, foi lançada uma nova edição atualizando o top 10 (OWASP, 2023). Apesar desses esforços, diversas vulnerabilidades às APIs Web não contam com contramedidas ou mecanismos de detecção.

Dentre essas vulnerabilidades, destacam-se as que podem ser classificadas como Vulnerabilidades Lógicas ou *Business Logic Vulnerabilities*, falhas no projeto e na implementação das aplicações. Essas vulnerabilidades representam um desafio porque não possuem um arquétipo bem definido (Hoffman, 2020), sendo inerentes às especificidades de cada aplicação, exigindo contexto das regras de negócio do sistema para sua detecção. Além disso, usualmente, não são detectáveis por métodos automatizados, mas sim por testes manuais ou por revisões de código com foco nesse tipo de vulnerabilidade. Quando exploradas por um atacante, podem levar ao vazamento e manipulação de informações sensíveis, como dados pessoais dos usuários, fraudes e prejuízos financeiros. Um usuário mal-intencionado pode, por exemplo, alterar o status de um pedido, deletar uma conta de usuário ou adicionar dados não autorizados ao servidor. De acordo com a Salt Security (2025), os ataques contra as Vulnerabilidades Lógicas representam um risco ainda maior do que as outras vulnerabilidades, além de serem difíceis de detectar por qualquer método de detecção tradicional. Segundo relatório da Imperva (2024), 27% de todos os ataques registrados contra APIs foram alvejando Vulnerabilidades Lógicas, sendo o vetor de ataque mais explorado, seguido por Ataques Automatizados, através de bots, (19.3%) e Execução Remota de Código (12.4%).

As Vulnerabilidades Lógicas podem ser divididas em três subclasses: (1) Adulteração de Parâmetros (*Parameter Tampering*), que permite uma violação no fluxo de dados da aplicação; (2) Desvio do Fluxo da Aplicação (*Application Flow Bypass*), que permite uma violação no fluxo de controle da aplicação; e (3) Vulnerabilidade nos Controles de Acesso (*Access-Control Vulnerability*), que envolve falha no mecanismo de autenticação ou autorização da aplicação (Deepa and Thilagam, 2016; Deepa, 2018).

Um exemplo difundido de Vulnerabilidade Lógica é a primeira posição do top 10 da OWASP (2019, 2023). A *Broken Object Level Authorization* ou simplesmente

BOLA foi a primeira Vulnerabilidade Lógica a chegar à primeira posição de um dos rankeamentos da OWASP e é considerada a vulnerabilidade mais impactante do top 10 por ser simples de ser explorada, comum no mundo real e poder levar ao acesso não autorizado de informações sensíveis. Nessa vulnerabilidade, atacantes manipulam referências a objetos nas requisições da API (manipulação de parâmetros), obtendo acesso indevido a outros objetos, caso não estejam implementados mecanismos de proteção (controle de acesso).

1.1 Motivação

O aumento da oferta e adoção de serviços Web baseados em APIs implicou no aumento das preocupações relacionadas à segurança, especialmente quanto às vulnerabilidades específicas desse tipo de arquitetura. Relatório da empresa de segurança Imperva (2024) revelou que em 2023 71% do tráfego Web monitorado por eles estava associado a APIs Web. Enquanto o relatório de 2022 da empresa Salt Security (2022) apontou crescimento de 321% no tráfego Web relacionado a APIs e um crescimento de 681% no tráfego associado a ataques a esse tipo de aplicação. A OWASP, em seu projeto API Security Top 10, listou as 10 maiores ameaças às APIs Web, onde a vulnerabilidade *Broken Object Level Authorization* foi listada como primeiro lugar dentre as mais relevantes e comuns. Essa vulnerabilidade é uma refatoração da conhecida IDOR, *Insecure Direct Object Reference* (Yalon and Shkedy, 2019), que afeta aplicações Web, já havia sido classificada como Vulnerabilidade Lógica em (Li and Xue, 2014) e já foi explorada em serviços de empresas como Facebook[1], Uber[2], T-Mobile[3], Apple[4], entre outras.

De forma geral, as Vulnerabilidades Lógicas, como *Broken Object Level Authorization*, surgem da ausência de validações no lado do servidor das entradas de dados enviadas pelos usuários, incluindo a ausência ou falha no mecanismo de autorização nos recursos. As contramedidas e prevenções existentes são dependentes do caso particular da aplicação. De acordo com a OWASP (2019, 2021), as medidas preventivas contra BOLA são i) implementar mecanismos de autorização adequados que dependam das políticas e hierarquia do usuário; ii) utilizar mecanismos de autorização para verificar se o usuário logado tem acesso para

[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.

realizar a ação solicitada no registro em cada função que utiliza uma entrada do cliente para acessar um registro no banco de dados; iii) usar valores aleatórios e imprevisíveis como GUIDs para identificadores de recursos; iv) escrever testes para avaliar vulnerabilidades do mecanismo de autorização.

A adoção desses mecanismos de prevenção, por sua vez, pode se tornar trabalhosa e complexa quando não realizada desde o início do desenvolvimento da API, o que acaba facilitando o aparecimento de *bugs* e vulnerabilidades. Os desenvolvedores precisam passar a se preocupar com novas e mais específicas políticas de acesso e implementar novos mecanismos que regulem e enderecem cada recurso, tornando a aplicação mais complexa, principalmente quando existem níveis variados de autenticação e recursos que se relacionam.

Apesar de não proporem meios para detectar ataques a APIs no *API Security Top 10* (OWASP, 2021), o relatório da Salt Security (2022) destaca que 55% das empresas analisadas utilizam métodos automatizados para identificar ataques com arquétipos bem definidos, como *Cross-Site Scripting* (XSS) e *SQL injection* (SQLi), enquanto 45% das empresas analisadas utilizam análise manual de registros de eventos (logs) para identificar esses e outros ataques, como explorações de Vulnerabilidades Lógicas.

Isso porque a detecção de Vulnerabilidades Lógicas, como a BOLA, difere fundamentalmente da identificação de falhas com arquétipo definido, pois não residem em um padrão de entrada malicioso, mas sim na violação do fluxo de execução e das regras de negócio esperadas pela aplicação. Para detectá-las, deve-se primeiro ter contexto do funcionamento da aplicação, propósito e suas regras de negócio, estabelecendo um modelo do comportamento do sistema (Hoffman, 2020; Deepa and Thilagam, 2016). Nesse contexto, modelos capazes de representar os fluxos de um sistema, como as Redes de Petri Coloridas (CPNs), surgem como um formalismo essencial para a verificação de violações, atacando o cerne do problema do BOLA.

É com base na lacuna de um método para identificar ataques a Vulnerabilidades Lógicas em APIs Web que, nesta tese, propõe-se utilizar a capacidade de descrição da especificação *OpenAPI* (OpenAPI Initiative, 2020) para gerar um modelo normativo de uma API REST através de Redes de Petri Coloridas - uma abstração representando os fluxos de dados na API - e utilizar algoritmos de Verificação de Conformidade para identificar ataques a Vulnerabilidades Lógicas de Adulteração de Parâmetros e Controles de Acesso, por meio da identificação de violações entre os fluxos de dados observados e os esperados *by-design*.

No relatório *2022 State of the API Report* da Postman (Postman, 2023b,a), onde mais de 37,000 desenvolvedores e outros profissionais envolvidos com APIs compartilharam suas experiências, foi revelado que o *OpenAPI* era o formato de especificação mais usado, onde 55% dos entrevistados revelaram usar a versão

2.0 da especificação (conhecida como *Swagger*), enquanto 39% afirmaram usar a versão 3.0. A larga adoção desse formato de especificação e a sua capacidade de expressar relacionamentos entre operações da API levaram-nos a adotá-lo neste trabalho.

De acordo com Van der Aalst (2016), dependendo dos dados de entrada e das análises que precisam ser feitas, um modelo adequado e um nível de abstração devem ser escolhidos. Nesse sentido, as Redes de Petri Coloridas alinham-se diretamente com a estrutura hierárquica do OpenAPI, preservando a semântica das interações da API, enquanto permitem a verificação formal de propriedades dentro dos modelos. A capacidade de representar fluxos de controle e de dados em um único modelo também é coerente com o objetivo de detectar violações nesses fluxos, que são inerentes às vulnerabilidades BOLA, como discutido em mais detalhes nos Capítulos 4 e 5. Além disso, segundo Van der Aalst (2016), as Redes de Petri são as linguagens de modelagem de processos mais antigas e melhor investigadas que permitem a modelagem de concorrência e dependência causal em sistemas, em outras palavras, o fluxo de controle (*control-flow*) do sistema. No entanto, como destacado por Carrasquel et al. (2020), quando comparadas com Redes de Petri tradicionais, as Redes de Petri Coloridas se mostram mais apropriadas quando é necessário modelar o fluxo de dados de um sistema, além do fluxo de controle. Essas são algumas das razões pelas quais as adotamos neste trabalho.

Conformance Checking, ou Verificação de Conformidade, refere-se a análise das relações entre o comportamento planejado de um processo, descrito por um modelo, e o comportamento observado durante a execução de tal processo (Carmona et al., 2018). Além disso, a análise de *Conformance Checking* é independente do modelo sendo utilizado para descrever o processo (Redes de Petri, BPMN, CPN, etc). Dessa forma, tornando-se apropriado para os casos de uso descritos neste trabalho, que envolvem transformação de modelos e verificação de conformidade.

1.2 Objetivos

O objetivo geral desta tese é elaborar e avaliar a eficácia de um método semi-automatizado para a detecção de ataques contra Vulnerabilidades Lógicas de Adulteração de Parâmetros e Controles de Acesso em APIs Web REST, através da verificação de conformidade entre um modelo normativo e os comportamentos exibidos pela aplicação, de forma a aumentar a observabilidade sobre esses ataques e detectar a exploração de tais vulnerabilidades. O modelo normativo é desenvolvido usando Redes de Petri Coloridas, representando o fluxo legítimo de dados da aplicação, o qual é inicialmente expresso em *OpenAPI*. A identificação de divergências é feita por meio de um algoritmo de Verificação de Conformidade.

Para atingir o objetivo geral mencionado, os objetivos específicos são:

- Desenvolver uma representação em Rede de Petri Colorida (CPN) compatível com as estruturas e fluxos descritos em documentos *OpenAPI*;
- Criar e implementar um algoritmo de transformação de documentos *OpenAPI* para CPNs.
- Demonstrar a aplicabilidade do método proposto através da implementação de um algoritmo de Verificação de Conformidade sobre CPNs para identificar divergências entre modelo e comportamento real da API, extraído de logs de APIs reais e de APIs de exemplo instrumentadas.

1.3 Estrutura do Documento

Este documento está organizado em 7 capítulos (incluindo este). No Capítulo 2 são apresentados os conceitos básicos relevantes para a compreensão deste trabalho. O Capítulo 3 descreve os trabalhos relacionados, incluindo a detecção de violações com OpenAPI, ataques a Vulnerabilidades Lógicas em aplicações Web e Modelagem de APIs Web. No Capítulo 4 são discutidas transformações de modelos relacionadas a APIs REST, sendo elas, no contexto deste trabalho, especificação OpenAPI e Redes de Petri Coloridas. O Capítulo 5 apresenta o algoritmo de Verificação de Conformidade utilizado, o método proposto e detalha a prova de conceito desenvolvida. O Capítulo 6 apresenta a validação experimental e resultados obtidos nos cenários de teste. Por fim, o Capítulo 7 discute as limitações da abordagem proposta, analisa o atingimento dos objetivos e propõe trabalhos futuros.

Capítulo 2

Conceitos Básicos

Neste capítulo são apresentados os principais conceitos relacionados às APIs Web, REST, especificação OpenAPI, Vulnerabilidades Lógicas, Redes de Petri Coloridas e Verificação de Conformidade. Esses temas são abordados levando-se em consideração suas relações com o tema deste projeto e as suas definições básicas, visando auxiliar o melhor entendimento sobre o desenvolvimento do trabalho.

2.1 APIs Web

Uma API é uma interface utilizada como meio para que diferentes componentes de uma arquitetura possam interagir programaticamente, abstraindo regras de negócio e detalhes implementacionais (Jin et al., 2018). Enquanto o termo API foi utilizado inicialmente por Cotton and Greatorex Jr (1968), o termo API Web só passou a ser utilizado nos anos 2000, quando as companhias SalesForce e Ebay lançaram suas APIs Web (Kopecký et al., 2014). Uma API Web é uma interface que possibilita que um cliente Web, como um navegador ou aplicação móvel, por exemplo, se comunique e realize operações sobre dados ou recursos em um servidor Web (Hunter, 2017). Essas APIs podem ser construídas utilizando diversos paradigmas e estilos, como *Remote Procedure Calls* (RPC), *Simple Object Access Protocol* (SOAP), *REpresentation State Transfer* (REST), GraphQL, entre outros (Jin et al., 2018).

No entanto, atualmente, as APIs Web são frequentemente implementadas utilizando o padrão de design REST com requisições HTTP (*Hypertext Transfer Protocol*) como protocolo de comunicação (Postman, 2023a). Dentre os fatores que justificam essa adoção, podemos citar: (i) o suporte a diferentes formatos, como JSON e XML; (ii) o baixo acoplamento entre a aplicação cliente e o servidor, o que permite o desenvolvimento independente dessas partes; e (iii) sua escalabilidade

e a aderência ao protocolo HTTP(S), largamente utilizado em comunicações na Web (Jin et al., 2018).

2.1.1 REST

REST 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. Segundo relatório da Postman (2023a), com larga vantagem, é o estilo mais usado na implementação de APIs. Ele é baseado na manipulação de 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;
- ***Cacheabilidade***: Determinados recursos podem ser sinalizados no servidor como cacheáveis, diminuindo o volume de tráfego e aumentando o desempenho tanto do servidor como do cliente;
- **Comunicação em camadas**: A arquitetura do servidor deve ser transparente para o cliente, de forma que este possa 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 posse 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* (cada recurso da API é referido como um *endpoint* - (Hunter, 2017)), 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 a todas as 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 aos requisitos de interface uniforme e comunicação em camadas, 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. A cacheabilidade 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).

Entretanto, a natureza *stateless* introduz também desafios de segurança. Como cada requisição deve ser autossuficiente e não há persistência de estado no servidor, torna-se mais difícil gerenciar fluxos de autorização, sessões de usuários e validação de sequências lógicas de execução (fluxos de controle). De forma prática, o lado servidor da aplicação precisa implementar soluções específicas para mitigar ataques que exploram falhas de autorização, inconsistências no tratamento de estados do cliente e na validação do fluxo de controle.

De acordo com Jin et al. (2018), o recurso mais básico necessário para cada API é a documentação. Um dos padrões mais amplamente usados para documentar e descrever APIs Web é a especificação OpenAPI (Postman, 2024), que é descrita em detalhes abaixo.

2.1.2 OpenAPI

A *OpenAPI Specification* (OAS), também conhecida como especificação *OpenAPI*, é um padrão aberto e independente de tecnologias específicas para descrever APIs RESTful e APIs HTTP. Além disso, é independente de fornecedores e projetada para ser compreensível tanto por humanos quanto por softwares, o que facilita aos usuários a exploração dessas APIs, por reduzir a necessidade de acesso ao código fonte e à outras formas de documentação. A OAS é derivada da especificação *Swagger*, que foi doada para a *Linux Foundation*, pela *SmartBear Software*, para a formação da *OpenAPI Initiative* em 2015, uma iniciativa que visa difundir os benefícios da padronização na descrição de APIs.

Uma descrição de API aderente a OAS, conhecida como documento OAS, permite que uma aplicação cliente interaja com um serviço remoto com um mínimo de lógica implementacional, podendo ser usada para automatizar a geração do código fonte de servidores e clientes nas mais diversas linguagens, além de servir como base para implementação de ferramentas de teste, análise, entre outras (OpenAPI Initiative, 2020). De acordo com o relatório *2022 State of the API Report* da Postman (Postman, 2023b,a), onde mais de 37,000 desenvolvedores e outros profissionais envolvidos com APIs compartilharam suas experiências, foi revelado que o *OpenAPI* é o formato de especificação mais usado atualmente, onde 55% dos entrevistados revelaram usar a versão 2.0 da especificação (conhecida como *Swagger*), enquanto 39% afirmaram usar a versão 3.0.

Atualmente, a especificação OpenAPI encontra-se na versão 3.1.0[1] e permite a descrição de APIs HTTP de maneira geral. Na versão 3.0.0 da OpenAPI foram incluídos como parte da especificação *Links* (ou *Link Objects*). Através de *Links*, podem ser modeladas as relações entre valores retornados por uma operação e sua utilização em operações futuras, em outras palavras, o fluxo de dados. Além disso, *Links* são fundamentais para a descrição das APIs, considerando o item HATEOAS do requisito de Interface Uniforme do design REST.

De maneira concreta, um documento OAS 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. Podem ser descritos nos formatos JSON ou YAML, sendo, em outras palavras, uma estrutura composta por objetos com estruturas preestabelecidas, definindo nomenclaturas, ordens e conteúdos. A Tabela 2.1 descreve os campos base de um documento OAS.

Os terminais onde cada serviço pode ser requisitado pelo cliente são denominados *endpoints* e são usualmente representados por URLs (Jin et al., 2018) seguindo a *RFC 3986*[2]. Na especificação OpenAPI, a URL de um *endpoint*, ou

[1] Acessível em `https://spec.openapis.org/oas/v3.1.0`
[2] Acessível em `https://datatracker.ietf.org/doc/html/rfc3986`

Tabela 2.1: Componentes base de um documento OAS

Nome	**Tipo**	**Descrição**
openapi	Texto	Campo obrigatório representando o número da versão da especificação OpenAPI utilizada no documento.
info	*Info Object*	Campo obrigatório contendo metadados como título, descrição, versão da API, etc.
servers	*[Server Object]*	Uma coleção de *Server Objects* que informam as URLs base da API.
paths	*Paths Object*	Conjunto de *paths* e operações disponíveis na API. Campo obrigatório.
components	*Components Object*	Campo com um conjunto de objetos e definições reusáveis que podem ser referenciadas em diversas partes do documento.
security	*[Security Requirement Object]*	Coleção de mecanismos de segurança utilizáveis na API, por exemplo: API Key e requisitos de *OAuth2*.
tags	*[Tag Object]*	Coleção de identificadores para definir metadados e organizar os endpoints em grupos.
jsonSchema-Dialect	*Texto*	Permite o uso de um valor $schema customizado para todos os objetos contidos em um documento OAS.
externalDocs	*External Documentation Object*	Campo para referenciar documentação externa adicional.

URL absoluta, é dividida em duas informações, a URL do servidor (ou *Host*), representada por um *Server Object*, e a URL relativa (ou *path* relativo), contida no *Paths Object*. A partir daqui, se referirá a URLs relativas com simplesmente *paths*.

As nomenclaturas utilizadas na OAS para os métodos aceitos por um *endpoint* são as mesmas utilizadas na padronização do HTTP (*RFC 9110*[3]), que define 9 tipos de métodos HTTP que podem ser utilizados em requisições: *GET, POST, PUT, PATCH, DELETE, OPTIONS, HEAD*. A seguir, descrevemos algumas das mais utilizadas.

- GET: Transfere uma representação atual do recurso alvo;

[3]Acessível em: `https://www.rfc-editor.org/rfc/rfc9110.html`.

- 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;

A Listagem 2.1[4] ilustra a parte principal de um documento OAS, contendo os elementos da Tabela 2.1. Os parágrafos seguintes explicam a Listagem.

Listagem 2.1: Trecho YAML representando um exemplo de documento OpenAPI

```yaml
openapi: 3.1.0
info:
  title: Link Example
  version: 1.0.0
paths:
  /2.0/users/{username}:
    get:
      operationId: getUserByName
      parameters:
      - name: username
        in: path
        required: true
        schema:
          type: string
      responses:
        '200':
          description: The User
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/user'
          links:
            userRepositories:
              $ref: '#/components/links/UserRepositories'
  /2.0/repositories/{username}:
    get:
      operationId: getRepositoriesByOwner
      parameters:
        - name: username
          in: path
          required: true
          schema:
            type: string
```

[4]Disponível em `https://github.com/OAI/OpenAPI-Specification/blob/main/examples/v3.0/link-example.yaml`

```
34        responses:
35          '200':
36            description: repositories owned by the supplied user
37            content:
38              application/json:
39                schema:
40                  type: array
41                  items:
42                    $ref: '#/components/schemas/repository'
43            links:
44              userRepository:
45                $ref: '#/components/links/UserRepository'
46  components:
47    links:
48      UserRepositories:
49        # returns array of '#/components/schemas/repository'
50        operationId: getRepositoriesByOwner
51        parameters:
52          username: $response.body#/username
53      UserRepository:
54        # returns '#/components/schemas/repository'
55        operationId: getRepository
56        parameters:
57          username: $response.body#/owner/username
58          slug: $response.body#/slug
59    schemas:
60      user:
61        type: object
62        properties:
63          username:
64            type: string
65          uuid:
66            type: string
67      repository:
68        type: object
69        properties:
70          slug:
71            type: string
72          owner:
73            $ref: '#/components/schemas/user'
```

Na estrutura base de um documento OAS, o campo *paths* (linha 5 da Listagem 2.1) contém um objeto de tipo *Paths Object*, que é responsável por listar os *paths* e operações disponíveis. Esse objeto é composto por um conjunto de pares chave-valor, onde a chave é um *path* (linhas 6 e 25 da Listagem 2.1) e o valor é um objeto *Path Item Object* (linhas 7 e 26 da Listagem 2.1). O *Path Item Object* é responsável por descrever as operações HTTP disponíveis sobre uma *path* específico, através de um conjunto de pares chave-valor, onde a chave

é um método HTTP em formato textual, discutido anteriormente, e o valor é um objeto *Operation Object* (linhas 8 e 27 da Listagem 2.1). O *Path Item Object* pode conter uma lista de parâmetros aplicáveis às operações descritas sob esse *path*. Esses parâmetros podem ser redefinidos no nível de operação (no *Operation Object*), por isso os detalharemos nos próximos parágrafos.

O *Operation Object* especifica uma única operação sob um *path* da API, apresentando um sumário, uma descrição, um identificador único, os parâmetros da operação, os *payloads*, respostas possíveis, definições de callback e redefinições de *Server Objects* e *Security Requirement Objects*. Exploraremos em mais detalhes como são especificadas as entradas de dados de uma operação (*parameters* e *requestBody*) e as saídas de dados de uma operação (*responses*).

O campo *parameters* (linhas 9 e 28 da Listagem 2.1) contém uma coleção de *Parameter Objects* aplicáveis a operação em que está contido. Cada *Parameter Object* descreve um parâmetro utilizando os atributos obrigatórios *in* (localização da informação: *path, query, header,* e *cookie*) e um identificador, chamado *name* (linhas 10, 11, 29 e 30 da Listagem 2.1). Adicionalmente, existe o campo *schema* (linhas 13 e 32 da Listagem 2.1) que permite definir estruturas e tipos usados no parâmetro (números, *strings*, booleanos, listas e objetos) através de um *Schema Object* que é uma extensão da especificação *JSON Schema Specification Wright Draft* [5].

O campo *requestBody* contém uma coleção de objetos *Request Body Object* responsáveis por descrever o corpo das requisições (*HTTP Request Body*). O único campo obrigatório do *Request Body Object* é o *content*, contendo um mapeamento chave-valor, onde a chave é uma string representando o *media type* de acordo com o padrão *RFC6838 Media Types*[6] (por exemplo, *application/json, text/html*, etc.) e o valor é um *Media Type Object* descrevendo o corpo da requisição. Os *Media Type Objects* são compostos por *Schema Objects*, descritos no parágrafo anterior, por exemplos e detalhes de codificação (*encoding*).

As saídas de dados das operações são descritas através do campo *responses* (linhas 15 e 34 da Listagem 2.1) que contém um *Responses Object* definindo um mapeamento chave-valor, onde a chave é um *HTTP Status Code* (definido pela *RFC7231*[7]) em string (linhas 16 e 35 da Listagem 2.1) e o valor é um *Response Object*. Um *Response Object* descreve uma única resposta de uma operação, contendo o campo obrigatório *description* (linhas 17 e 36 da Listagem 2.1) para detalhar para o leitor o significado da resposta dentro do contexto da operação e contendo o campo *content* (linhas 18 e 37 da Listagem 2.1), descrevendo os formatos das respostas (*payloads*) possíveis, de forma similar ao campo *content* do

[5] `https://json-schema.org/specification.html`
[6] `https://datatracker.ietf.org/doc/html/rfc6838`
[7] `https://datatracker.ietf.org/doc/html/rfc7231#section-6`

objeto *Request Body Object* discutido no parágrafo anterior. Além de descrever as respostas possíveis de uma operação, o *Response Object* pode definir uma ligação preestabelecida entre a resposta atual e uma próxima operação, representada pelo campo chamado *link* (linhas 22 e 43 da Listagem 2.1). Esse campo contém um mapeamento chave-valor onde a chave é um nome curto de identificação e o valor é um *Link Object*.

Um *Link Object* representa um relacionamento, estabelecido em tempo de design (*design-time*), entre uma resposta e uma operação seguinte, não garantindo, porém, que o cliente será capaz de invocar com sucesso a referida operação, devido a fatores como autenticação e autorização, por exemplo. A operação relacionada à resposta deve ser identificada através dos campos *operationRef* (quando referenciada através de uma URI) ou *operationId* (quando referenciada através do nome da operação) (linhas 50 e 55 da Listagem 2.1). O *Link Object* pode descrever também como os valores da resposta podem ser utilizados nas operações seguintes, através dos campos *parameters* (linhas 51 e 56 da Listagem 2.1) e *requestBody*, dependendo do tipo de entrada esperada pela operação seguinte. A estrutura desses campos segue a descrita nos parágrafos anteriores.

Inclusive, o poder de descrição da especificação OpenAPI já está sendo utilizado para fazer verificações semiautomáticas de segurança, como em (Cheh and Chen, 2021). De acordo com os mantenedores da OAS (OpenAPI Initiative, 2023), é preferível escrever a documentação da API (ou seja, a especificação) e depois escrever o código (abordagem *Design-first*). No entanto, é possível primeiro implementar a API e depois criar a documentação usando ferramentas que transformam comentários e anotações no código em documentação, ou até mesmo escrevendo-a manualmente (abordagem *Code-first*).

2.2 Vulnerabilidades Lógicas

As vulnerabilidades no lado servidor das aplicações Web podem ser classificadas comumente em três categorias: Vulnerabilidades de Injeção, também chamadas de Validação de Entrada (*Input Validation*), Vulnerabilidades Lógicas (*Business Logic Vulnerabilities*) e Vulnerabilidades de Gerenciamento de Sessão (*Session Management Vulnerabilities*) (Li and Xue, 2014; Deepa and Thilagam, 2016; Sadqi and Maleh, 2021).

As Vulnerabilidades Lógicas, ou *Business Logic Vulnerabilities* (BLVs), são falhas no design ou na implementação da aplicação que permitem a atacantes manipularem a lógica da aplicação, induzindo-a a comportamentos diferentes dos planejados pelos desenvolvedores (Deepa and Thilagam, 2016). São usualmente fáceis de serem exploradas e difíceis de serem detectadas, já que são transações aparentemente legítimas. Considere como exemplo um aplicativo de compras

que permite ao consumidor utilizar um cupom para usufruir de desconto em determinados itens. Idealmente, o cupom pode ser usado uma única vez, mas uma falha lógica no aplicativo pode permitir que um usuário malicioso aplique o cupom um número arbitrário de vezes, proporcionando um percentual maior de desconto do que o pretendido pelos desenvolvedores.

As Vulnerabilidades Lógicas não são facilmente detectáveis através de métodos automatizados, uma vez que são transações legítimas usadas para realizar operações indesejadas, não possuem um arquétipo bem definido e são particulares para cada aplicação. Sua detecção exige contexto do objetivo do código da aplicação e suas regras de negócio, o que comumente só é alcançado através de revisões de segurança do código da aplicação (*code security review*) e testes de penetração na aplicação (*pentest*) (Hoffman, 2020). As Vulnerabilidades Lógicas podem ser subdivididas em três outros subtipos (Deepa, 2018):

- **Adulteração de Parâmetros (*Parameter Tampering*)**: Essas vulnerabilidades permitem a modificação de valores de variáveis críticas na aplicação. Os ataques são causados devido à violação das restrições semânticas das entradas do usuário, onde essa entrada pode ser fornecida pela interface gráfica do usuário ou manipulada nas solicitações HTTP. Por exemplo, dada uma aplicação de *e-commerce*, alterar o preço de um produto para 0 em uma requisição HTTP, pode permitir adquiri-lo gratuitamente se não houverem validações apropriadas considerando esse cenário.

- **Desvio do Fluxo da Aplicação (*Application Flow Bypass*)**: Ataques a essas vulnerabilidades permitem que um atacante contorne o fluxo de funcionamento (*workflow*) pretendido da aplicação, ou seja, uma violação na sequência de etapas a serem seguidas para concluir uma determinada tarefa na aplicação. Por exemplo, em uma aplicação de *e-commerce*, um atacante recebendo uma confirmação de compra sem passar pelo processo de pagamento pelos itens solicitados.

- **Vulnerabilidade de Controle de Acesso (*Access-Control Vulnerability*)**: São falhas de implementação na aplicação das regras de autenticação e autorização na aplicação que permitem o acesso não autorizado a recursos da aplicação. Essas vulnerabilidades podem ser subdivididas em desvio (*bypass*) de autenticação e desvio de autorização, de acordo com qual desses mecanismos está sendo explorado. Um exemplo de desvio de autenticação é se um usuário não logado na aplicação consegue acessar um recurso reservado apenas a usuários logados. Um exemplo de desvio de autorização é quando um usuário consegue acessar determinado recurso que tem acesso reservado apenas a outros usuários.

De acordo com Li and Xue (2014), três dos dez riscos de segurança, descritos no *OWASP Top 10 2013 - The Ten Most Critical Web Application Security Risks*, podem ser classificados com BLVs, sendo eles: *Missing Functional Access Control*, *Insecure Direct Object Reference*, e *Unvalidated Redirects and Forwards*. No *OWASP Top 10 2017*[8], *Unvalidated Redirects and Forwards* saiu do top 10 e *Missing Functional Access Control* e *Insecure Direct Object Reference* foram fundidos em um novo risco, chamado *Broken Access Control*, que é a primeira posição do atual *OWASP Top 10 2021*[9]. No *OWASP API Security Top 10 2023* (OWASP, 2023), temos como riscos associados a BLVs: *Broken Object Level Authorization* ou BOLA, a primeira posição no ranking; *Broken User Authentication*, a segunda posição; e *Broken Function Level Authorization*, na quinta posição. Mais detalhes desses riscos na próxima seção.

2.3 OWASP API Security Top 10

A OWASP (*Open Web Application Security Project*) Foundation é uma entidade sem fins lucrativos com o objetivo de atuar melhorando a segurança de softwares e da Web. A entidade é conhecida pelos seus projetos de ranqueamento Top 10, onde são elencadas, por ordem de impacto, as dez maiores vulnerabilidades de determinado segmento, baseando-se em um conjunto de dados coletados de ocorrências de explorações dessas vulnerabilidades e em pesquisas realizadas junto à comunidade de profissionais da área.

Em 2021, a entidade lançou uma nova versão do ranqueamento voltado para aplicações Web, conhecida como *Top 10 - The Most Critical Web Application Security Risks* (OWASP, 2021). Porém, dado o aumento dos riscos relacionados às APIs e sua reconhecida importância na arquitetura de aplicações modernas, em 2019 foi publicado um novo projeto voltado exclusivamente para a segurança de APIs, batizado de *API Security Top 10* (OWASP, 2019). Em 2023, foi lançada a última versão desse ranqueamento, o *OWASP Top 10 API Security Risks – 2023* (OWASP, 2023).

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** (difícil a fácil), **prevalência** (difícil a difundida), **detectabilidade** (difícil a fácil) e **impacto** (menor a severo).

As subseções abaixo discutem os ranqueamentos voltados à segurança de APIs de 2019 e 2023.

[8] Acessível em `https://owasp.org/www-project-top-ten/2017/`
[9] Acessível em `https://owasp.org/Top10/`

2.3.1 OWASP API Security Top 10 - 2019

De acordo com o *API Security Top 10* (OWASP, 2019), por natureza, as APIs manipulam e podem expor a lógica da aplicação e dados confidenciais, como informações de identificação pessoal (PII). Por causa disso, as APIs se tornaram cada vez mais um alvo para invasores, o que motivou a criação do ranqueamento específico para APIs. Foram incluídas na lista de mais relevantes as seguintes vulnerabilidades encontradas em APIs:

- API1:2019 -***Broken Object Level Authorization*** (Autorização Quebrada 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*** (Autenticação de Usuário Quebrada): 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 Dados): 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*** (Autorização Quebrada 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 seção seguinte introduz o ranqueamento *OWASP API Security Top 10 - 2023* e apresenta as principais diferenças em relação à edição de 2019.

2.3.2 OWASP API Security Top 10 - 2023

O *OWASP API Security Top 10 - 2023* (OWASP, 2023) é a segunda edição do ranqueamento da OWASP dos principais riscos às APIs Web, feita a partir da revisão de especialistas em segurança da API e feedbacks da comunidade. Da mesma forma que a primeira edição, a segunda tem como principal objetivo a educação dos envolvidos no desenvolvimento e manutenção de APIs, por exemplo, desenvolvedores, designers, arquitetos, gerentes ou organizações.

- API1:2023 -***Broken Object Level Authorization*** (Autorização Quebrada em Nível de Objeto): É equivalente ao API1:2019 - *Broken Object Level Authorization*, no entanto, (i) agora os autores salientam a importância da resposta do servidor na detecção de ataques; (ii) foi expandido o conceito de identificadores dos objetos, podendo ser inteiros sequenciais, UUIDs (*Universally Unique IDentifier*), ou strings genéricas; (iii) constatação de que a comparação entre o ID do usuário da sessão com o parâmetro de ID vulnerável do objeto não é uma solução suficiente; (iv) exemplificado que outros padrões fora REST podem ser vulneráveis, como GraphQL.

- API2:2023 - ***Broken Authentication*** (Autenticação Quebrada): Apesar da mudança de nome, é equivalente ao API2:2019 - ***Broken User Authentication***. Como principais diferenças, os autores adicionaram que i) é um indicador de vulnerabilidade permitir alteração de email, senha, e outras

informações críticas sem confirmação da senha do usuário; ii) mais foco em microsserviços que são vulneráveis se outros microsserviços o acessam sem autenticação, ou usam tokens fracos (ou previsíveis); iii) adição de exemplos com GraphQL, mostrando que a vulnerabilidade pode afetar outros padrões além de REST; iv) como contra medida, exigir re-autenticação do usuário para operações críticas, como alteração do endereço de e-mail e número de telefone 2FA da conta do usuário.

- API3:2023 - ***Broken Object Property Level Authorization*** (Autorização Quebrada a Nível de Propriedade de Objeto): A terceira posição do ranqueamento de 2019, *Excessive Data Exposure*, foi substituída por *Broken Object Property Level Authorization* em 2023. Os autores afirmaram que combinaram o *API3:2019 Excessive Data Exposure* com o *API6:2019 - Mass Assignment* com o objetivo ressaltar a causa raiz comum às duas posições: falhas de validação de autorização no nível da propriedade do objeto. Dessa forma, APIs possuem essa falha quando i) *endpoints* da API expõem propriedades de um objeto que são consideradas confidenciais e não deveriam ser lidas pelo usuário; ii) *endpoints* da API permitem que um usuário altere, adicione/ou exclua o valor de uma propriedade de objeto confidencial que o usuário não deveria poder acessar.

- API4:2023 - ***Unrestricted Resource Consumption*** (Consumo Irrestrito de Recursos): A quarta posição do ranqueamento de 2019, *Lack of Resources & Rate Limiting*, foi ligeiramente modificada para dar mais ênfase no consumo de recursos, em detrimento do ritmo em que são esgotados. Além dos critérios estabelecidos em 2019, foram adicionados para afirmar que a API é vulnerável não possuir limites para: i) tamanho máximo de arquivos para upload; ii) número máximo de operações a serem executadas em uma única solicitação de cliente de API (por exemplo, GraphQL); iii) gastos de provedores de serviços terceirizados.

- API5:2023 - ***Broken Function Level Authorization*** (Autorização Quebrada em Nível de Função): Se manteve idêntica a posição 5 do ranqueamento de 2019, basicamente sem mudanças.

- API6:2023 - ***Unrestricted Access to Sensitive Business Flows*** (Acesso Irrestrito a Fluxos de Negócios Sensíveis): Risco novo dentro do ranqueamento. Como descrito no texto da posição API3:2023, o risco da posição 6 do ranqueamento de 2019 foi combinado com a posição 3 para formar a terceira posição de 2023. De acordo com os autores, uma API possui essa falha quando expõe um fluxo de negócios sensível (como pagamento, criação de comentários, etc), sem restringir adequadamente o acesso a ele.

- API7:2023 - ***Server Side Request Forgery*** (Falsificação de Solicitação do Servidor): Novo risco no ranqueamento. O risco que ocupava a posição 7 anteriormente foi passado para posição 8. A API é vulnerável a *Server Side Request Forgery*, ou SSRF, quando busca um recurso remoto, através de uma requisição, sem validar a URL fornecida pelo usuário.

- API8:2023 - ***Security Misconfiguration*** (Erro em Configurações de Segurança): Essa posição se manteve idêntica a posição 7 do ranqueamento de 2019. Foram adicionados para afirmar que a API é vulnerável: falhas de configuração no cache e discrepâncias na forma como as solicitações recebidas são processadas pelos servidores. A posição 8 de 2019 *Injection*, saiu do ranqueamento.

- API9:2023 - ***Improper Inventory Management*** (Gestão Inadequada de Inventário): Trata-se de uma refatoração do risco *Improper Assets Management* (posição 9 de 2019). Foi adicionada que uma API é vulnerável se não há visibilidade e inventário de fluxos de dados confidenciais, o que se traduz em i) existir um "fluxo de dados confidenciais"onde a API compartilha dados confidenciais com terceiros e desatualizados; ii) não há justificativa comercial ou aprovação do fluxo; iii) versões de API antigas ou anteriores estão sendo executadas sem correção; iv) não há visibilidade profunda de qual tipo de dados confidenciais é compartilhado.

- API10:2023 - ***Unsafe Consumption of APIs*** (Consumo inseguro de APIs): Esse risco substituiu o *Insufficient Logging & Monitoring*. Os autores justificaram a mudança devido aos invasores terem começado a procurar os serviços integrados de um alvo para comprometê-lo, ao invés de atingir as APIs de seu alvo diretamente. Uma API tem essa vulnerabilidade quando: i) interage com outras APIs em um canal não criptografado; ii) não valida e limpa adequadamente os dados coletados de outras APIs antes de processá-los ou transmiti-los aos seus componentes; iii) segue redirecionamentos cegamente; iv) não limita o número de recursos disponíveis para processar respostas de serviços de terceiros; iv) não implementa *timeouts* para interações com serviços de terceiros.

Considerando que a vulnerabilidade *Broken Object Level Authorization* se encontra no topo dos ranqueamentos de 2019 e 2023, e é uma vulnerabilidade lógica, focaremos nela e a apresentaremos em mais detalhes na próxima seção.

2.3.3 API1:2023 -*Broken Object Level Authorization* (Autorização Quebrada em Nível de Objeto)

A vulnerabilidade *Broken Object Level Authorization* (BOLA), no ranqueamento de 2019, 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. No ranqueamento de 2023, a nota de detectabilidade foi alterada de 2 (mediana) para 3 (fácil), uma vez que, segundo os autores, a requisição e a resposta do servidor geralmente são suficientes para um atacante determinar se a solicitação foi bem-sucedida.

A vulnerabilidade BOLA 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 autorização dele, caracterizando uma Vulnerabilidade Lógica de Controle de Acesso. Nesse cenário, um atacante manipula identificadores que se referem a um determinado recurso, obtendo assim acesso a outros recursos aos quais este não deveria ter acesso, revelando características de uma Vulnerabilidade Lógica de Adulteração de Parâmetros. Isso acontece pelo fato de servidores desenhados pelas diretivas de design REST, por definição, não possuírem um controle 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, enviando uma requisição *POST* com as credenciais do usuário para o *endpoint* `/login` e 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/{ID}`, obtendo como resposta as informações do usuário cujo ID é (*ID01*). Em seguida, o ataque se dá com uma alteração (ou *tampering*) na requisição, pela qual o usuário malicioso substitui o ID original por outro. Assim, consegue acesso às informações de outros usuários, o que se enquadra nas definições da Vulnerabilidade Lógica de Adulteração de Parâmetros. 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;

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

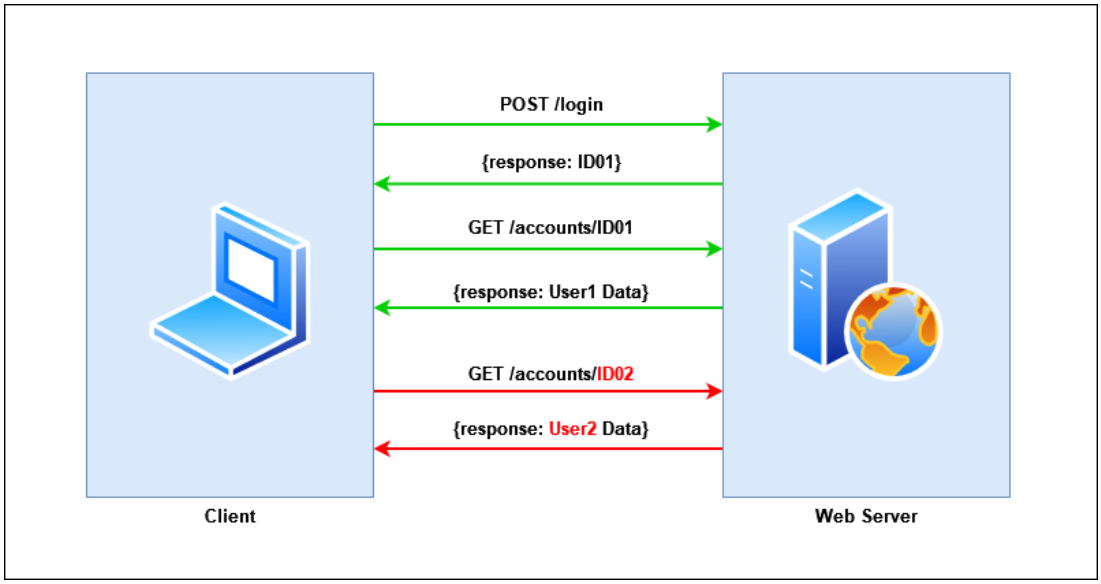


Figura 2.1: Exemplo de aplicação vulnerável a BOLA, representando requisições HTTP entre cliente e servidor. Fonte: Autor.

Como descrito no ranqueamento da OWASP, a vulnerabilidade, apesar de simples, é relativamente 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. Além disso, 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.

Apesar da vulnerabilidade BOLA ter surgido formalmente somente em 2019, ela não é nova. É uma refatoração da vulnerabilidade *IDOR*, presente na lista de vulnerabilidades críticas a aplicações web publicada pela OWASP em 2013 (OWASP, 2013). A vulnerabilidade foi reconceituada pelo fato de o 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. No entanto, analisando a evolução dos riscos publicados pela OWASP, ambas as vulnerabilidades descendem diretamente de uma vulnerabilidade que já existia desde a primeira versão do *OWASP Top 10 - The Ten Most Critical Web Application Security Risks* de 2003. A Figura 2.2 mostra a evolução da vulnerabilidade dentro das publicações do *Top 10*.

2.4 Redes de Petri Coloridas

Redes de Petri Coloridas, ou CPNs (*Colored Petri Networks*), podem ser definidas como uma linguagem gráfica para a construção e análise de modelos de sistemas concorrentes (Jensen, 1996; Jensen and Kristensen, 2009). São uma extensão de Redes de Petri que surgiram da combinação entre Redes de Petri e linguagens de programação, sendo consideradas Redes de Petri de alto nível. Elas têm esse nome porque permitem o uso de tokens que carregam valor e podem ser distinguidos entre si, ao contrário dos tokens utilizados em Redes de Petri de baixo nível, que são desenhados como círculos pretos. (Jensen, 1997). Cada valor, também chamado de *token colour*, pode ter um tipo específico (como string, inteiro, booleano, lista de valores, etc.) e cada lugar da rede (*place*) pode ser anotado com o tipo de token que aceita (*colour set*). As Redes de Petri permitem a modelagem do fluxo de execução de sistemas, como a ordem de execução, sincronização, entre outros, mas falham em descrever como os recursos devem ser processados nas execuções de atividades. Isto é tratado em alguns tipos de Redes de Petri de alto nível, como em CPNs.

De acordo com Jensen and Kristensen (2009); Gómez et al. (2019), podemos definir formalmente uma CPN com a tupla $CPN = (P, T, A, V, G, E)$, onde:

- P é um conjunto finito de *places* com cores em um conjunto finito não vazio Σ. Denotamos o *color set* de um *place* p através de Σ_p.
- T é um conjunto finito de transições, onde $P \cap T = \emptyset$.
- A é um conjunto finito de arcos direcionados, onde $A \subseteq (P \times T) \cup (T \times P)$. arcos-PT são aqueles conectando lugares com transições $(P \times T)$, enquanto arcos-TP conectam transições com lugares $(T \times P)$.
- V é o conjunto finito de variáveis tipadas em Σ, por exemplo, $Type(v) \in \Sigma$, para todos $v \in V$.
- $G : T \to EXPR_V$ é a função de guarda (*guard function*), que atribui uma expressão booleana a cada transição, por exemplo, $Type(G(t)) = Bool$. Para que uma transição seja executada, a função de guarda deve obrigatoriamente retornar verdadeiro.
- $E : A \to EXPR_V$ é a função de expressão do arco (*arc expression function*), que atribui uma expressão para cada arco. Expressões de arco resultam (*evaluate*) em multiconjuntos (*multisets*[10]) da *color set* do lugar conectado

[10]Multiconjuntos (*multisets*) são similiares a conjuntos (*sets*), tendo como diferença permitir a repetição de elementos (Jensen, 1996)

com o arco, definindo comportamentos de entrada-saída dos arcos. Para qualquer transição $t \in T$, a expressão do arco arco-PT conectado com t é chamada *expressão arco-PT de t* (respectivamente para arcos-TP).

Considerando uma CPN $N = (P,T,A,V,G,E)$, uma marcação M é definida como uma função $M : P \rightarrow \mathcal{B}(\Sigma)$, tal que $\forall p \in P, M(p) \in \mathcal{B}(\Sigma_p)$. Em outras palavras, a marcação de p deve ser um *multiset* de cores em Σ_p. Assim, uma CPN marcada é definida como um par (N,M), onde (N) é uma CPN e (M) a sua marcação.

Considerando uma CPN $N = (P,T,A,V,G,E)$, para toda transição t, $Var(t)$ denota o conjunto de variáveis que aparecem nas expressões arco-PT de t. Assim, o *binding* de uma transição $t \in T$ é uma função b que mapeia cada variável $v \in Var(t)$ em um valor $b(v) \in Type(v)$. $B(t)$ denota o conjunto de todos os possíveis *bindings* para $t \in T$. Para toda expressão $e \in EXPR_V$, $e\langle b \rangle$ denotará a avaliação (*evaluation*) de e para o *binding* b. Um elemento de *binding* é definido como um par (t, b), onde $t \in T$ e $b \in B(t)$. O conjunto de todos os elementos de *binding* é denotado por BE.

Para uma CPN marcada $MCPN = (CPN, M)$, podemos dizer que um elemento de *binding* $(t, b) \in BE$ está habilitado na marcação M quando a função de guarda de t é avaliada para verdadeiro para o *binding* $b : G(t)\langle b \rangle = true$, e para todo $p \in {}^\bullet t, E(p, t)\langle b \rangle$ está em $M(p)$[11]. Dado esse cenário, o disparo de (t, b) tem os seguintes efeitos em M:

1. Para todo $p \in {}^\bullet t$, os tokens em $E(p, t)\langle b \rangle$ são removidos de $M(p)$.

2. Para todo $p \in t^\bullet$, os tokens $E(t, p)\langle b \rangle$ são produzidos em $M(p)$.

2.5 Mineração de Processos: Verificação de Conformidade

Verificação de Conformidade (*Conformance Checking*) (Carmona et al., 2018) é um tópico da área de Mineração de Processos (*Process Mining*) (Van der Aalst, 2016) que se refere ao processo de análise da relação entre o comportamento pretendido de um processo e seu comportamento observado durante a execução. Na prática, Verificação de Conformidade é um termo usado para se referir a uma família de algoritmos que relaciona duas entradas principais, modelos de processos e registros de eventos (logs), fornecendo métodos para comparar e analisar instâncias

[11] Onde ${}^\bullet x$ denota uma pré-condição e $x^\bullet$ denota uma pós-condição, em outras palavras, $\forall x \in P \cup T : {}^\bullet x = \{y | (y, x) \in A\}, x^\bullet = \{y | (x, y) \in A\}$.

observadas de um processo em relação ao seu modelo. Um exemplo de comparação é se um processo está sendo executado conforme documentado em um modelo.

Em Mineração de Processos, um processo descreve o comportamento de um sistema através da definição de um conjunto de atividades (ou tarefas), unidades elementares de trabalho, juntamente com suas dependências causais que governam sua execução. Em outras palavras, seus fluxos de controle e dados. Essas atividades podem se relacionar de diversas formas, através de um padrão sequencial, de escolha exclusiva, execução paralela, dentre outros. Diversas notações podem ser utilizadas para descrever um mesmo processo, como Redes de Petri, UML e BPMN. Nesse contexto, podemos chamar essas notações de modelos de processo (*process models*).

Nesse contexto, algoritmos de Verificação de Conformidade contam com duas entradas, um modelo, representando um processo, e um conjunto de registros de eventos. Como resultado, obtemos informações sobre a capacidade do modelo de descrever o que é observado na prática no sistema, que podem ser expressas através de diversas métricas, como quantidade de violações, *fitness* e precisão. *Fitness* mede a capacidade de um modelo de "explicar" a execução de um processo conforme registrado em um log de eventos. Para explicar um determinado *trace* (sequência de eventos), o modelo de processo é consultado para avaliar sua capacidade de reproduzir a sequência, o que também é chamado de *token replay*, levando em consideração a lógica de fluxo de controle expressa no modelo. *Fitness* é expresso por:

$$Fitness = \frac{|L \cap M|}{|L|}$$

onde $|L \cap M|$ representa a quantidade de comportamentos que foram observados no log e estão representados no modelo, enquanto $|L|$ é a quantidade de comportamentos registrados no log. Em outras palavras, um $fitness = 1$ representa que todos os comportamentos observados estão de acordo com o modelo. *Fitness* é considerada a principal medida para avaliar se um modelo é adequado para explicar o comportamento registrado (Carmona et al., 2018).

Precisão exprime quanto do modelo do processo aparece no log de eventos registrados, quantificando o comportamento do modelo em relação ao log. Assim, um modelo subespecificado, que nos permite fazer qualquer coisa, tem baixa precisão.

$$Precisao = \frac{|L \cap M|}{|M|}$$

Da mesma forma que para *fitness*, $|L \cap M|$ representa a quantidade de comportamentos que foram observados no log e estão representados no modelo e $|M|$

representa o número total de comportamentos no modelo.

A comparação entre o comportamento de um modelo de um processo e o comportamento registrado em um log de eventos pode resultar em medidas de conformidade globais e locais. Um exemplo de medida de conformidade global é "85% dos casos no log de eventos podem ser reproduzidos pelo modelo", enquanto um exemplo de medida de conformidade local é "a atividade x foi executada 15 vezes, embora isso não fosse permitido de acordo com o modelo".

As divergências entre modelo e comportamento real podem ter diversas causas. Para o caso onde um comportamento ocorre apenas no mundo real (não ocorre no modelo), pode-se assumir que uma atividade deveria ter sido modelada e não foi, um comportamento incorreto aconteceu no sistema ou o log de eventos contém um erro. Para o caso onde uma atividade esperada no modelo não ocorreu no sistema, apesar do processo ter sido executado, pode-se considerar qual atividade foi executada no sistema, mas o log falhou em registrar a execução da atividade ou pode-se considerar que a atividade não foi executada.

Apesar de os algoritmos de Verificação de Conformidade terem sido originalmente desenvolvidos para análises *offline*, após a execução do processo, existem trabalhos propondo sua aplicação *online*, durante a execução do processo, como (Burattin and Carmona, 2017; van Zelst et al., 2019).

2.6 Conclusão do Capítulo

Neste capítulo foram vistos os conceitos necessários para a compreensão desta tese. 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 introduzido o conceito de Vulnerabilidades Lógicas e a dificuldade que representam para processos de detecção automatizados. Foi apresentada a fundação OWASP, uma entidade sem fins lucrativos que trabalha com a divulgação dos riscos de segurança para a Web, contramedidas, e com seus projetos de ranqueamento Top 10, que elencam as principais vulnerabilidades em determinado segmento. As vulnerabilidades que formam o *API Security Top 10* foram brevemente descritas. Após isso, foi apresentada em mais detalhes e exemplificada a vulnerabilidade que se encontra em primeiro lugar no ranqueamento da OWASP *API Security*, *Broken Object Level Authorization* (BOLA), que representa um caso de estudo para a análise das Vulnerabilidades Lógicas. Finalmente, foram introduzidos os conceitos de Rede de Petri Colorida, que é uma linguagem gráfica para a construção e análise de modelos de sistemas concorrentes, e Verificação de Conformidade, que permite analisar divergências entre modelos de sistemas e comportamentos observados nesses sistemas.

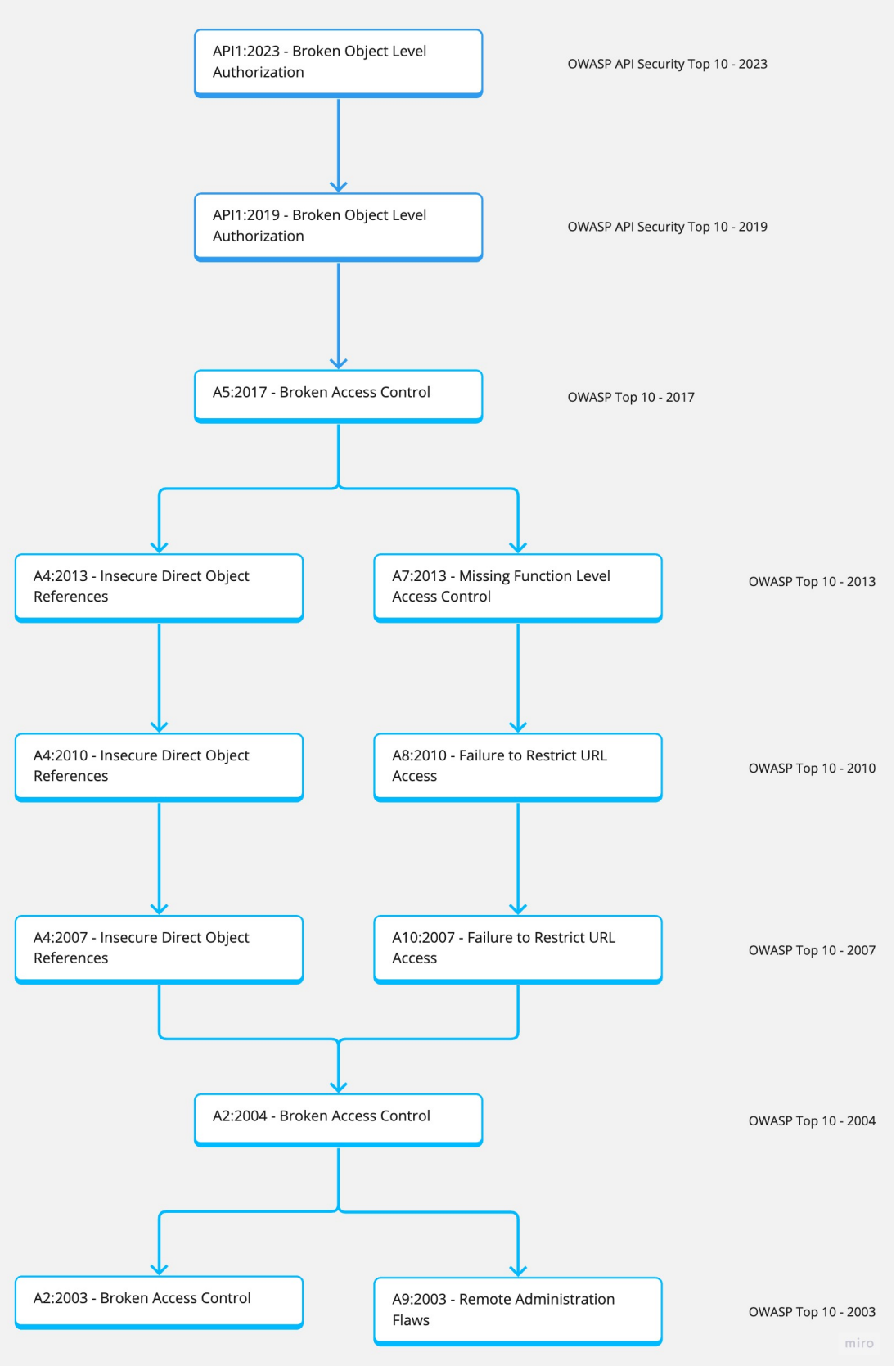


Figura 2.2: Evolução das vulnerabilidades no OWASP até o surgimento do BOLA.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são apresentados trabalhos relevantes para o tema, focando em pesquisas que abordam a detecção de ataques e vulnerabilidades em APIs Web através de um modelo normativo inicial, incluindo abordagens de modelagem de fluxos de comunicação em APIs RESTful, testes automatizados em APIs Web e análise de especificações OAS para extração do relacionamento entre operações. Começamos revisando a literatura sobre estudos que utilizam a especificação OpenAPI para detectar ataques e vulnerabilidades em APIs REST. Em seguida, analisamos modelagens de APIs REST usando linguagens específicas e não específicas de domínio.

3.1 Revisão da Literatura

Devido ao caráter recente de preocupação em segurança de APIs Web, a maior parte do trabalho nesse sentido tem caráter prático e vem de entidades formadas por profissionais, em contraste com trabalhos realizados por instituições acadêmicas. Apesar da quantidade de trabalhos acadêmicos relacionados especificamente ao tema ser relativamente pequena, foram encontradas uma revisão da literatura recente (Barabanov et al., 2022) e uma revisão sistemática (Loebens, 2022). A primeira focada na detecção de vulnerabilidades BOLA e a segunda focada na detecção de ataques contra vulnerabilidades BOLA. Dessa forma, foi adotada uma 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, com o objetivo de estender a revisão da literatura existente.

Além disso, foram consideradas definições de vulnerabilidades estabelecidas em recursos reconhecidos (ex.: 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, 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 (Barabanov et al., 2022), já identificada há alguns anos, foram pesquisadas também técnicas de detecção desta vulnerabilidade.

A quantidade de trabalhos focados na detecção/mitigação de vulnerabilidades lógicas em APIs Web denota que os avanços na detecção desse tipo de ataque têm sido lentos, sendo os esforços focados no processo de desenvolvimento. As próximas seções apresentarão esses trabalhos.

3.1.1 Detecção de Ataques e Vulnerabilidades com OpenAPI

Até onde foi possível verificar na literatura, ainda não existem estudos que utilizem a especificação OpenAPI como um modelo normativo para detectar ataques contra APIs Web REST. No entanto, alguns trabalhos utilizam essa especificação para realizar testes de segurança e revelar vulnerabilidades. A seguir, discutimos como esses estudos utilizam o OpenAPI e os desafios que enfrentam.

Barabanov et al. (2022) apresentaram o primeiro algoritmo para detecção automática de vulnerabilidades relacionadas a IDOR/BOLA em APIs através da análise estática de especificações OpenAPI. Os autores conduziram uma revisão da literatura nas principais bases digitais da literatura acadêmica (ScienceDirect, ACM Digital Library, IEEE Xplore, Springer Link) e na literatura cinzenta (apresentações em conferências, relatórios de segurança, blogs e vídeos) com o objetivo de agrupar os ataques considerando em que parte da requisição o vetor de ataque se encontra (na URL, corpo da requisição, ou cabeçalho) e se exige conhecimento prévio. Após isso, realizaram análises para determinar possíveis propriedades de especificações OpenAPI usadas para descrever *endpoints*, a fim de entender quais propriedades poderiam ser uma evidência ou estarem relacionadas a potenciais vulnerabilidades BOLA.

Em seguida, os autores propuseram um algoritmo para analisar especificações OpenAPI e identificar potenciais vulnerabilidades BOLA. O algoritmo consiste em i) avaliar uma especificação OpenAPI válida, definindo valores de atributos relacionados a potenciais vulnerabilidades BOLA e anotando-os na especificação; ii) usar a especificação anotada para determinar quais técnicas de ataque são aplicáveis. Caso seja determinado que alguma técnica de ataque é aplicável, a combinação de *endpoints*, operações e parâmetros é considerada potencialmente vulnerável e precisa ser testada posteriormente, manualmente, por um analista.

Para avaliar a abordagem proposta, os autores conduziram dois experimentos. No primeiro cenário, os autores geraram exemplos de especificação que conti-

nham pelo menos uma vulnerabilidade de cada tipo analisado. Para o segundo cenário, os autores usaram especificações de API públicas que continham potenciais vulnerabilidades, de acordo com a análise manual dos autores. Embora a abordagem tenha sido testada em dois cenários diferentes, o algoritmo apresentou alguns falsos positivos e falsos negativos, indicando que são necessários mais refinamentos.

Em sua Dissertação, Loebens (2022) propôs uma abordagem de detecção e mitigação em tempo de execução de tentativas de exploração da vulnerabilidade BOLA, através da identificação e verificação de relacionamentos produtor-consumidor entre os *endpoints* descritos nas especificações OpenAPI da API. Um produtor é um *endpoint* que responde com identificadores de recursos da API, enquanto um consumidor é um *endpoint* que espera receber um identificador de recurso e realiza alguma operação sobre ele.

Inicialmente, o autor conduziu uma revisão de trabalhos na literatura em bola de neve (*snowball sampling*) a partir de artigos cuja base pudesse ser adaptada à detecção de ataques BOLA em APIs Web e, em seguida, realizou uma revisão sistemática. A abordagem proposta possui um formato de *proxy*, consistindo em 3 elementos: um validador de requisições, um classificador e uma *allowlist*. O validador verifica se a requisição está conforme o descrito na especificação OpenAPI. O classificador categoriza a requisição como do tipo produtor ou consumidor e faz a verificação na *allowlist* para confirmar se aquela requisição está permitida. Caso positivo, a requisição segue o fluxo normal ao servidor e, em caso negativo, é identificado um caso de tentativa de ataque BOLA.

Para verificar a eficácia da abordagem proposta, o autor utilizou aplicações educacionais de código aberto, desenvolvidas para testes de segurança - *crAPI* (Piyush et al., 2021), *PIXI* (Becher and contributors, 2020) e *VAmPI* (Anonymous, 2021) - e reproduziu as explorações das vulnerabilidades. Apesar de a metodologia proposta ter sido capaz de detectar e evitar tentativas de exploração a vulnerabilidades BOLA, não há dados relacionados a falsos positivos, falsos negativos ou eficiência. Além disso, o autor destacou que, para a utilização do método, é necessária a existência de uma descrição completa da especificação OpenAPI da API que se deseja monitorar, incluindo os *Links* das respostas com as próximas requisições.

Em Haddad and Malki (2022), os autores propuseram uma extensão à especificação OpenAPI, chamada OAS ESS (*OpenAPI Specification Extended Security Scheme*). Esta extensão visa incluir controles de segurança declarativos para objetos, que podem ser posteriormente utilizados em tempo de execução por módulos de autorização personalizados para realizar novas verificações de autorização e mitigar ataques BOLA. No entanto, segundo os autores, a abordagem proposta ainda precisa ser testada em implantações reais de APIs e a implementação do

módulo de autorização depende da tecnologia utilizada na construção da API. Neste trabalho, foi implementada para o framework *FastAPI*. Por fim, esta abordagem exige que os desenvolvedores atualizem manualmente as especificações OpenAPI para incluir as novas estruturas e adaptar ferramentas existentes que interagem com a especificação, o que poderia dificultar sua adoção mais ampla.

Da mesma forma Barabanov et al. (2022) e Deng et al. (2023) propuseram o uso de uma especificação OpenAPI anotada que codifica as relações de operação e as estratégias de geração de parâmetros para APIs RESTful. No entanto, ao contrário de trabalhos anteriores, o objetivo aqui é o teste de penetração de diferentes tipos de vulnerabilidades em APIs RESTful. Os autores apresentaram uma ferramenta automatizada chamada *NAUTILUS* que processa a especificação anotada e testa vulnerabilidades criando sequências de requisições de API. As relações entre operações são automaticamente inferidas usando convenções de nomes de atributos na descrição de requisições e respostas, juntamente com heurísticas, como, por exemplo, produtor-consumidor. Porém, segundo os autores, o processo automático de inferência pode levar a relações conflitantes entre operações, em certas situações. A abordagem proposta foi comparada com quatro scanners de vulnerabilidade e ferramentas de teste de API RESTful em seis serviços RESTful, demonstrando sua capacidade de detectar vulnerabilidades como *access bypass*, *directory traversal* e injeção SQL. No entanto, as especificações OpenAPI usadas nos testes foram anotadas manualmente. Os autores sugeriram integrar o processo de anotação na fase de criação do OAS, que é tipicamente realizado por desenvolvedores.

Em Du et al. (2024), os autores propuseram um framework de inspeção de APIs para teste de segurança, chamado VOAPI [2], com o objetivo de identificar vulnerabilidades SSRF (*Server-side request forgery*), uploads sem restrições, *Path Traversal*, injeção de comandos, injeção SQL e XSS (*Cross-site scripting*). O framework se baseia na identificação de funções da API associadas a vulnerabilidades dentro das especificações da API e na condução de testes de segurança direcionados a essas funções. Por exemplo, interfaces de upload de arquivos podem apresentar risco de uploads maliciosos, enquanto interfaces de proxy podem ser vulneráveis a SSRF. Os autores analisaram um conjunto de CVEs (*Common Vulnerabilities and Exposures*) relacionados a APIs para estabelecer conexões entre as palavras-chave, as funcionalidades correspondentes da API e os tipos de vulnerabilidade. Em seguida, eles utilizaram algoritmos introduzidos em Atlidakis et al. (2019) para analisar a especificação da API e extrair possíveis tipos de vulnerabilidades e operações da API. Após essa etapa, para cada operação da API a ser testada, os autores aplicaram um algoritmo para criar uma sequência de requisições à API para alcançar um estado no qual fosse possível executar corretamente a operação sob teste. O algoritmo é semelhante ao apresentado

em Deng et al. (2023), diferindo pela presença de uma heurística extra, *produtor-consumidor*, também usada em tal artigo. Posteriormente, um Gerador de Casos de Teste é utilizado para tentar explorar potenciais vulnerabilidades, e um Verificador de Vulnerabilidade Baseado em Feedback para verificar se a operação é de fato vulnerável. A abordagem foi testada com 7 APIs RESTful do mundo real e comparada com 5 outras ferramentas que detectam as mesmas vulnerabilidades. Embora a Geração de Sequência de Testes tenha sido comparada a outros métodos automatizados, ela não foi comparada a casos em que Links estão disponíveis na especificação da API para indicar sequências de solicitações válidas. De acordo com os autores, nos testes realizados, a Geração de Sequência de Testes conseguiu alcançar 62,7% de cobertura sobre os *endpoints*.

3.1.2 Detecção de Ataques e Vulnerabilidades Lógicas em Aplicações Web

Em Bernardi et al. (2017), os autores propuseram um método baseado em *Process Mining* e *Model-Driven Engineering* para identificar determinados padrões de ataques a aplicações Web. A princípio, a aplicação que se deseja proteger é modelada através de diagramas *UML Behavioral Diagrams* que capturam o fluxo de funcionamento da aplicação. Em seguida, esses diagramas são convertidos em Redes de Petri, o que é chamado de modelo normativo (*normative model*). Por fim, são aplicados algoritmos de *Process Mining* sobre o modelo normativo e um conjunto de logs representando as requisições HTTP recebidas pela aplicação Web, com o objetivo de identificar divergências entre o comportamento observado e o definido pela Rede de Petri.

A abordagem proposta tem como vantagens ser independente das tecnologias utilizadas na construção da aplicação Web e, uma vez que utilizam algoritmos já existentes de *Process Mining*, poder ser aplicada *offline* (depois da execução, através de logs) ou ser adaptada pra ser aplicada online (em tempo de execução). No entanto, os autores relataram que podem haver dificuldades ao utilizar a abordagem *offline*, processando logs de eventos que ocorreram anteriormente, porque os logs podem não conter todas as informações necessárias para a análise. Além disso, caso o modelo UML existente da aplicação, que é escrito manualmente por um desenvolvedor, não represente fidedignamente a aplicação, o método tende a falhar e, consequentemente, gerar falsos positivos e negativos.

Para demonstrar a aplicabilidade da abordagem proposta, os autores aplicaram-na como caso de estudo em um sistema Web da universidade dos autores, chamado *SID Digital Library*, onde foram analisados 10 anos de logs e foram detectados diversos tipos de ataque contra a aplicação, como *cross-site scripting* (XSS) e ataques de força bruta.

Em Menemencioğlu and Orak (2017), os autores propuseram um mecanismo baseado em Máquinas de Estado Finitas Determinísticas, *Deterministic Finite State Machine* ou DFSM, para prevenir ataques a vulnerabilidades de Adulteração de Parâmetros, combinando análises dinâmica e estática. Inicialmente, todos os parâmetros utilizados na aplicação são classificados em uma classe dentre três possíveis: numérico, não numérico e id. Cada parâmetro é modelado através de uma DFSM com três estados (estado inicial, válido e inválido), onde a função de transição de estado é uma função que verifica se o parâmetro é válido. A partir daí, o conjunto de todos os valores possíveis para essas classes de parâmetros é armazenado em uma base de dados antes da execução da aplicação. Quando a aplicação é executada, antes de qualquer requisição ser processada, é verificado se os parâmetros passados na requisição são válidos (existem na base de dados), transicionando a DFSM do parâmetro.

Semelhante ao trabalho de Bernardi et al. (2017), os autores avaliaram sua abordagem através de testes com uma aplicação da universidade dos autores. Neste caso, foram analisados 3 meses de logs de uma aplicação Web chamada *Academic Curriculum Vitae Based Faculty Information System*. De 825,930 sessões registradas no período, 54,098 ataques foram detectados.

A solução proposta pelos autores é principalmente teórica, uma vez que a análise manual de todos os parâmetros de uma aplicação e seus valores possíveis não é factível em grandes aplicações. No entanto, os resultados dos testes da abordagem em uma aplicação real revelam que ela é capaz de detectar com sucesso ataques, impondo um *overhead* temporal de 1.2 milisegundos por requisição, nos testes realizados.

Em Deepa et al. (2018), os autores propuseram uma ferramenta chamada **DetLogic** para a detecção das Vulnerabilidades Lógicas de Adulteração de Parâmetros, Desvio do Fluxo da Aplicação e Vulnerabilidades de Controle de Acesso em aplicações Web, utilizando uma abordagem caixa preta. Na abordagem proposta, inicialmente, extrai-se o comportamento legítimo da aplicação (*control-flow* e *data-flow*), através de um conjunto de *traces* (um registro das requisições HTTP geradas pelo uso da aplicação). Então, a ferramenta utiliza uma máquina de estados finitos (FSM) anotada para modelar o comportamento observado na aplicação. Baseado nesse modelo, são gerados vetores de ataque a partir de restrições (*constraints*) com o objetivo de identificar as vulnerabilidades. Os vetores de ataque são executados e as respostas obtidas são comparadas com as obtidas durante a execução normal da aplicação.

Para testar a abordagem proposta, os autores utilizaram 4 aplicações Web de *benchmark*, isto é, aplicações desenvolvidas com vulnerabilidades utilizadas para testar a eficácia de *scanners*. Os autores reportaram como resultados 99.1% de precisão e 97.9% de verdadeiros positivos.

Devido à ferramenta utilizar uma abordagem caixa preta, é possível que os *traces* obtidos para a criação do modelo da aplicação não representem a aplicação em sua totalidade. As *constraints* utilizadas para a geração dos vetores de ataque são dependentes das tecnologias utilizadas na construção da aplicação Web. Por exemplo, as *constraints* utilizadas de Controle de Acesso são relacionadas a sessões armazenadas no lado do servidor. Para o caso de APIs REST, por exemplo, devido à propriedade *Statelessness* do REST, as sessões são armazenadas no lado do cliente. Exigindo que a ferramenta proposta seja adaptada de acordo com a aplicação que se deseja usar. Além disso, existem limitações quanto a tecnologias, por exemplo, a solução não suporta aplicações com clientes dinâmicos que executam requisições *AJAX*.

No trabalho de Filho and Feitosa (2019) foi demonstrado que, utilizando a diagramação **RESTalk**, o comportamento típico exibido por *API Scrapers* (*bots* com o objetivo de extrair informações de uma API) é fundamentalmente diferente do de clientes legítimos da API, permitindo a identificação desses bots. Enquanto a sequência de requisições de um cliente legítimo obedece à 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 do diagrama **RESTalk** é manual, tornando-se 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.

A Tabela 3.1, sumariza a discussão das duas últimas subseções sobre os trabalhos que abordaram a detecção de ataques e vulnerabilidades lógicas com OpenAPI e em aplicações web. Na tabela, a segunda coluna destaca o mecanismo de detecção sendo usado no método proposto; a terceira coluna apresenta se o método pode ser executado online ou offline; a coluna seguinte informa se a abordagem é independente da tecnologia de implementação da aplicação sendo analisada ou protegida; a quinta coluna descreve se o trabalho discute o uso do método com Vulnerabilidades Lógicas; a última coluna apresenta se o trabalho discute a aplicação dos métodos em APIs Web.

3.1.3 Modelagem de APIs Web

Apesar da larga adoção do estilo de arquitetura REST para o desenvolvimento de APIs Web, ainda permanece a discussão sobre a forma mais apropriada de modelar e visualizar graficamente uma API REST. Diversos autores propuseram diferentes abordagens e possíveis soluções para essa questão, como, por exemplo, a criação de Linguagens de Domínio Específico, ou *Domain Specific Language*

Tabela 3.1: Comparação Trabalhos de Detecção de Vulnerabilidades e Ataques

Trabalho	**Mecanismo de Detecção**	**Tempo de Execução**	**Independente Tecnologia**	**Vuln. Lógicas**	**APIs**
Barabanov et al. (2022)	Análise estática de especificações OpenAPI anotadas para detectar possíveis vulnerabilidades	Offline	Sim	Sim	Sim
Loebens (2022)	Análise dinâmica de requisições baseada na conformidade com a especificação *OpenAPI*	Online	Sim	Sim	Sim
Haddad and Malki (2022)	Análise dinâmica de requisições baseada na conformidade com a especificação *OpenAPI* estendida	Online	Não	Sim	Sim
Deng et al. (2023)	Análise estática de especificações OpenAPI anotadas e envio de requisições para testar vulnerabilidades	Offline e Online	Sim	Sim	Sim
Du et al. (2024)	Análise estática de especificações OpenAPI anotadas e envio de requisições para testar vulnerabilidades	Offline e Online	Sim	Não	Sim
Bernardi et al. (2017)	Process Mining sobre UML para detectar divergências com o modelo normativo	Offline	Sim	Sim	Não
Menemencioğlu and Orak (2017)	Modelagem em Deterministic Finite State Machine (DFSM) para validar parâmetros e identificar adulterações	Offline e Online	Sim	Sim	Não
Deepa et al. (2018)	Máquina de estados finitos (FSM) para detectar divergências com o comportamento legítimo	Online	Não	Sim	Não
Filho and Feitosa (2019)	Modelagem em RESTalk para representar o comportamento legítimo e divergências	Offline	Sim	Não	Sim

(DSL) (Ivanchikj, 2021), o uso de Cálculo de Processos (*Process calculus*) (Wu and Zhu, 2016) e Redes de Petri (Decker et al., 2009; Li and Chou, 2011; Kallab et al., 2017).

Nesse cenário, Linguagens de Domínio Específico são concebidas para serem capazes de modelar todas as características e comportamentos de uma API REST. Porém, apresentam uma curva de aprendizado associada à necessidade de aprender uma nova linguagem, além de carecerem de todo o ecossistema de ferramentas para que desenvolvedores de software possam adotá-las como ferramentas do dia-a-dia (editores gráficos e textuais e ferramentas de simulação, por exemplo). Enquanto

isso, Linguagens Sem Domínio Específico, como as linguagens formais tradicionais, podem não atender inteiramente a todas as propriedades e comportamentos das APIs REST. No entanto, possuem uma curva de aprendizado menor, por já serem difundidas e potencialmente utilizadas em áreas adjacentes. Além disso, contam com ferramentas consolidadas tanto na comunidade acadêmica quanto na indústria, o que facilita sua adoção. Dessa forma, observamos uma dualidade entre capacidade de modelagem e usabilidade, em outras palavras, quão específica a modelagem é e quão simples é o seu uso para seus usuários.

Ivanchikj (2016) propôs uma Linguagem de Domínio Específico (*Domain Specific Language* ou DSL) para modelar fluxos de mensagens entre cliente e servidor que obedecem aos requisitos do padrão de design REST. O principal objetivo era a obtenção de uma ferramenta que permitisse a modelagem 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 ferramenta para 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 de 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, os autores propuseram uma ferramenta para detecção de padrões nessas *conversas*, através do uso de logs (Ivanchikj et al., 2018a), o que possibilitou investigações dos comportamentos das APIs e permitiu buscas por padrões conhecidos ou extração automática de padrões dos registros.

Os trabalhos para o desenvolvimento da ferramenta **RESTalk** demonstraram 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 a 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.

Em Decker et al. (2009), os autores introduziram um modelo formal para

process enactment em sistemas REST (não RESTful) usando *Service Nets*, uma classe de Redes de Petri. Posteriormente, em (Alarcon et al., 2010), os autores utilizaram o formalismo introduzido por Decker et al. (2009) para converter sistemas REST descritos através de uma linguagem de descrição chamada ReLL (*Resource Linking Language*), para uma *Service Net*. No entanto, conforme discutido em (Kallab et al., 2017), essas abordagens ignoram links de hipermídia internos, descrevem todos os tokens apenas em XML, não verificam o comportamento de composição da API e não descrevem APIs RESTful, apenas REST.

Em Li and Chou (2011, 2015), os autores propuseram um framework de modelagem XML baseado em Petri-Net, chamado *REST Chart*, para descrever APIs REST sem violar restrições REST. Sua abordagem é baseada em modelar APIs REST como um conjunto de representações hipermídia e transições entre elas, onde cada transição especifica as possíveis interações com o recurso referenciado por um hiperlink em uma representação. Devido ao objetivo de não violar as restrições REST, a modelagem proposta se torna mais abstrata, não lidando com detalhes implementacionais da API, como protocolos, *endpoints*, etc. A figura 3.1 mostra um exemplo da modelagem proposta pelos autores, composta por uma transição conectando dois lugares de entrada e um lugar de saída. Essa Rede de Petri representa que um cliente pode seguir um link hipermídia (*hyperlink*), contido no recurso *login* para mudar seu estado de login para conta, através do consumo de um recurso chamado *credenciais*. Apesar de ser capaz de modelar as APIs REST sem violar nenhuma das restrições, a abordagem proposta pelos autores não modela o fluxo de dados da API e adota um nível de abstração maior do que o da abordagem proposta neste trabalho, tornando a adoção da linguagem mais difícil. Além disso, ela adota o uso da linguagem XML para descrever as APIs textualmente, em contrapartida ao proposto por esse trabalho, que é utilizar *OpenAPI Specification*, que é o padrão mais utilizado atualmente na descrição de APIs.

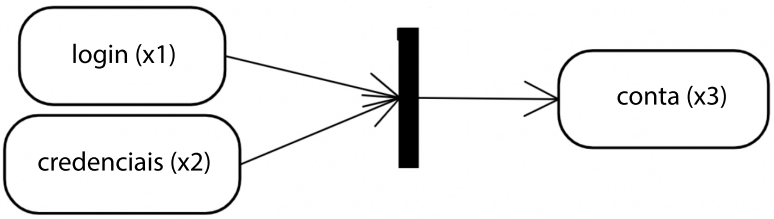


Figura 3.1: Exemplo de uma API modelada através de um Rede de Petri com interação sobre uma transição. Fonte: Tradução e simplificação da figura em (Li and Chou, 2011).

Em Kallab et al. (2017); Kallab (2019), os autores propuseram uma linguagem

formal baseada em Redes de Petri Coloridas (CPNs), para verificar a composição de serviços RESTful. Eles definiram tipos de dados como cores, uma definição única de um recurso como um serviço identificado por um *URI* e focaram no uso de propriedades CPNs para verificar os comportamentos de composição da API. Na Figura 3.2 é mostrado um exemplo dessa modelagem. Podemos ver duas transições, representando as URLs da API e 9 novos lugares. Os lugares inscritos com *Req* indicam as requisições aos *endpoints*, enquanto os lugares com *Stat* representam os *Status Codes* que resultaram da chamada à API. Os demais lugares representam informações de entrada e saída da API. Esta modelagem adota um nível de abstração mais baixo do que a discutida no último parágrafo, facilitando a adoção dos usuários. No entanto, os autores não propuseram uma linguagem textual para a descrição das APIs ou discutiram como sua abordagem pode ser usada para representar vários usuários simultâneos utilizando a API.

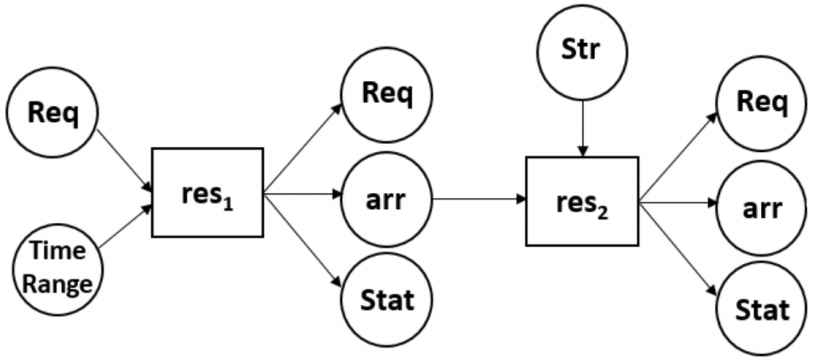


Figura 3.2: Exemplo de uma composição de API modelada através de uma Rede de Petri. Círculos representam lugares e retângulos representam transições. Fonte: Simplificação da figura em (Li and Chou, 2011).

.

A Tabela 3.2, sumariza a discussão desta seção sobre os trabalhos que abordaram a modelagem de APIs Web REST. Na tabela, a segunda coluna destaca a linguagem de descrição sendo utilizada no método proposto; a terceira coluna apresenta o tipo da linguagem; a última coluna apresenta se a modelagem proposta captura todos os requisitos do *REST*, sendo *RESTful*.

3.2 Conclusão do Capítulo

Quanto a modelagem de APIs, nos trabalhos onde foram propostas as ferramentas RESTalk e RESTler, comentados nas seções anteriores, quando trazidos para o contexto da detecção de ataques a APIs, servem como ponto de partida para a

Tabela 3.2: Comparação Trabalhos de Modelagem de APIs Web

Trabalho	**Linguagem**	**Tipo**	**RESTful**
Ivanchikj (2016) Ivanchikj et al. (2018a,b)	RESTalk	Linguagem de Domínio Específico	Não
Decker et al. (2009) Alarcon et al. (2010)	Service Nets e ReLL	Linguagem de Domínio Específico e Modelagem Formal	Não
Li and Chou (2011, 2015)	REST Chart	Linguagem de Domínio Específico	Sim
Kallab et al. (2017) Kallab (2019)	Colored Petri Nets	Linguagem de Modelagem Formal	Sim
Bernardi et al. (2017)	Petri Nets e UML	Linguagem de Modelagem Formal	Não

concepção da solução proposta nesse trabalho. Os autores da ferramenta RESTalk demonstraram que em uma API REST, a obtenção de um recurso não é feita de forma direta, mesmo que no design REST o controle de estados esteja ausente do lado do servidor. Para que determinado cliente obtenha um recurso de seu interesse, ele pode necessitar executar uma série de requisições, em uma determinada ordem, trilhando um caminho para a obtenção do recurso. O trabalho de Kallab et al. (2017) reforça esse padrão, expandindo-o para a composição de diferentes APIs envolvidas em uma única requisição do cliente. A ordem de execução das requisições pode ser modelada como uma arquitetura produtor-consumidor, onde um *endpoint* produz um identificador primeiro, para posteriormente outro *endpoint* consumi-lo.

Ambos os trabalhos apontam para a existência de fluxos de execução em uma API REST. Em cenários comuns, recursos não são acessados diretamente, ou seja, uma aplicação legítima segue um fluxo de requisições que permite ao cliente adquirir as informações necessárias para executar uma requisição seguinte de forma bem sucedida e que retorne os recursos desejados. 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.

Esse cenário é explorado 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 em 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*. 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.

No trabalhos de Bernardi et al. (2017) e Kallab et al. (2017) é demonstrado que modelos normativos podem ser utilizados no mundo real para detectar divergências entre o modelo teórico e o comportamento observado na prática. Sendo que essas divergências podem ser divergências de fato entre o modelo real e o teórico, ou na violação de propriedades e comportamentos que a aplicação deveria ter. Esse mesmo conceito é explorado neste trabalho.

Até onde foi possível verificar na literatura, enquanto vários trabalhos propuseram ferramentas e métodos para a detecção de vulnerabilidades em aplicações web e em testes automatizados, somente em Loebens (2022) e Haddad and Malki (2022) foi abordada diretamente a detecção de ataques BOLA em APIs Web REST. Este trabalho contribui para preencher essa lacuna, oferecendo uma abordagem para detectar ataques BOLA usando redes de Petri Coloridas derivadas de especificações OpenAPI. Além disso, como mostrado, apesar de muitos trabalhos também considerarem a especificação OpenAPI como uma fonte de verdade para estabelecer um *baseline* de como a API deve funcionar, eles se concentram na modelagem de sistemas REST de maneiras não padronizadas, através de extensões nas especificações OpenAPI, o que pode tornar difícil o suporte e a adoção dessas abordagens. Ao invés disso, na abordagem proposta, o fluxo de dados da API é modelado respeitando a especificação OpenAPI.

Dessa forma, consideramos as abordagens apresentadas em (Barabanov et al., 2022; Loebens, 2022) como complementares a esta, que podem fornecer outras análises de vulnerabilidade como forma de detectar falsos positivos (ou falsos negativos) da abordagem anterior. Ainda que existam soluções que tentam descrever o fluxo de dados em APIs de maneira automatizada, até o momento, essas abordagens não conseguiram descrever corretamente todos os fluxos, alcançando aproximadamente 60% nos melhores casos. Finalmente, ao contrário de outros trabalhos, fornecemos uma ferramenta para facilitar o uso de nossa abordagem.

Nas provas de conceito das etapas iniciais deste trabalho, utilizamos o formalismo proposto em (Kallab et al., 2017; Kallab, 2019) como base para a represen-

tação de APIs REST em Redes de Petri Coloridas. Devido, principalmente, ao seu nível de abstração, que, quando comparado com outros modelos, é mais baixo, facilitando a adoção e o entendimento pelos seus usuários, enquanto mantém um nível de formalismo suficiente para fazermos generalizações de casos e ser extensível. No entanto, devido ao enfoque em composição de APIs, a modelagem proposta pelos autores não suporta a descrição de certos tipos de comportamentos de APIs, como condicionais no fluxo de execução, mais especificamente operações de *Ou Exclusivo* (XOR). Tal modelo não prevê que a execução de uma chamada à API pode levar a diferentes fluxos de execução. No mundo real, isso poderia se traduzir em:

- Um *endpoint* que aceita diferentes métodos HTTP que levam a diferentes fluxos de execução. Por exemplo, uma requisição $POST$ a um *endpoint* $/frutas$ pode levar a um fluxo de execução, enquanto uma requisição GET ao mesmo *endpoint* pode levar a outro fluxo de execução.

- Um *endpoint* que gera diferentes *status codes* que levam a diferentes fluxos de execução. Por exemplo, o *status code* 200 de uma requisição $POST$ a um *endpoint* $/login$ pode levar a um fluxo de execução, enquanto o *status code* 401 para o *endpoint* citado pode levar a outro fluxo de execução.

O próximo capítulo detalha as limitações dessa abordagem e como as superamos, propondo uma nova modelagem simplificada que modela apenas as estruturas necessárias para o algoritmo de Verificação de Conformidade.

Capítulo 4

Transformações de Modelos em APIs REST

Este capítulo descreve os algoritmos de transformação de modelos OpenAPI para Redes de Petri Coloridas, as motivações do método proposto para representação, como se dá o processo de transformação de especificações OpenAPI em Redes de Petri Coloridas, limitações e dificuldades encontradas.

4.1 Modelagem de APIs REST através de Redes de Petri

Como discutido na Seção 3, diversos autores investigaram possíveis soluções para a modelagem, representação e processamento das descrições de APIs REST. De acordo com Ivanchikj (2021), usualmente as abordagens podem ser divididas em Linguagem de Domínio Específico e Linguagem Sem Domínio Específico, onde um exemplo da primeira é a linguagem RESTalk (Ivanchikj, 2021) e um exemplo da segunda é a Rede de Petri. A escolha entre uma linguagem e outra depende do problema que se deseja modelar e comumente acompanha a dualidade entre a especificidade da modelagem e sua usabilidade, ou qual a curva de aprendizado aceitável, considerando, por exemplo, o ecossistema de ferramentas e quão difundida é a linguagem.

Considerando a modelagem de APIs REST, as DSLs propostas na literatura foram desenvolvidas para serem capazes de modelar todas as características e comportamentos de uma API. Porém, elas carecem de ecossistemas de ferramentas, como ferramentas de simulação, editores gráfico-textuais e ferramentas de análise automática, além da curva de aprendizado associada a aprender uma nova linguagem. Uma vez que essas linguagens precisam ser adotadas inicialmente por

desenvolvedores de software e profissionais de tecnologia para que a abordagem proposta nesse trabalho seja adotada, este é um ponto negativo considerável. Enquanto isso, as Linguagens Sem Domínio Específico oferecem ricos ecossistemas de ferramentas bem estabelecidas, incluindo ferramentas comerciais, comunidades ativas e facilidade na adoção. Além de já serem potencialmente utilizadas em áreas adjacentes.

Dessa forma, como apresentado na seção anterior, diversos autores já investigaram como representar APIs REST através de Redes de Petri Coloridas (Decker et al., 2009; Li and Chou, 2011; Kallab et al., 2017; Kallab, 2019), demonstrando que essa representação pode ser utilizada para identificar divergências em aplicações do mundo real (Kallab, 2019; Bernardi et al., 2017; Carrasquel et al., 2020), maturidade, viabilidade da abordagem, e que são largamente empregadas na modelagem do fluxo de execução de sistemas, incluindo ordem de execução, sincronização e concorrência.

Dadas essas características, neste trabalho, propomos que as CPNs fornecem uma abstração e modelo capazes de representar APIs REST pelas seguintes razões:

1. As CPNs integram fluxo e fluxo de dados em um único modelo, capturando a natureza dinâmica das interações da API;

2. Elas modelam formalmente sistemas concorrentes, distribuídos e orientados a eventos, o que está alinhado com a natureza das interações das APIs REST, como demonstrado na literatura (Decker et al., 2009; Li and Chou, 2011, 2015);

3. As CPNs suportam algoritmos de verificação de conformidade (Carrasquel et al., 2020), que são essenciais nesse trabalho para verificar comportamentos da API.

4. Sua representação visual é diretamente equivalente à sua definição matemática, permitindo a análise da representação gráfica sem perda de informação (Jensen and Kristensen, 2009). Além disso, caso necessário, é possível aplicar outros algoritmos de transformação de modelo sobre CPNs.

Nas seções seguintes, analisaremos e discutiremos o uso do modelo formal proposto por Kallab (2019) como base para modelar o comportamento e a composição de recursos em APIs REST, uma vez que essa abordagem se demonstrou capaz de modelar fluxos de controle e de dados em aplicações do mundo real. Em seguida, considerando as limitações desse modelo, discutiremos a proposição de um novo modelo com estrutura simplificada e que modela apenas os elementos necessários para os casos de uso propostos neste trabalho.

4.1.1 Modelagem de APIs REST proposta por Kallab

No modelo proposto por Kallab et al. (2017), modela-se um recurso RESTful que faz a composição de outras requisições, não uma API RESTful propriamente dita. Antes de descrever como cada componente da API é mapeado para um componente da CPN, descreveremos alguns conjuntos utilizados nesses mapeamentos:

- $RequestLine = (HTTP \times URL) \times Id$, onde $HTTP$ é um conjunto de *métodos HTTP* (definidos pela já mencionada *RFC 9110*), URL é um conjunto de caminhos relativos e Id é um conjunto de identificadores de usuário exclusivos em formato de string. Exemplos de identificadores são endereços IP, IDs de sessão e tokens.

- $ResponseStatus = (StatusCodes \times Id)$, onde $StatusCodes$ é um conjunto formado pelos *HTTP Status Codes*, definidos pela já mencionada *RFC 9110*, em formato string, ou seja, $StatusCodes = \{200, 201, 401, ...\}$. Id é um conjunto de identificadores de usuário exclusivos em formato de string.

- $DataType = \{null, boolean, object, array, number, string\} \times Id$, em outras palavras, é um conjunto formado pelos seis tipos primitivos definidos pelo já mencionado *JSON Schema Specification Wright Draft* e um conjunto de identificadores únicos de usuário em formato string, denominado Id.

A Rede Colorida de Petri é definida como $CPN = (P,T,A,V,G,E)$, onde:

- Σ é o conjunto de cores da CPN, formado pela união dos conjuntos de tipos descritos no parágrafo anterior, ou seja, $\Sigma \subseteq \{RequestLine \cup ResponseStatus \cup DataType\}$, onde o URL definido em $RequestLine$ é composto pelos caminhos relativos dos *endpoints* da API.

- P é o conjunto finito de lugares com cores associados a um recurso da API. Cada recurso requer ao menos um local de entrada com a cor $RequestLine$, representando a requisição enviada a ele, e um local de saída com a cor $ResponseStatus$, representado a resposta gerada devido a requisição recebida.

- T é o conjunto finito de transições, fornecem representações de recursos. Na prática, pode ser visto como o *path* de um *endpoint* da API.

- A é o conjunto finito de arcos da CPN, conectando lugares com transições de acordo com a estrutura da API. Contém uma função de expressão do arco.

- V é o conjunto finito de variáveis com tipos da CPN, onde $Type(v) \in \Sigma$, para todo $v \in V$.
- G é o conjunto finito de funções de guarda da CPN, expressando condições para que determinada transição se torne ativa.
- E é o conjunto finito de funções de expressão dos arcos da CPN. Podem conter constantes e variáveis $v \in V$, onde atribuindo valores às constantes, calcula-se o valor da expressão.

Considerando, por exemplo, a API descrita na Figura 2.1, a representaríamos como CPN conforme mostrado nas Figuras 4.1 e 4.2 , construída manualmente através da ferramenta CPNTools [1].

Na CPN apresentada na Figura 4.1, P é composta por:

- $Req1$ e $Req2$ com a cor $RequestLine$.
- $Email$, $Password$ e $data$ com a cor String.
- $Status$ com a cor $ResponseStatus$.
- id com a cor número inteiro.
- $authentication$ com a cor $authentication$ representando um objeto.

Na CPN da Figura 4.1, T é composto por 2 transições, $/login$ e $/accounts/id$. Temos A composto por 11 arcos com diferentes expressões. V é composto pelas variáveis $anyReq$, $email$ e $password$, representando as entradas da transição $/login$. As saídas de dados da transição $/login$ são representadas por $status$, $auth$ e $req2$. A variável $anyRequest$ representa a entrada da transição $/accounts/id$ e as variáveis $data$ e $status$ representam a saída da transição $/accounts/id$.

No exemplo da figura, as funções de guarda em G são utilizadas para validar os métodos HTTP das requisições passadas. Finalmente, temos a função de expressão do arco, E, composta por variáveis, como $anyReq$, e $\#id\ auth$, indicando que aquele arco consome um token com conteúdo igual ao atributo id da variável $auth$.

Considerando que se esta CPN recebesse uma requisição para o $endpoint$ $/login$, com o email igual a $email@email.com$ e a senha igual a $Pass123$, teríamos os tokens em verde mostrados na Figura 4.1. Disparando a transição $/login$, teríamos o resultado mostrado na Figura 4.2, mostrando que os tokens de entrada foram consumidos e novos foram criados, representando a resposta da requisição.

Considerando agora as limitações dessa abordagem, apresentadas inicialmente na seção 3.1.3, temos duas que são estruturais e inviabilizam o uso dessa abordagem para modelar APIs reais:

[1] Acessível em `https://cpntools.org/`

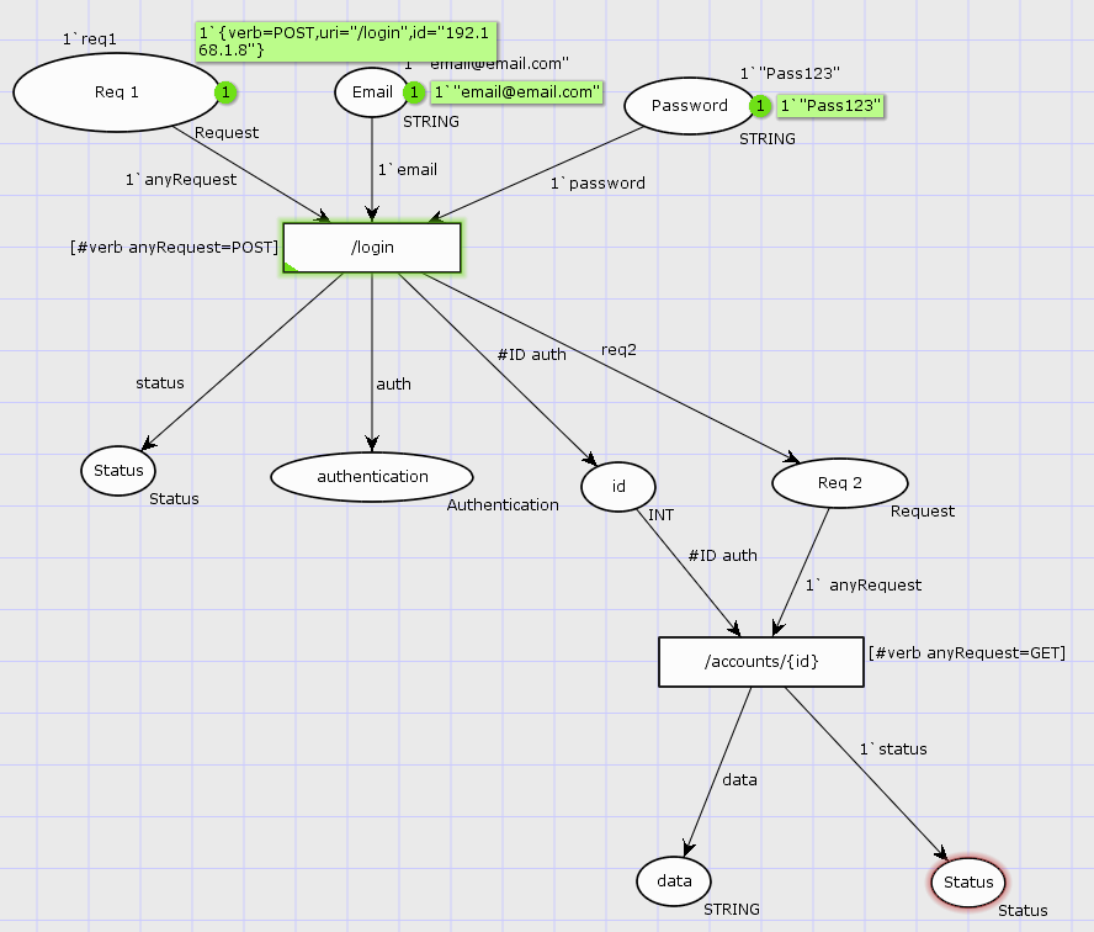


Figura 4.1: Exemplo de aplicação vulnerável a BOLA modelada com CPN na ferramenta CPNTools antes de executar a transição */login*. Círculos representam lugares e retângulos representam transições. Fonte: Elaborada pelo autor.

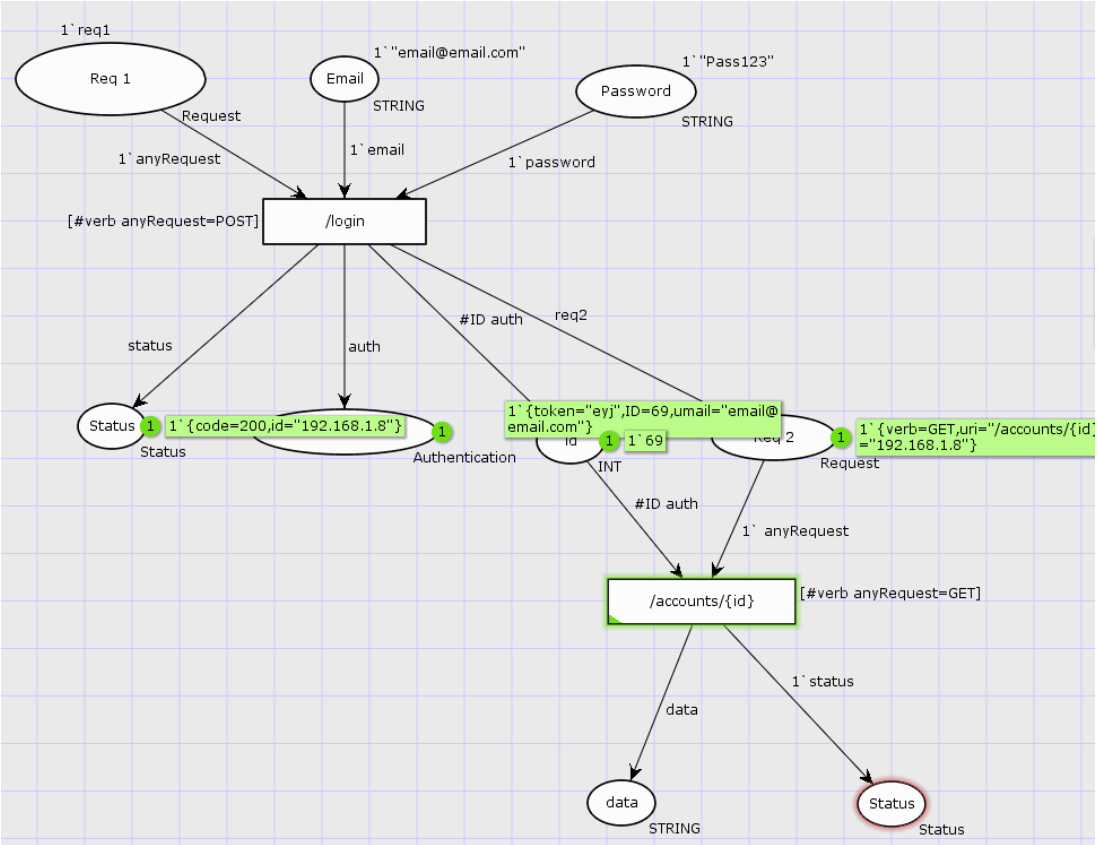


Figura 4.2: Exemplo de aplicação vulnerável a BOLA modelada com CPN na ferramenta CPNTools após executar a transição */login*. Círculos representam lugares e retângulos representam transições. Fonte: Elaborada pelo autor.

i) Um *endpoint* que gera diferentes *status codes* que levam a diferentes fluxos de execução. Por exemplo, o *status code* 200 de uma requisição *POST* a um *endpoint* */login* pode levar a um fluxo de execução, enquanto o *status code* 401 para o endpoint citado pode levar a outro fluxo de execução. As figuras 4.3 e 4.4 ilustram um exemplo onde isso ocorre. Na figura 4.3 é possível ver uma especificação OpenAPI que define um *endpoint* */login* que aceita um método *POST* e dois possíveis *status codes*. Caso ocorra o *status code* 200, a operação possível seguinte é a *getAccountById* (*endpoint* */accounts/id* que não aparece no código) e caso ocorra o *status code* 404, a operação possível seguinte é a *signup* (*endpoint* */signup* que não aparece no código). No entanto, quando analisamos a figura 4.4 que representa a CPN equivalente à especificação, podemos notar que outro comportamento está descrito, que após a chamada ao *endpoint* */login*, pode-se chamar tanto */accounts/id*, quanto */signup* independentemente do *status code* recebido.

ii) Um *endpoint* que aceita diferentes métodos HTTP que levam a diferentes fluxos de execução. Um exemplo desse caso é mostrado na figura 4.5. Há um *endpoint* */login* que aceita os métodos HTTP *POST* e *PUT*. Quando o método HTTP *POST* é usado, a operação possível seguinte é a *getAccountById* (*endpoint* */accounts/id* que não aparece no código). Quando o método HTTP *PUT* é usado, a operação possível seguinte é a *signup* (*endpoint* */signup* que não aparece no código). A CPN equivalente a essa especificação OpenAPI é a mesma mostrada no caso anterior. Como resultado, independentemente do método HTTP utilizado na operação ao *endpoint* */login*, o usuário pode prosseguir sua execução para o *endpoint* */accounts/id* ou o *endpoint* */signup*.

Essas limitações surgem devido ao objetivo do modelo formal proposto pelos autores ser a modelagem da composição de APIs e não das APIs em si. Como apresentado, a composição não prevê caminhos opcionais ou de *Ou Exclusivo* (XOR) nos fluxos de execução da API. Inviabilizando o uso dessa abordagem para casos de uso que exigem mais flexibilidade.

4.1.2 Modelagem de APIs REST proposta neste trabalho

Apesar da robustez do modelo anterior para a composição de serviços, sua aplicação para a modelagem de APIs requer adaptações para lidar com certos comportamentos, apresentados na última subseção. Para superar essas limitações, propomos uma especialização do formalismo que simplifica a representação, focando nos elementos essenciais para a verificação de conformidade e introduzindo a flexibilidade necessária para modelar fluxos de execução alternativos.

De forma similar ao modelo apresentado na seção anterior, antes de descrever como cada componente da API é mapeado para um componente da CPN, descreveremos alguns conjuntos empregados. Primeiramente, a cor utilizada nos tokens

```yaml
/login:
  post:
    tags: ["Log-in"]
    operationId: login
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/LoginUserRequest'
    responses:
      '200':
        description: search results matching criteria
        content:
          application/json:
            schema:
              type: object
              $ref: '#/components/schemas/LoginUserResponse'
        links:
          getBasketById:
            operationId: getAccountById
            parameters:
              id: $response.body#/authentication.id
      '404':
        description: user not found
        content:
          application/json:
            schema:
              type: object
              $ref: '#/components/schemas/LoginUserResponse'
        links:
          goToSignup:
            operationId: signup
```

Figura 4.3: Exemplo de OpenAPI em YAML que não é corretamente modelada seguindo o modelo de (Kallab et al., 2017). Fonte: Elaborada pelo autor.

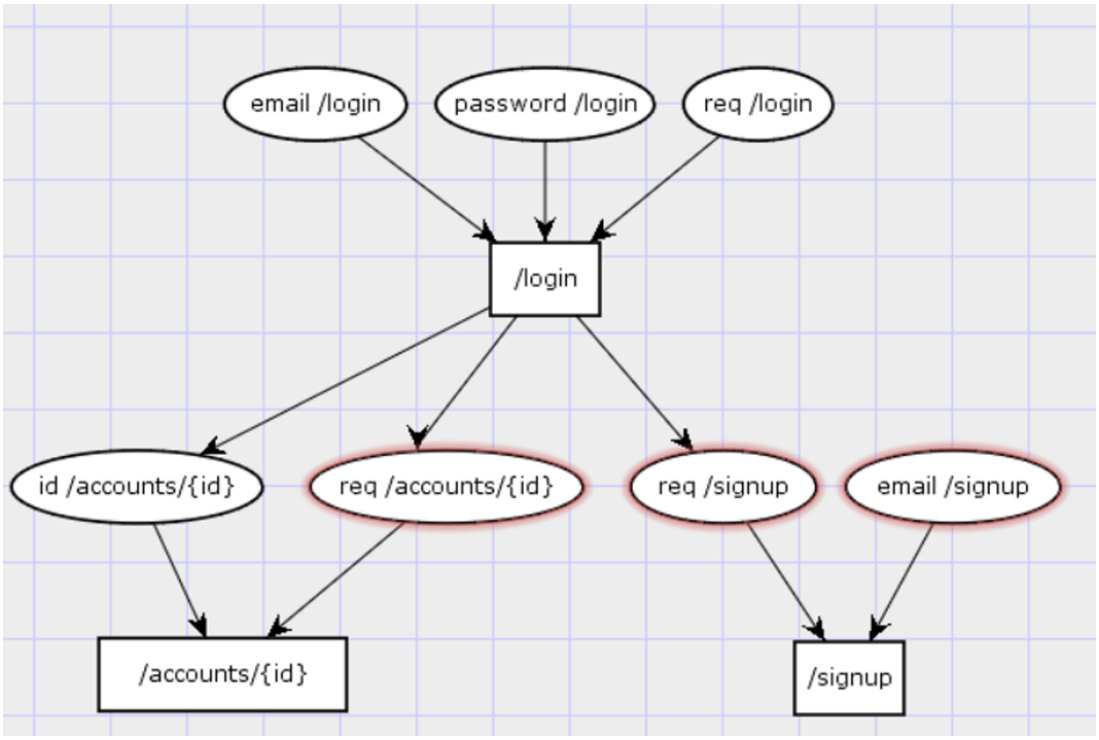


Figura 4.4: Exemplo de CPN que não é corretamente modelada seguindo o modelo de (Kallab et al., 2017) (simplificada para melhorar a legibilidade). Fonte: Elaborada pelo autor.

```yaml
/login:
  post:
    tags: ["Log-in"]
    operationId: login
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/LoginUserRequest'
    responses:
      '200':
        description: search results matching criteria
        content:
          application/json:
            schema:
              type: object
              $ref: '#/components/schemas/LoginUserResponse'
        links:
          getBasketById:
            operationId: getAccountById
            parameters:
              id: $response.body#/authentication.id
  put:
    tags: [ "Log-in" ]
    operationId: getLogin
    responses:
      '200':
        description: search results matching criteria
        content:
          application/json:
            schema:
              type: object
              $ref: '#/components/schemas/LoginUserResponse'
        links:
          goToSignup:
            operationId: signup
```

Figura 4.5: Um exemplo de OpenAPI que não é corretamente modelada seguindo o modelo de (Kallab et al., 2017) onde um endpoint pode levar a dois diferentes endpoints, de acordo com o método HTTP. Seu correspondente em CPN é equivalente ao apresentado na Figura 4.4. Fonte: Elaborada pelo autor.

trata-se do $DataType = \{null, boolean, object, array, number, string\} \times UserId$, em outras palavras, é um conjunto formado pelos seis tipos primitivos definidos pelo já mencionado *JSON Schema Specification Wright Draft* e um conjunto de identificadores únicos de usuário em formato string, denominado $UserId$. Exemplos de identificadores são endereços IP, IDs de sessão e tokens.

A Rede Colorida de Petri é definida como $CPN = (P,T,A,V,G,E)$, onde:

- Σ é o conjunto de cores da CPN, formado pelo tipo descrito no parágrafo anterior, ou seja, $\Sigma \subseteq \{DataType\}$.
- P é o conjunto finito de lugares com cores associados a representação de um recurso da API.
- T é o conjunto finito de transições, representam uma única operação da API sobre um *path* e sua resposta. Na prática, pode ser visto como uma tríade formada pelo *método HTTP*, o *path* de um *endpoint* da API e um *HTTP Status Codes*, descrevendo assim uma única operação da API e seu resultado.
- A é o conjunto finito de arcos da CPN, conectando lugares com transições de acordo com a estrutura da API. Contém uma função de expressão do arco.
- V é o conjunto finito de variáveis com tipos da CPN, onde $Type(v) \in \Sigma$, para todo $v \in V$.
- G é o conjunto finito de funções de guarda da CPN, expressando condições para que determinada transição se torne ativa.
- E é o conjunto finito de expressões de arco da CPN. Podem conter constantes e variáveis $v \in V$, onde atribuindo valores às constantes, calcula-se o valor da expressão.

Considerando, por exemplo, a API descrita na Figura 2.1, a representaríamos como CPN conforme mostrado na Figura 4.6, construída manualmente através da ferramenta CPNTools.

Na CPN da Figura 4.6, T é composto por 2 transições, $POST/login200$ e $GET/accounts/id200$. Temos A composto por 2 arcos com 2 expressões diferentes. V é composto pelas variáveis $authentication$ e $id\_in\_request2$, a saída da primeira transição e a entrada da segunda transição, respectivamente.

No exemplo da figura, as funções de guarda em G não estão sendo utilizadas, assumindo assim o valor de *Verdadeiro* sempre. Finalmente, temos a função de expressão do arco, E, composta pelas variáveis $authentication$ e $id\_in\_request2$,

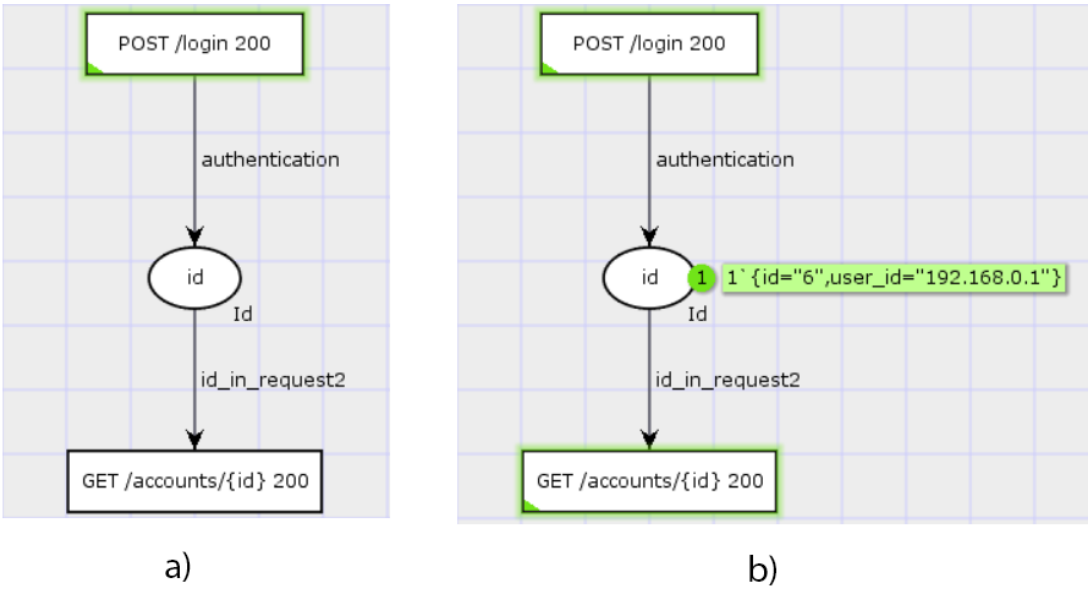


Figura 4.6: Exemplo de aplicação vulnerável a BOLA modelada com CPN na ferramenta CPNTools. a) antes de executar a transição */login* e b) após executar a transição , mostrando um token no lugar id. Fonte: Elaborada pelo autor.

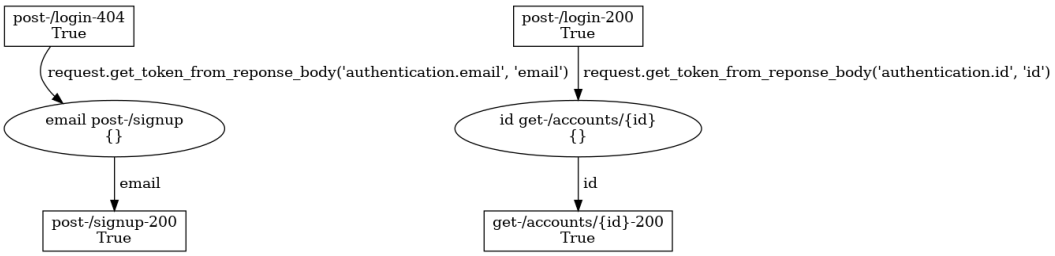
indicando que aquele arco consome um token com conteúdo igual ao atributo àquelas variáveis.

Considerando que a API representada pela CPN da Figura 4.6 a) recebesse uma requisição válida para o *endpoint /login*, resultando em uma resposta com *status code* 200 e um objeto *authentication*, teríamos como resultado a CPN exibida na Figura 4.6 b), onde o lugar *id* possui um token (mostrado em verde na figura), demonstrando que a transição *GET /accounts/{id} 200* tornou-se disparável para uma requisição com *id* igual a 6 e *user_id* igual a *192.168.0.1*.

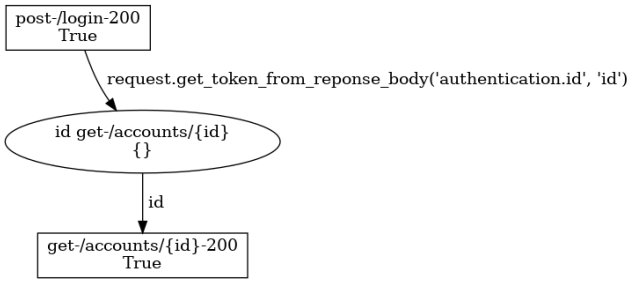
Analisando as Figuras 4.6 e 4.5 que representam a mesma API, é possível notar que a representação proposta nesse trabalho é mais sucinta, modelando o mínimo de informações necessárias para a verificação do fluxo de dados entre endpoints. Além disso, a modelagem proposta é capaz de representar os casos que Kallab (2019) não era, como i) Um endpoint que gera diferentes status codes que levam a diferentes fluxos de execução; e ii) Um endpoint que aceita diferentes métodos HTTP que levam a diferentes fluxos de execução.

Esses dois casos são apresentados na Figura 4.7.

Na Figura 4.7a é possível ver o cenário onde um endpoint gera diferentes status codes (200 e 400) que levam a diferentes fluxos de execução (*post-/signup-200* e *get-/accounts/{id}-200*). Na Figura 4.7b é possível ver o cenário onde um endpoint aceita diferentes métodos HTTP (*POST* e *PUT*) que levam a diferentes fluxos



(a) CPN de um endpoint que gera diferentes status codes que levam a diferentes fluxos de execução; equivalente a Figura 4.4, mas representada com a nova modelagem.



(b) CPN de um endpoint que aceita diferentes métodos HTTP que levam a diferentes fluxos de execução; equivalente a Figura 4.5, mas representada com a nova modelagem.

Figura 4.7: Representação na nova modelagem dos casos não corretamente representados na Seção 4.1.1.

de execução. Nesse caso, a representação foi simplificada eliminando totalmente a parte relacionada ao *put-/login-200*, isso porque não existe fluxo de dados dessa operação para a seguinte, eliminando partes da CPN que não agregam para a análise do fluxo de dados. A representação de operações por meio da tupla método HTTP + URL + *status code* foi apresentada como solução para situações como a i) e ii) em (Ivanchikj, 2021) que foi o que nos levou a adotá-la nesse trabalho.

Dessa forma, pode-se considerar que, para compor o modelo, são utilizadas as seguintes informações:

- URL.
- Parâmetros da URL.
- Método HTTP.
- *Status code* da resposta.
- Cabeçalhos HTTP e/ou IP do cliente (usado para para identificar o usuário, no exemplo, é o valor do campo *user_id*).
- Corpo da requisição.
- Corpo da resposta.

4.2 Transformando Especificações OpenAPI em Redes de Petri

A seção anterior apresentou os elementos disponíveis para construir uma CPN representando uma API REST. Esta seção demonstrará como representações de APIs REST em OAS (*OpenAPI Specification*) podem ser transformadas em representações em Redes de Petri Coloridas. Inicialmente, apresentaremos os passos para realizar a transformação de modelo de um documento OAS para uma CPN. Em seguida, será introduzido, como exemplo, o documento OAS correspondente à API discutida na seção anterior na Listagem 4.1 e como os passos da transformação se relacionam ao modelo.

O Algoritmo 1 ilustra o processo de criação de uma CPN a partir de uma especificação OpenAPI, onde a entrada é uma especificação OpenAPI *doc* e a saída é uma Rede de Petri Colorida $\mathcal{C}$, formada pela tupla (P, T, A, V, G, E) definida na Seção 2.4.

No algoritmo, nomes em *camel case*, como *responses* e *propertyKey* são usados para nomes de atributos da especificação OpenAPI e variáveis do algoritmo, enquanto nomes em *pascal case*, como *PathItemObject* e *Response* são usados

Algoritmo 1 Criação de CPN a partir de documento OpenAPI.

Input: *doc*, uma especificação OpenAPI
Output: $\mathcal{C}$, uma rede de Petri colorida

$\mathcal{C} \leftarrow createEmptyCPN()$;
foreach $(path, PathItemObject\ \mathcal{P}) \in doc.paths$ **do**
  foreach $(httpMethod, OperationObject\ \mathcal{O}) \in \mathcal{P}$ **do**
    foreach $(httpStatusCode, ResponseObject) \in \mathcal{O}.responses$ **do**
      $transtion \leftarrow createTransition(path, httpMethod, httpStatusCode)$;
      $CPN.addToCPN(transtion)$
      $requestBody \leftarrow \mathcal{O}.requestBody$
      foreach $(mediaType, MediaTypeObject\ \mathcal{M}) \in requestBody.content$ **do**
        $propertiesList \leftarrow \mathcal{M}.schema.properties$
        foreach $(propertyKey, propertyValue) \in propertiesList$ **do**
          if $isTargetedByLink(propertyKey, \mathcal{O})$ **then**
            $place \leftarrow createPlace(\mathcal{O}.RequestBodyObject, transition)$;
            $\mathcal{C}.addToCPN(place)$;
            $\mathcal{C}.createArc(transition, place)$;
          end if
        end for
      end for
      foreach $ParameterObject \in \mathcal{O}.parameters$ **do**
        if $isTargetedByLink(ParameterObject)$ **then**
          $place \leftarrow createPlace(ParameterObject, transition, \mathcal{O})$;
          $\mathcal{C}.addToCPN(place)$;
          $\mathcal{C}.createArc(transition, place)$;
        end if
      end for
    end for
  end for
end for
$createLinkArcs(doc.paths, \mathcal{C})$
$removeDisconnectedTransitions(\mathcal{C})$
return $\mathcal{C}$;

para nomes de objetos da especificação OpenAPI, comumente acompanhados de uma versão resumida, como $\mathcal{P}$ e $\mathcal{O}$, para deixar o algoritmo mais sucinto.

Inicialmente, cria-se uma CPN vazia (linha 1). Em seguida, acessa-se o objeto *PathsObject* contido no atributo *paths* da especificação *doc*, formado por um conjunto de pares chave-valor, onde a chave é um *path* e o valor é um *PathItemObject* (linha 2). Iteramos sobre cada um dos *PathItemObject*, obtendo um conjunto de pares chave-valor, onde a chave é um método HTTP $httpMethod$ e o valor é um *OperationObject* (linha 3). Itera-se sobre o atributo *responses* do objeto *OperationObject*, obtendo pares chave-valor, onde a chave é um código de status HTTP *httpStatusCode* e o valor é um *Response Object* (linha 4). Criamos, então, uma nova transição $t \in T$ formada pela tríade *(path, httpMethod, httpStatusCode)* (linha 5) e a adicionamos na CPN (linha 6).

Acessamos o parâmetro *requestBody* do objeto *OperationObject* e atribuímos à variável *requestBody* (linha 7). Iteramos sobre o atributo *content* de *requestBody*, obtendo pares chave-valor formados por uma string *mediaType* e um *MediaTypeObject* (linha 8). Atribuímos à variável *propertiesList* o conteúdo do atributo *properties* do atributo *schema* do *MediaTypeObject* (linha 9). Iteramos sobre o *propertiesList*, obtendo pares chave-valor, onde a chave é uma string e o valor é um dos tipos primitivos definidos no *JSON Schema Specification Wright Draft* (linha 10). De posse do *propertyKey* e do *OperationObject*, podemos verificar se essa propriedade está envolvida com algum link da especificação OpenAPI (linha 11). Caso esteja, criamos um lugar $p \in P$ com uma cor definida pelo *SchemaObject* do *OperationObject* (linha 12), o adicionamos a CPN (linha 13) e o conectamos à transição recém-criada, através de um arco-PT $a \in A$ (linha 14).

A partir daí, acessamos o atributo *parameters* do *OperationObject*, obtendo uma lista de *ParameterObjects* e iteramos sobre ela (linha 18). Para cada *ParameterObject*, verificamos se ele está envolvido com algum link da especificação OpenAPI (linha 19), em caso positivo, criamos um lugar $p \in P$ com uma cor definida pelo *SchemaObject* do *OperationObject* (linha 20), o adicionamos a CPN (linha 21) e o conectamos à transição recém-criada, através de um arco-PT $a \in A$ (linha (22)).

Após fecharmos todos os laços de repetição, analisamos todos os links definidos na especificação OpenAPI e criamos os arcos $a \in A$ que representam as conexões dos links (linha 28). Por fim, removemos todas as transições que não possuem conexões com lugares (linha 29) e retornamos a CPN (linha 30).

A Listagem 4.1[2] é a representação em OAS da API discutida na seção 4. Como é possível observar, mesmo descrevendo apenas 2 *endpoints*, a documentação da API pode chegar próxima a ter 100 linhas.

[2]Acessível em `https://github.com/ailton07/openapi-links-to-CPNs/blob/main/examples/BOLA_Example_login_and_accounts_id.yaml`

O objeto *PathsObject* utilizado na linha 2 dos passos de transformação de modelo está representado pelas linhas 19 a 58 da OAS. Os objetos *PathItemObject* são representados pelas linhas 20 a 27 e 42 a 50. Os objetos *OperationObject* são representados pelas linhas 21 a 27 e 43 a 50. Os objetos *ResponsesObject* estão representados pelas linhas 29 a 35 e 52 a 58. O *ParameterObject*, utilizado na linha 18, é representado pelas linhas 46 a 50 do documento OpenAPI. Os objetos *SchemaObject* são representados pelas linhas 27, 34 a 35, 50 e 57 a 58. Por fim, o *RequestBodyObject* utilizado na linha 12 do algoritmo é representado pelas linhas 24 a 27 do documento OpenAPI.

Listagem 4.1: YAML representando uma API em documento OpenAPI.

```
1  openapi: 3.1.0
2  servers:
3    - description: SwaggerHub JuiceShop API
4      url: https://virtserver.swaggerhub.com/ailton07/JuiceShop/1.0.0
5  info:
6    description: BOLA Example API Description.
7    version: "1.0.0"
8    title: BOLA Example API Description
9    contact:
10     email: you@your-company.com
11   license:
12     name: Apache 2.0
13     url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
14 tags:
15   - name: Log-in
16     description: Log-in and get User
17 paths:
18   /login:
19     post:
20       tags: ["Log-in"]
21       operationId: login
22       requestBody:
23         content:
24           application/json:
25             schema:
26               $ref: '#/components/schemas/LoginUserRequest'
27       responses:
28         '200':
29           description: search results matching criteria
30           content:
31             application/json:
32               schema:
33                 type: object
34                 $ref: '#/components/schemas/LoginUserResponse'
35           links:
36             getBasketById:
```

```
37                operationId: getAccountById
38                parameters:
39                  id: $response.body#/authentication.id
40    /accounts/{id}:
41      get:
42        tags: ["Log-in"]
43        operationId: getAccountById
44        parameters:
45        - name: id
46          in: path
47          required: true
48          schema:
49            type: string
50        responses:
51          '200':
52            description: search results matching criteria
53            content:
54              application/json:
55                schema:
56                  type: object
57                  $ref: '#/components/schemas/GetAccountByIdResponse'
58  components:
59    schemas:
60      LoginUserRequest:
61        type: object
62        properties:
63          email:
64            type: string
65          password:
66            type: string
67      LoginUserResponse:
68        type: object
69        properties:
70          authentication:
71            type: object
72            properties:
73              token:
74                type: string
75                example: "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
      eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
      .SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c"
76              id:
77                type: integer
78                example: 69
79              umail:
80                type: integer
81                example: email@email.com
82      GetAccountByIdResponse:
83        type: object
```

```
84          properties:
85            status:
86              type: string
87            data:
88              type: object
89              properties:
90                id:
91                  type: integer
92                  example: 69
93                name:
94                  type: string
95                  example: "I'm a User"
```

Analisando a Listagem 4.1, pode-se notar que a API possui dois *endpoints*: */login* e */accounts/*{*id*}. Onde o primeiro aceita o método HTTP *POST* e o segundo aceita o método HTTP *GET*. Podemos ver também que o primeiro *endpoint* pode ter uma resposta com *status code* 200 e há um link entre o primeiro e o segundo *endpoint*. Esse link descreve que a saída (resposta) *authentication.id* do primeiro *endpoint* pode ser usada para compor a requisição ao segundo *endpoint*, se tornando o parâmetro *id*.

Na Figura 4.8 temos a aplicação do Algoritmo 1 na especificação OpenAPI descrita na Listagem 4.1.

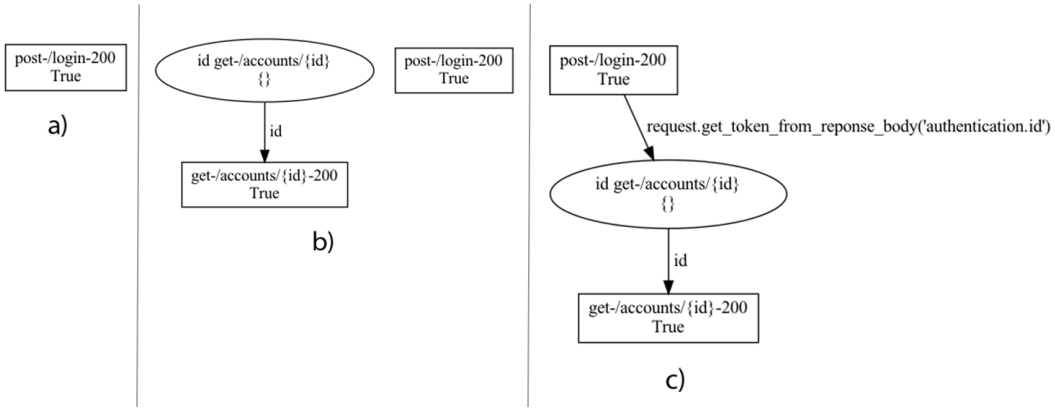


Figura 4.8: CPN obtida a partir da transformação da especificação OpenAPI da Listagem 4.1.

A primeira parte da figura, *a)* mostra o resultado da primeira iteração do algoritmo (linhas 1 a 27) que é sobre as linhas 19 a 40 da especificação OpenAPI, gerando como resultado uma CPN que possui apenas uma transição. A parte *b)* da figura descreve a segunda iteração que ocorre sobre as linhas 41 a 58 da especificação OpenAPI, gerando uma CPN com duas transições e um lugar. A última parte da figura, *c)*, ilustra o resultado da última parte do algoritmo (linhas

28 a 30) que gera como resultado uma CPN com duas transições e um lugar, mas agora com arcos os conectando. A CPN retornada ao final indica que a operação *POST* sobre o endpoint */login* não requer que outras chamadas tenham sido feitas anteriormente e que quando tem como resposta o *status code* 200, ela gera uma informação que é usada pelo endpoint */accounts/*{*id*}. A operação *GET* sobre o endpoint */accounts/*{*id*}, por sua vez, requer uma informação proveniente do endpoint */login* e gera uma resposta que não é utilizada por nenhum outro endpoint da API.

4.3 Conclusão do Capítulo

Neste capítulo foi discutido o modelo formal proposto em (Kallab et al., 2017; Kallab, 2019) para a representação de APIs REST em CPNs e suas limitações. Em seguida, foi introduzido um novo modelo de representação em CPNs, com o objetivo de representar as APIs de forma sucinta, incluindo os casos em que não era possível anteriormente. Em seguida, foi apresentado um algoritmo de transformação modelo para modelo (*model-to-model transformation*), onde a representação em documento OAS de uma API REST é convertida em uma representação em Redes de Petri Colorida, incluindo um exemplo, onde apresentamos passo a passo o processo de transformação. O próximo capítulo demonstrará como o modelo resultante das transformações discutidas neste capítulo pode ser utilizado para a detecção de divergências entre o modelo de uma aplicação e seu comportamento real.

Capítulo 5

Verificação de Conformidade em CPNs de APIs REST

Este capítulo descreve o método proposto para detecção de divergências entre uma Rede de Petri Colorida, representando o comportamento esperado de uma API REST e o comportamento observado durante a operação da API no mundo real, processo também chamado de verificação de conformidade. Contudo, antes de apresentá-lo, é necessário que sejam feitas algumas definições para o funcionamento da proposta.

A Verificação de Conformidade é um tema da Mineração de Processos (*Process Mining*) que se refere à análise da relação entre o comportamento esperado de um processo e o comportamento registrado observado durante sua execução. Na prática, a Verificação de Conformidade consiste em uma família de técnicas e algoritmos que relacionam duas principais entradas: modelos de processos e registros de eventos (*event logs*), fornecendo métodos para comparar e analisar instâncias observadas de um processo em relação ao seu modelo (Van der Aalst, 2016; Carmona et al., 2018). Um exemplo de comparação é verificar se um processo está sendo executado conforme documentado em seu modelo.

Dentre os formalismos existentes em Verificação de Conformidade, um dos mais usados é chamado *token replay*, onde logs de eventos são "reproduzidos" sobre um modelo de processo, executando um *replay* das tarefas de acordo com a ordem dos eventos. Ao observar os estados do modelo de processo durante a reprodução, pode-se determinar se, e quanto, os eventos realmente correspondem a uma sequência de execução válida do modelo. Apesar de sua simplicidade, algoritmos de *token replay* tornaram-se um padrão não apenas para verificação de conformidade, mas também para mineração de decisões, análise de desempenho e outras áreas (Carmona et al., 2018).

Em (Carrasquel et al., 2020), os autores apresentaram um algoritmo de Verifi-

cação de Conformidade em CPNs através de *token replay* que é capaz de identificar desvios no fluxo de controle e no de dados devido a recursos indisponíveis, violações de regras e diferenças entre recursos modelados e reais, ao reproduzir log de eventos em cima de CPNs. Como temos um CPN modelado a partir de uma especificação OpenAPI e um conjunto de solicitações e respostas HTTP (ou seja, logs de eventos), adotaremos esse algoritmo neste trabalho e descreveremos nas seções seguintes seu funcionamento e como o adotamos.

5.1 Método Proposto

A Figura 5.1 descreve o funcionamento da abordagem proposta para detectar divergências entre um modelo em CPN e um conjunto de registros de eventos, e, posteriormente, classificar essas divergências.

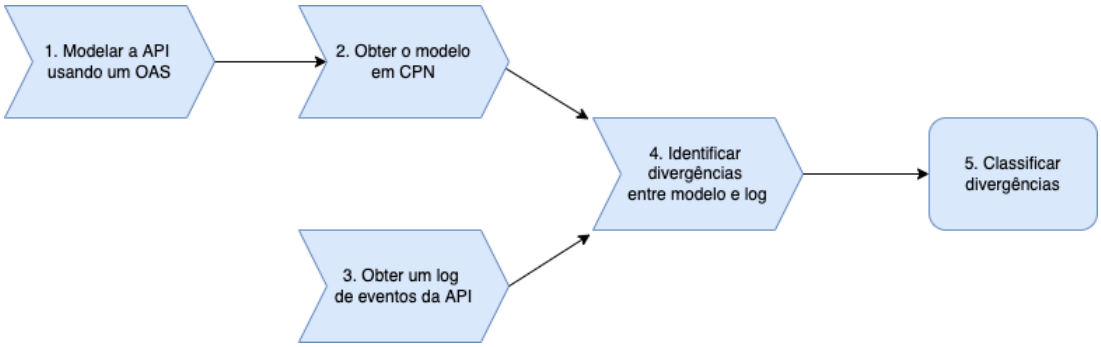


Figura 5.1: Método proposto para detecção de ataque a BLV

.

Inicialmente, em 1, obtém-se um documento representando uma especificações OpenAPI descrevendo uma API. Em 2, aplica-se o algoritmo de transformação de modelo descrito na Seção 4.2, a fim de obter a CPN correspondente à API. Em 3, obtém-se um conjunto de logs representando as comunicações realizadas entre clientes e a API (requisições e respostas) em determinado período de tempo. Note que esse passo pode ser conduzido de forma independente aos passos 1 e 2. Em 4, aplica-se o algoritmo de *token replay* sobre a CPN para identificar divergências entre eventos e o modelo. Por último, em 5, as divergências são classificadas e analisadas manualmente por um analista, a fim de confirmar a existência de ataques.

O pseudo-código do *token replay* sobre CPNs, extraído de (Carrasquel et al., 2020) e traduzido para português, está listado no Algoritmo 2, onde as entradas são uma CPN com a marcação inicial vazia, um log de eventos e um conjunto de lugares iniciais $P_0 \in P$.

Algoritmo 2 Verificação de Conformidade em CPNs.

Input: $\mathcal{C}$, uma rede de Petri colorida
P_0, um conjunto não vazio com os lugares iniciais
L, log finito de eventos
Output: L_{error}, conjunto de eventos divergentes
$fitness$, comportamentos observados permitidos pelo modelo

1: $L_{error} \leftarrow \emptyset$, $fitness \leftarrow 0$
2: **foreach** $\sigma \in L$ **do**
3: $M \leftarrow populaLugaresIniciais(P_0, R(\sigma))$;
4: **foreach** $e = (a, R(e)) \in \sigma$ **do**
5: $t \leftarrow selecionaTransicao(a)$;
6: **if** $divergenciaFluxoControle({}^\bullet t, M, R(e))$ **then** $add(\sigma, L_{error})$; **break**;
7: **end if**
8: $b \leftarrow selecionaBinding(t, M, R(e))$;
9: **if** $violacaoRegra(t, M, b)$ **then** $add(\sigma, L_{error})$; **break**;
10: **end if**
11: $M \leftarrow fire(t, M, B)$;
12: **if** $recursoCorrompido(t^\bullet, M, R(e))$ **then** $add(\sigma, L_{error})$; **break**;
13: **end if**
14: **end for**
15: **end for**
16: $fitness \leftarrow 1 - (|L_{error}/|L||)$;
17: **return** $(L_{error}, fitness)$;

Em um log de eventos L, um *trace* de eventos (σ) em L é composto por uma lista de tuplas formadas por $(e, a, R(e))$, onde e é o evento registrado, a é a atividade associada ao evento e $R(e)$ são os recursos associados ao evento.

Ao iniciar o processamento, para cada *trace* de eventos (σ), os lugares definidos em P_0 são populados com tokens, de acordo com os recursos definidos no *trace* ($R(\sigma)$) e suas cores, através da função $populaLugaresIniciais$. Em seguida, inicia-se o *replay* de σ na CPN. Para cada evento $e = (a,R(e)) \in \sigma$, tenta-se disparar uma transição t associada. Antes de executar a transição, dado o estado atual de marcações da CPN, é verificado se cada recurso envolvido no evento em questão está em um lugar de entrada da transição, realizado pela função $divergenciaFluxoControle$. Se essa função retornar Falso, é possível selecionar um *binding* na CPN, através da função $selecionaBinding$, em outras palavras, as variáveis envolvidas na transição têm um valor calculado com sucesso e os tokens que serão consumidos são selecionados corretamente.

A função $violacaoRegra$ verifica para cada lugar de entrada p da transição t, se os tokens selecionados pelo *binding* b violam alguma das regras expressas através da função de guarda da transição e das funções de expressão dos arcos. Se não há violação, a transição é disparada, consumindo os tokens dos lugares de entrada da transição e produzindo tokens nos lugares de saída, de acordo com a definição nas funções de expressão dos arcos.

A função $recursoCorrompido$ compara os estados observados dos recursos no modelo após a execução da transição, com os estados dos recursos descritos no log, explorando o fato de que cada evento no log possui informações sobre o novo estado dos recursos após a execução da atividade a. Considerando que o disparo de uma transição t, relacionada a um *binding* selecionado b, gera uma nova marcação M na CPN, é possível verificar se cada recurso no sistema, após a execução do evento, foi modificado conforme realizado no modelo, após o disparo da transição.

Por fim, é calculado o *fitness* que é retornado como resultado, junto com o conjunto de *traces* que apresentaram divergências com o modelo.

5.2 Implementação

A fim de implementar o método proposto, de forma similar à implementação de Carrasquel et al. (2020), utilizamos a biblioteca Python *SNAKES*[1]. Essa biblioteca facilita a prototipagem de redes de Petri de alto nível, como CPNs. Utilizando-a, é possível instanciar modelos CPN como objetos Python, que podem ser usados no método descrito na Figura 5.1.

[1] Acessível em `https://snakes.ibisc.univ-evry.fr/`

As Figuras 5.2, 5.3 a) e b) ilustram a prova de conceito desenvolvida [2] utilizando a biblioteca *SNAKES*. Nela, modelamos de forma automatizada a API através do algoritmo descrito na seção anterior, usando como entrada a especificação OpenAPI mostrada na Listagem 4.1, e realizamos a execução da CPN sobre logs reais da execução da API. A Figura 5.2 ilustra o passo 2 do método proposto, mostrado na Figura 5.1), onde a CPN é obtida através da transformação automatizada de modelo, OAS para CPN. Pode-se observar a semelhança do modelo gerado com a modelagem manual mostrada na Figura 4.6. Nesse cenário, se durante o *token replay*, tenta-se disparar a transição *get-/accounts/id-200*, teríamos um caso de violação de fluxo de controle, uma vez que não podemos executar essa transição sem ter executado a transição *post-/login-200* anteriormente. Na implementação da prova de conceito, esse comportamento geraria uma exceção e uma mensagem informando o erro.

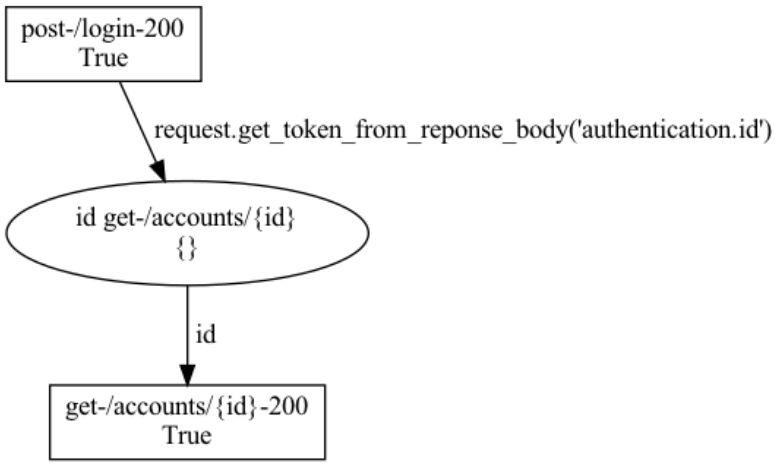


Figura 5.2: API modelada em CPN através da prova de conceito desenvolvida.

Em seguida, o lado *a)* da Figura 5.3 demonstra um cenário onde está sendo processado um evento do log de eventos da API que registrou o recebimento de uma requisição ao endpoint */login*. Essa figura mostra o estado da CPN imediatamente após o disparo da transição *post-/login-200*, primeira iteração do Algoritmo 2. Pode-se observar que temos o número 6 como conteúdo do lugar *id get-/accounts/id*, além da identificação do usuário que realizou a requisição.

Por fim, o lado *b)* da Figura 5.3 demonstra o estado da CPN após o disparo da segunda transição, associada ao endpoint */accounts/{id}*, onde os tokens dos lugares de entrada foram consumidos com sucesso (no mundo real, não removería-

[2] Acessível em `https://github.com/ailton07/openapi-links-to-CPNs`

mos o token do lugar de entrada, removemos aqui para facilitar a visualização), segunda iteração do Algoritmo 2. Se a transição *get-/accounts/id-200* fosse disparada com um valor de *id* diferente de 6, teríamos uma violação do fluxo de dados. Na prova de conceito implementada, esse comportamento geraria uma exceção e uma mensagem informando o erro. Essas figuras demonstram o funcionamento do algoritmo de *token replay*.

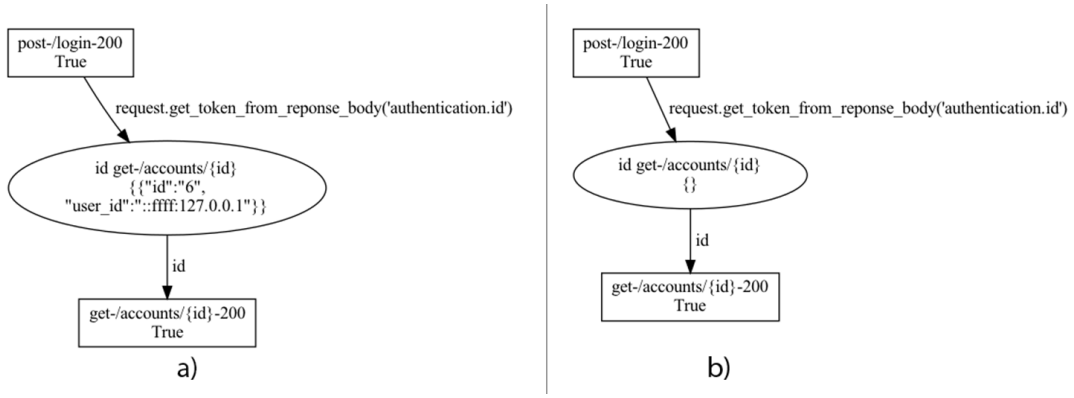


Figura 5.3: a) CPN após a execução da primeira transição; b) CPN após a execução da segunda transição.

5.3 Geração dos Logs de Eventos

Conforme discutido nas últimas seções, para a realização da verificação de conformidade e, portanto, execução do método proposto, são utilizadas uma série de informações que compõem os registros de eventos (logs de eventos), sendo elas:

- URL.
- Parâmetros da URL.
- Método HTTP.
- *Status code* da resposta.
- Cabeçalhos HTTP de autorização e/ou IP do cliente.
- Corpo da requisição.
- Corpo da resposta.

A URL, seus parâmetros e o método HTTP da requisição podem ser obtidos do elemento da requisição conhecido como *Request-Line* [3]. Esses dados, em adição ao *Status code* da resposta, são informações utilizadas para compor os lugares e as transições da CPN. Os cabeçalhos HTTP e o IP do cliente são utilizados para criar a informação *UserId* do modelo, usada para identificar o cliente da requisição. O corpo da requisição e o corpo da resposta são utilizados na criação dos tokens.

As informações envolvidas nesta etapa de processamento podem ser obtidas de qualquer servidor web, como *Apache* ou *Nginx* e apenas pequenas adaptações são necessárias para a geração do log de eventos. Por exemplo, utilizando como base dos logs o *Common Logfile Format* (CLF)[4], são necessárias apenas a adição do cabeçalho de autorização, bem como os corpos de solicitação e resposta.

Nos testes que realizamos, para facilitar a execução de operações programáticas, utilizamos o modelo de logs estruturados em JSON, onde a Listagem 5.1 exibe um exemplo. O campo *timestamp* indica a data e hora em que a requisição foi recebida pelo servidor Web. *ip* representa o endereço IP do cliente que enviou a requisição. *message* contém a combinação das informações método HTTP, *URL*, *status code* e o tempo para geração do log. O campo *method* assinala o método HTTP usado na requisição. *uri* assinala o *path* da requisição. *requestBody* apresenta o conteúdo do corpo da requisição, enquanto *responseBody* apresenta o corpo da resposta à requisição. *statusCode* apresenta o *status code* HTTP da resposta à requisição. E, finalmente, *headerAuthorization* apresenta o conteúdo do cabeçalho *Authorization* da requisição.

```
1  {
2     "timestamp":"2022-11-01T22:35:33.107Z",
3     "ip":"::1",
4     "message":"GET /rest/user/whoami 200 4ms",
5     "method":"GET",
6     "uri":"/rest/user/whoami",
7     "requestBody":{
8
9     },
10    "responseBody":{
11       "user":{
12
13       }
14    },
15    "statusCode":200,
16    "headerAuthorization":"Bearer
      eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9..."
```

[3] https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html
[4] https://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format

```
17 }
```

Listagem 5.1: Exemplo de log em JSON.

Registrar cada solicitação HTTP recebida é uma prática comum em servidores web, como Apache, que usa o CLF por padrão. Nossa abordagem estende o CLF registrando o corpo da requisição e o corpo da resposta e, embora isso adicione alguma sobrecarga (*overhead*), ela pode ser mitigada através de técnicas de gerenciamento de logs, como registro assíncrono ou API gateways com recursos de registro integrados. Assim, embora haja alguma sobrecarga, ela é consistente com as práticas típicas de registro de eventos e pode ser gerenciada de forma eficiente. Além disso, quando se considera as melhorias de segurança que esse registro oferece, o *overhead* é gerenciável na maioria dos casos.

5.4 Análise de Divergências

De acordo com Van der Aalst (2016), a interpretação das não conformidades depende da finalidade do modelo, se é descritivo ou normativo. Neste caso, temos um modelo normativo, onde discrepâncias entre o modelo e o comportamento observado podem significar uma de três coisas:

1. O modelo está desatualizado em relação a API em execução;
2. Desvio desejável, ou seja, um desvio com o objetivo de melhorar a operação sendo executada. Em outras palavras, o comportamento não conforme é benigno e legítimo;
3. Desvio indesejável, ou seja, um comportamento que não deveria estar sendo realizado e não deveria ser permitido. Em outras palavras, o comportamento não conforme não é legítimo.

Quando identificadas divergências através do modelo proposto neste trabalho, um analista deve verificar em qual dos itens a divergência se encaixa. Caso seja no terceiro (Desvio indesejável), ainda existem duas possibilidades:

1. Disparo de uma transição falhou porque o usuário não possuía tokens nos lugares pré-requisitos da transição;
2. Disparo de uma transição falhou porque os lugares pré-requisitos não possuíam tokens do usuário que atendessem a expressão do arco da transição.

Ambos os casos estão relacionados a Vulnerabilidade Lógicas, discutidas na Seção 2.2. O primeiro caso está relacionado a *Desvio do Fluxo da Aplicação*, enquanto o segundo caso está relacionado a *Adulteração de Parâmetros*. Os dois

casos podem representar um ataque a uma vulnerabilidade BOLA, por isso, um analista deve confirmar se o usuário realizando a requisição deveria ser capaz de fazê-la.

5.5 Conclusão do Capítulo

Nesse capítulo, foi descrita a metodologia proposta para a verificação de conformidade e detecção de divergências entre um modelo em Rede de Petri Colorida e os comportamentos descritos pela API REST, através dos tópicos Método Proposto, Implementação, Geração de Logs e Análise de Divergências.

Inicialmente, apresentou-se a abordagem proposta de detecção de divergências, descrita em 5 etapas na Figura 5.1, e um pseudo-código representando o algoritmo de verificação de conformidade em CPNs. Após isso, discutiu-se a implementação do método proposto utilizando a biblioteca Python *SNAKES* que resultou em uma prova de conceito disponível online [5], demonstrando a viabilidade técnica da metodologia, ilustrando como a CPN gerada automaticamente pode ser utilizada para simular e validar sequências de requisições reais. Além disso, com fins ilustrativos, a prova de conceito foi aplicada sobre a API de exemplo discutida anteriormente na Listagem 4.1 e um log de eventos, resultando nas Figuras 5.2, 5.3 a) e b). Esse exemplo evidencia como falhas no token replay podem corresponder diretamente a cenários de desvios de fluxo de controle ou violações de regras de dados, que são indicadores potenciais de exploração de vulnerabilidades lógicas.

Em seguida, foi discutida a geração de logs de eventos, onde foram descritas todas as informações que precisam ser registradas e o formato de representação usado neste trabalho. Embora essa coleta de informações introduza um *overhead* de desempenho e armazenamento, é um requisito para a análise de divergência entre modelo e comportamento no mundo real que a abordagem propõe. Conforme discutido, técnicas como registro de log assíncrono e o uso de API Gateways podem mitigar esse impacto.

Por fim, foi discutido o processo de análise e classificação das divergências encontradas, o que embasa a necessidade de um analista verificá-las manualmente a fim de eliminar falsos positivos e realizar a distinção entre modelos desatualizados, desvios benignos e desvios maliciosos.

[5] Acessível em `https://github.com/ailton07/openapi-links-to-CPNs`

Capítulo 6

Resultados

Nesta seção, apresentamos a validação experimental realizada e as ameaças à validade da abordagem proposta. De forma análoga a outros trabalhos da literatura (Collado et al., 2020; Schoenborn and Althoff, 2021), para verificar o funcionamento da abordagem proposta e testar a acurácia de detecção de ataques, realizamos testes com uma aplicação de estudos com vulnerabilidades *by-design*, o *Juice Shop*, e com uma aplicação do mundo real que teve vulnerabilidades publicamente confirmadas, nesse caso, a aplicação de código aberto Memos. Primeiro testamos a abordagem no exemplo em execução apresentado no Capítulo 4. Em seguida, executamos uma avaliação baseada em usuários para validar a abordagem. Por fim, avaliamos um software de código aberto vulnerável a BOLA no mundo real com a ferramenta `Links2CPN`.

6.1 Juice Shop

O OWASP Juice Shop [1] é uma aplicação web educacional que replica um site de *e-commerce*. Foi desenvolvida para ser utilizada em treinamentos de segurança, demos e eventos de CTF (*capture the flag*). Contém *by-design*, isto é, intencionalmente, vulnerabilidades do OWASP Top 10 e outras mais do mundo real. A aplicação foi desenvolvida utilizando as tecnologias Node.js, Express e Angular e contém um considerável número de desafios de segurança (100) de dificuldade variável, onde o usuário deve explorar as vulnerabilidades subjacentes. Os usuários conseguem acompanhar seu progresso através de um painel com placar.

De acordo com os desenvolvedores da solução, além dos casos de uso de treinamentos hacker, conscientização, prática de testes de penetração e comparação de scanners de segurança, o Juice Shop também pode ser usado para testar a

[1] `https://owasp.org/www-project-juice-shop/`

eficácia de ferramentas de segurança com uma aplicação Web com frontend e REST APIs que replica um sistema do mundo real. Essa é a principal razão de estarmos testando a solução proposta neste trabalho com o Juice Shop.

O Juice Shop foi criado com os seguintes desafios relacionados à vulnerabilidade lógica *Broken Object Level Authorization*:

- Vulnerabilidade 1.: *View Basket*, que consiste em visualizar o cesto de compras de outro usuário;
- Vulnerabilidade 2.: *Manipulate Basket*, que consiste em colocar um produto no cesto de compras de outro usuário.

Os criadores do Juice Shop publicaram um guia de como explorar esses 2 desafios, em outras palavras, a solução para eles [2]. Dessa forma, o teste que faremos consiste em reproduzir os passos para a exploração dessas vulnerabilidades e em seguida, verificar se a solução proposta nesse trabalho é capaz de detectar os ataques.

6.1.1 Preparações para os testes

O primeiro passo para executar os testes é instrumentar o OWASP Juice Shop para gerar arquivos de logs com o formato descrito na Seção 5.3. O novo código dessa versão alterada do Juice Shop pode ser acessado online[3]. Após isso, para ter a aplicação rodando no cenário mais próximo possível de uma aplicação de e-commerce real, seguimos as instruções da documentação do Juice Shop para hospedar a aplicação Web na *AWS EC2*, um serviço em nuvem que permite a criação de um servidor virtual para execução de aplicativos na infraestrutura da Amazon Web Services. Assim, a aplicação ficou publicamente disponível online e acessível através de uma URL com o formato `http://ec2-XXX-XXX-XXX-XXX.compute-1.amazonaws.com:3000`, onde os X representam o IP público da instância em execução. A cada vez que reiniciamos a instância que executa a aplicação Web, todas as informações e logs são deletados e a aplicação volta ao seu estado original.

6.1.2 Vulnerabilidade 1: View Basket

A Vulnerabilidade 1, *View Basket*, consiste em um caso clássico de *Broken Object Level Authorization*. O comportamento legítimo da aplicação é, ao carregar o website e realizar o login, o frontend envia uma requisição *POST* ao endpoint

[2] `https://pwning.owasp-juice.shop/appendix/solutions.html`
[3] `https://github.com/ailton07/juice-shop-with-winston`

/rest/user/login da API, retornando para o cliente da requisição o token do usuário e o identificador do cesto de compras do usuário, representado pelo atributo *bid*. Após isso, é exibida a página inicial do website. Uma das requisições enviadas nesse processo é a *GET /rest/basket/6*, onde o valor 6 refere-se ao *bid* obtido na requisição de login.

Seguindo o guia de como explorar as vulnerabilidades do Juice Shop, o primeiro passo para a exploração dessa vulnerabilidade é realizar o login na aplicação Web, o que exige que tenhamos criado previamente uma conta. A tela de login consiste em um formulário de login e um botão de login social, como mostrado na Figura 6.1. Ao digitar o email e a senha de usuário e clicar em *Log in*, a aplicação cliente envia ao servidor a requisição *POST /rest/user/login*, apresentada no último parágrafo.

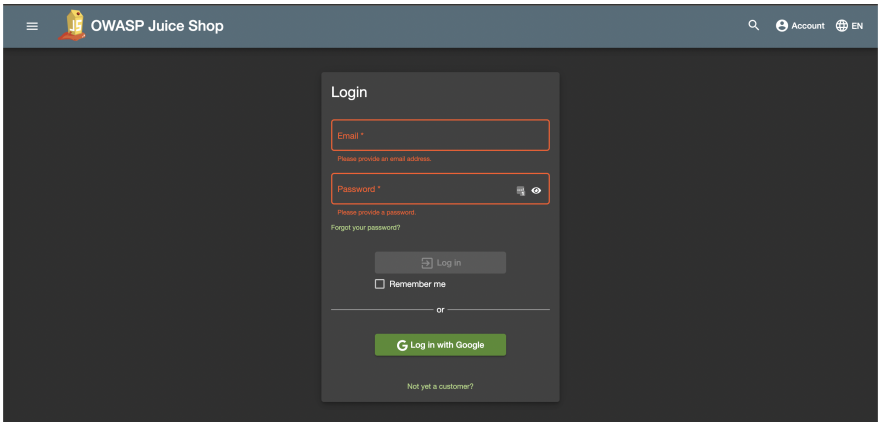


Figura 6.1: Tela de login do Juice Shop.

Após o login, o usuário é automaticamente redirecionado para a tela inicial do Juice Shop, mostrada na Figura 6.2. Dentre as várias requisições enviadas ao servidor para renderizar a tela, é enviada a requisição *GET /rest/basket/6*, apresentada anteriormente. O próximo passo é a execução do ataque. Observando que a requisição *GET /rest/basket/6* possui o valor 6 sendo passado como um *path parameter*, basta enviar uma nova requisição ao servidor mudando o valor 6 para outro, como 4 ou 5, até encontrar um identificador de cesto de compras válido. De acordo com o guia de como explorar as vulnerabilidades do Juice Shop, basta subtrair ou somar um valor (5 e 7, respectivamente) para explorar a vulnerabilidade com sucesso.

Após executar esse processo, obtemos um arquivo de log [4] com 30 linhas. Executamos a solução proposta com esse arquivo de log e com o documento Ope-

[4] `https://github.com/ailton07/openapi-links-to-CPNs/blob/main/logs/real_logs_juice_shop_signup_view_basket.log`

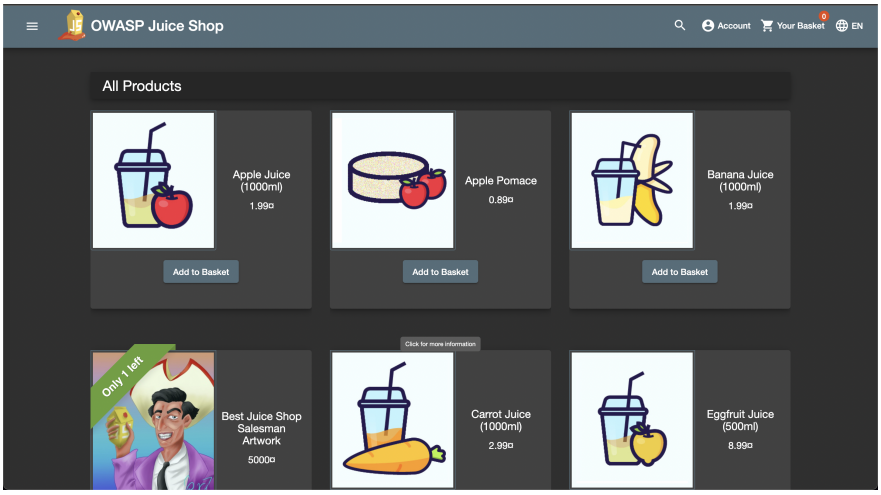
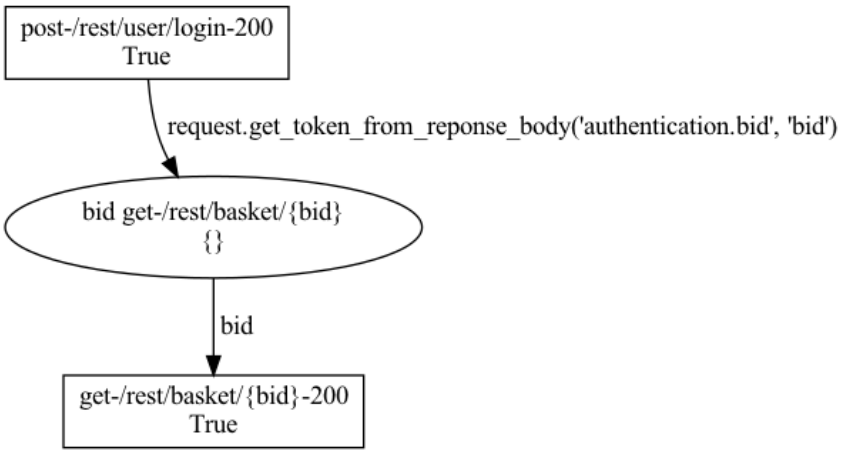


Figura 6.2: Tela inicial do Juice Shop.

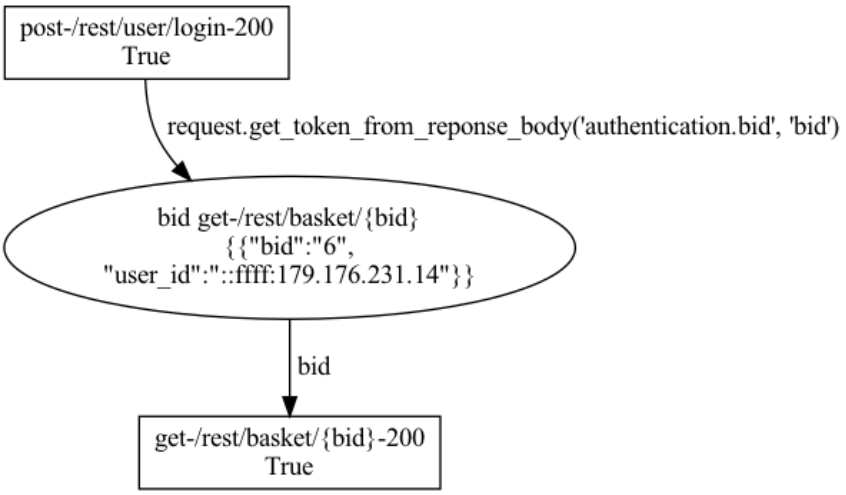
nAPI mostrado na Listagem 4.1 como entradas. Como resultado, obtivemos as 2 imagens mostradas na Figura 6.3 e a mensagem de erro mostrada na Listagem 6.1. A Figura 6.3a mostra o estado inicial da CPN, sendo o resultado imediato da transformação de OpenAPI para CPN. A Figura 6.3b mostra o estado da CPN após o processamento da linha 23 do arquivo de log que representa a requisição *POST /rest/user/login* com resposta $bid = 6$ e o estado da CPN após o processamento da linha 26, representando a requisição *GET /rest/basket/6*. Por fim, a Listagem 6.1 representa o processamento da linha 29 do arquivo de log, requisição *GET /rest/basket/7* que é o ataque BOLA que realizamos. A mensagem na listagem descreve brevemente que não existe um token disponível no lugar da CPN para executar a transição, indicando um desvio no fluxo normal da aplicação.

```
1 Fire error, Line 29: GET /rest/basket/7 200 10ms
2 transition not enabled for
3 {
4    "bid ->"{
5       "bid":"None",
6       "user_id":"::ffff:179.176.231.14"
7    },
8    "request ->"{
9       "uri":"/rest/basket/7",
10      "method":"GET",
11      "user_id":"::ffff:1..."}
12 }
```

Listagem 6.1: Mensagem de erro recebida ao executar Vulnerabilidade 1



(a) Resultado da execução do algoritmo sobre o log de eventos da Vulnerabilidade 1, estado inicial da CPN passo a).



(b) Resultado da execução do algoritmo sobre o log de eventos da Vulnerabilidade 1, passo b).

Figura 6.3: Resultado da execução do algoritmo sobre o log de eventos da Vulnerabilidade 1. Passos a) e b).

6.1.3 Vulnerabilidade 2: Manipulate Basket

A Vulnerabilidade 2, *Manipulate Basket*, consiste em um caso de *Broken Object Level Authorization* explorável por *HTTP Parameter Pollution* (HPP)[5]. O comportamento legítimo da aplicação é ao carregar o website e realizar o login, é enviada uma requisição *POST* ao endpoint */rest/user/login* da API , retornando para o cliente da requisição o token do usuário e o identificador do cesto de compras do usuário, atributo *bid*, de forma similar ao desafio anterior. Após isso, é exibida a página inicial do website. Ao clicar em um produto qualquer e adicioná-lo na cesta de compras, através do botão *Add to Basket*, é enviada uma requisição *POST* ao endpoint */api/BasketItems/* da API com os valores *ProductId* (código do produto), *BasketId* (identificador do cesto de compras do usuário) e *quantity* (quantidade do produto a ser adicionada).

Seguindo o guia de como explorar as vulnerabilidades do Juice Shop, o primeiro passo para a exploração dessa vulnerabilidade é realizar o login na aplicação Web. Após isso, adicionar um item qualquer ao carrinho de compras, inspecionando a requisição *POST /api/BasketItems/* e a modificando para conter dois valores *BasketId*, onde o primeiro valor é o identificador da cesta do usuário logado e o segundo valor é o identificador da cesta de compras da vítima. A requisição final a ser enviada ao servidor é como a mostrada na Listagem 6.2, onde o campo *"BasketId":"6"* refere-se ao cesto de compras do atacante e *"BasketId":"7"* refere-se ao cesto de compras da vítima.

```
1 {
2     "ProductId":1,
3     "BasketId":"6",
4     "quantity":1,
5     "BasketId":"7"
6 }
```

Listagem 6.2: Requisição para explorar Desafio 2.

Após executar o processo de cadastro de conta, login e exploração do ataque, obtivemos um arquivo de log com 23 linhas, acessível online aqui[6]. Executamos a solução proposta com esse arquivo de log e com o documento OpenAPI disponível online[7] como entradas. Como resultado, obtivemos as 2 imagens mostradas na Figura 6.4 e a mensagem de erro mostrada na Listagem 6.3.

[5] https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/04-Testing_for_HTTP_Parameter_Pollution

[6] https://github.com/ailton07/openapi-links-to-CPNs/blob/main/logs/real_log_Juice_Shop_manipulate_basket.log

[7] https://github.com/ailton07/openapi-links-to-CPNs/blob/main/examples/OWASP-Juice-Shop-manipulate-basket.yaml

A Figura 6.4 *a)* mostra o estado inicial da CPN, sendo o resultado imediato da transformação de OpenAPI para CPN. A Figura 6.4 *b)* mostra o estado da CPN após o processamento da linha 11 do arquivo de log que representa a requisição *POST /rest/user/login* com resposta $bid = 6$. Por fim, a Listagem 6.1 representa o processamento da linha 21 do arquivo de log, requisição *POST /api/BasketItems/* que é o ataque BOLA que realizamos. A mensagem na listagem descreve brevemente que não existe um token disponível no lugar da CPN para executar a transição. Identificando corretamente o ataque.

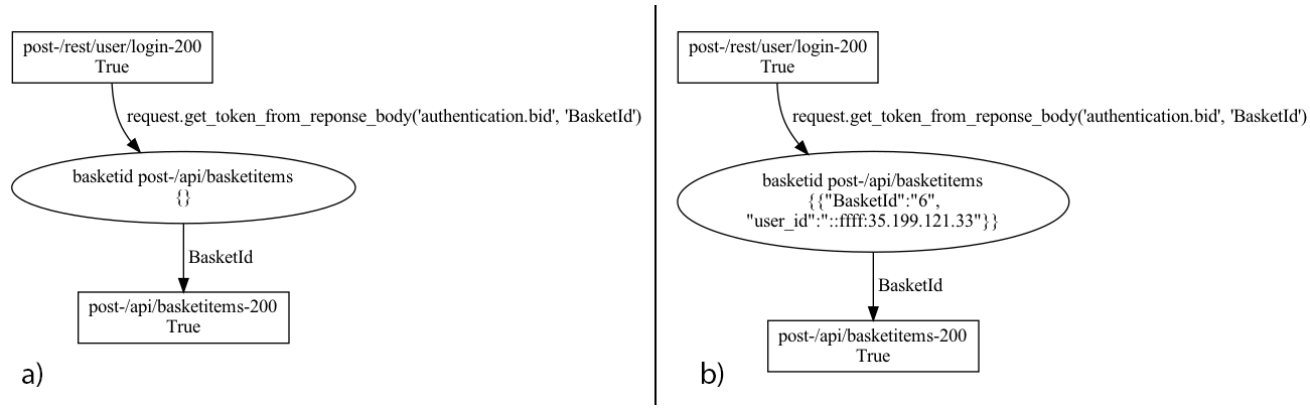


Figura 6.4: Resultado da execução do algoritmo sobre o log de eventos do desafio 2.

```
1 Fire error, Line 21: POST /api/BasketItems/ 200 11ms
2 transition not enabled for
3 {
4    "BasketId ->"{
5       "BasketId":"5",
6       "user_id":"::ffff:35.199.121.33"
7    },
8    "request ->"{
9       "uri":"/api/BasketItems/",
10      "method":"POST",
11      "user_id":"::ff...
12   }
13 }
```

Listagem 6.3: Mensagem de erro recebida ao executar Vulnerabilidade 2.

6.1.4 Avaliação Baseada em Usuários

Para testar a solução proposta em um cenário mais realista, realizamos uma avaliação através da participação dos utilizadores do sistema, as pessoas a quem o sistema se destina, em outras palavras, fizemos um *user-based evaluation*.

Para isso, convidamos estudantes de computação da Universidade Federal do Amazonas, inscritos na disciplina de Segurança de Redes de Computadores (ICC303), a tentar explorar a Vulnerabilidade 1 e Vulnerabilidade 2, descritas nas seções anteriores. Havia 20 alunos matriculados na disciplina, onde 15 cursaram até o final, que foram os convidados para esta avaliação.

Como ambiente de teste, configuramos uma instância do Juice Shop em um servidor Web da AWS (EC2) de 25 de Janeiro de 2023 a 8 de Fevereiro de 2023. Primeiro, explicamos aos estudantes os fundamentos da vulnerabilidade BOLA e demos exemplos de exploração. Em seguida, apresentamos a aplicação Juice Shop e instruímos os alunos a explorarem ambas as vulnerabilidades no carrinho de compras como achassem apropriado, escrevendo um texto/relatório sobre o processo. Também os aconselhamos a não copiar soluções da Internet, mas a tentar criar suas próprias explorações das vulnerabilidades.

No final, obtivemos 12 relatórios de ataques, aproximadamente 4.000 linhas de logs, 2.000 requisições analisadas, 517 requisições associadas à Vulnerabilidade 1 e 192 requisições associadas à Vulnerabilidade 2. Para cada um dos relatórios recebidos, analisamos manualmente os registros gerados, especialmente aqueles associados ao processo de registro, login, à Vulnerabilidade 1 e à Vulnerabilidade 2. Graças a essa análise manual, fomos capazes de calcular o número de solicitações legítimas, o número de tentativas de ataque e o número de ataques bem-sucedidos. Posteriormente, processamos os logs gerados com a solução proposta neste trabalho para obter as requisições classificadas como legítimas e como ataques e as comparamos com os dados obtidos através da nossa análise manual.

Como resultado, obtivemos os dados mostrados na Tabela 6.1. Para a Vulnerabilidade 1, *Total de Requisições* refere-se ao número de requisições registradas em `GET /rest/basket/{id}`, enquanto para a Vulnerabilidade 2, refere-se ao número de requisições registradas em `POST /api/BasketItems/`. *Tentativas de Ataque* refere-se ao número de solicitações que classificamos manualmente como tentativas de ataque (mal-sucedidas e bem-sucedidas) nos endpoints citados, enquanto *Ataques Bem-Sucedidos* refere-se ao número de solicitações que foram classificadas manualmente como ataques bem-sucedidos. Por fim, *Requisições Classificadas como Ataques* refere-se ao número de solicitações que foram classificadas pela ferramenta `Links2CPN` como ataques nos endpoints citados.

De acordo com Kononenko and Kukar (2007), matrizes de confusão e métricas como *Acurácia*, *Precisão*, *Recall* e *F1-score* são comumente usadas para avaliar a qualidade de algoritmos de classificação. Por isso, para expressar o desempenho da abordagem proposta, calculamos essas métricas através das seguintes equações:

Tabela 6.1: Resumo dos dados obtidos a partir dos ataques.

	Vuln. 1	Vuln. 2
Total de Requisições	517	192
Tentativas de Ataque	101	88
Ataques Bem-Sucedidos	98	20
Requisições Classificadas como Ataques	101	88

$$
\begin{aligned}
\textit{Acurácia} &= \frac{TP + TN}{TP + FP + FN + TN} \\
\textit{Precisão} &= \frac{TP}{TP + FP} \\
\textit{Recall} &= \frac{TP}{TP + FN} \\
\textit{F1-score} &= \frac{2 \cdot \textit{Recall} \cdot \textit{Precision}}{\textit{Recall} + \textit{Precision}}
\end{aligned}
\tag{6.1}
$$

Onde falsos positivos (FP) se referem a requisições benignas que são erroneamente classificadas como ataques, falsos negativos (FN) se referem a requisições de ataque que não foram classificadas como ataques, verdadeiros positivos (TP) se referem a requisições de ataque que são corretamente classificadas como tal, e verdadeiros negativos (TN) se referem a requisições benignas que são corretamente classificadas como benignas.

A matriz de confusão é mostrada na Tabela 6.2(a). Para a Vulnerabilidade 1 (V1) e Vulnerabilidade 2 (V2), obtivemos $\textit{Acurácia} = \textit{Precisão} = \textit{Recall} = \textit{F1-score} = 1$. Em ambos os casos, *Acurácia*, *Precisão*, *Recall* e *F1-score* permaneceram constantes.

Para aprofundar a investigação nas métricas de desempenho entre V1 e V2, criamos a matriz de confusão comparando os ataques bem-sucedidos e aqueles classificados por nossa abordagem. Os resultados estão resumidos na Tabela 6.2(b). Neste caso, obtivemos $\textit{Acurácia} = 0{,}99$; $\textit{Precisão} = 0{,}97$; $\textit{Recall} = 1$; $\textit{F1-score} = 0{,}98$ para Vulnerabilidade 1, enquanto para Vulnerabilidade 2 obtivemos $\textit{Acurácia} = 0{,}64$; $\textit{Precisão} = 0{,}22$; $\textit{Recall} = 1$; $\textit{F1-score} = 0{,}37$. Em ambos os casos, o *Recall* permaneceu constante, indicando consistência na identificação de casos positivos (ataques com sucesso). No entanto, a precisão relativamente baixa para a Vulnerabilidade 2 demonstra que a abordagem gera um número substancial de predições positivas incorretas, resultando em uma alta taxa de falsos positivos para detecção de ataques bem-sucedidos. Estes resultados indicam que a abordagem proposta requer heurísticas adicionais para distinguir entre tentativas

Tabela 6.2: Matrizes de confusão.

			Valores estimados	
			Positivo	**Negativo**
Reais	V1	**Positivo**	101	0
		Negativo	0	416
	V2	**Positivo**	88	0
		Negativo	0	104

(a) Tentativas de ataques versus detecções

			Valores estimados	
			Positivo	**Negativo**
Reais	V1	**Positivo**	98	0
		Negativo	3	416
	V2	**Positive**	20	0
		Negative	68	104

(b) Ataques bem-sucedidos versus detecções.

de ataque e ataques bem-sucedidos, tais como considerar o *HTTP Status Code* da resposta ou exigir análise manual por um especialista.

Os resultados também mostram que as vulnerabilidades Vulnerabilidade 1 e Vulnerabilidade 2 têm taxas de *tentativa de ataque/sucesso no ataque* muito diferentes. Para Vulnerabilidade 1, houve 98 ataques bem-sucedidos em 101 tentativas de ataque (ou seja, taxa de sucesso de 97,02%), enquanto para Vulnerabilidade 2, houve 20 ataques bem-sucedidos em 88 tentativas de ataque (ou seja, taxa de sucesso de 22,72%). Uma explicação para essa diferença na taxa de sucesso na exploração das vulnerabilidades é a diferença na dificuldade de exploração. De acordo com o guia de exploração do Juice Shop [8], Vulnerabilidade 1 tem 2 estrelas de dificuldade, enquanto Vulnerabilidade 2 tem 3 estrelas. De fato, 3 dos 12 alunos que enviaram relatórios não conseguiram explorar com sucesso a Vulnerabilidade 2.

6.2 Avaliação com Aplicação do Mundo Real

Com o objetivo de testar a abordagem proposta com uma ferramenta do mundo real com vulnerabilidade BOLA, elegemos o serviço de anotações de código aberto Memos [9] que é gratuito e foi criado para priorizar a privacidade, através do *self-hosting*.

Em 2022, foi identificada uma vulnerabilidade BOLA no *backend* do Memos, onde um usuário poderia arquivar as notas de outros usuários apenas trocando o identificador na nota na requisição da API REST. Após aplicarem uma correção na aplicação, a vulnerabilidade foi divulgada sob o identificador *CVE-2022-4814* [10] e foi publicada uma prova de conceito demonstrando como reproduzir a vulnerabilidade [11]. Para verificar se a `Links2CPN` é capaz de detectar explorações dessa vulnerabilidade, realizamos alguns preparativos descritos na próxima seção.

[8] Disponível publicamente em `https://pwning.owasp-juice.shop/appendix/solutions.html`.

[9] Disponível publicamente em `https://usememos.com/`

[10] `https://nvd.nist.gov/vuln/detail/CVE-2022-4814`

[11] `https://huntr.com/bounties/e65b3458-c2e2-4c0b-9029-e3c9ee015ae4`

6.2.1 Preparações para os testes

A fim de sermos capazes de testar a vulnerabilidade, realizamos os seguintes passos:

1. Criamos um *fork* do repositório original [12];
2. Revertemos a correção implementada sobre a vulnerabilidade BOLA, a fim de tornar a aplicação vulnerável novamente;
3. De forma análoga ao Juice Shop, modificamos a aplicação para gerar arquivos de logs com o formato descrito na Seção 5.3;
4. Convertemos a documentação da API para OpenAPI 3.0 e adicionamos *links* nos endpoints envolvidos nos cenários de teste;
5. Executamos a aplicação localmente e reproduzimos dois cenários: i) usuário legítimo realiza o login, cria uma nota e a arquiva; ii) usuário malicioso realiza o login e arquiva uma das notas do usuário legítimo (exploração da vulnerabilidade).

6.2.2 Teste com os Casos Legítimo e Malicioso

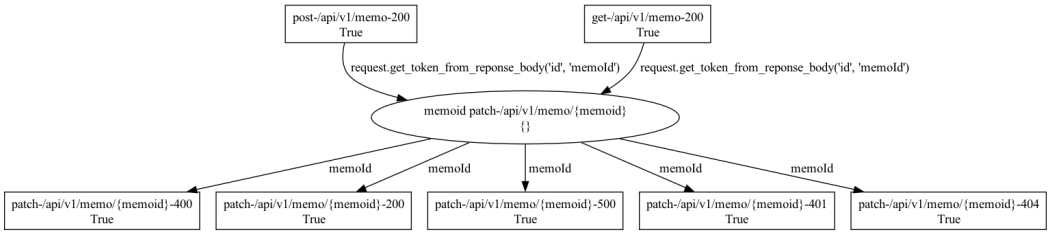
Como dito anteriormente, a vulnerabilidade em questão trata-se de um caso de BOLA. Após realizar o login, a tela carrega as notas existentes do usuário automaticamente, através da requisição *GET* a */api/v1/memo*. A partir daí, o usuário pode criar novas notas com a requisição *POST* ao endpoint */api/v1/memo*, ou arquivar notas existentes com a requisição *PATCH* a */api/v1/memo/{id}*, onde *id* é um número sequencial identificando a nota. O ataque consiste em enviar requisições *PATCH* a */api/v1/memo/{id}* com diferentes *ids*.

Ao executar a solução proposta com o arquivo de log do primeiro cenário [13], onde o usuário legítimo realiza o login, cria uma nova nota e a arquiva, obtemos o resultado apresentado na Figura 6.5. A Figura 6.5 *a)* mostra o estado inicial da CPN, sendo o resultado imediato da transformação de OpenAPI para CPN. A Figura 6.5 *b)* mostra o estado da CPN após o processamento da linha 9, uma requisição *GET* ao endpoint */api/v1/memo* que lista as notas existentes do usuário, nesse caso com os *ids* 4 e 6. Por sua vez, foram gerados na CPN tokens com *memoId=4* e *memoId=6*. A Figura 6.5 *c)* mostra o estado da CPN após o processamento da linha 12 que é uma requisição *POST* ao endpoint */api/v1/memo*, criando uma nova nota. Como resultado, foi gerado um novo

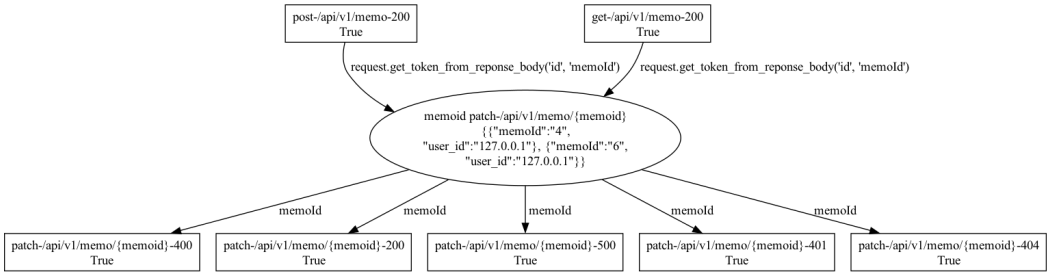
[12] Disponível publicamente em `https://github.com/ailton07/memos-with-BOLA/`

[13] Disponível publicamente em `https://github.com/ailton07/memos-with-BOLA/blob/main/memos_login_create_archieve.log`

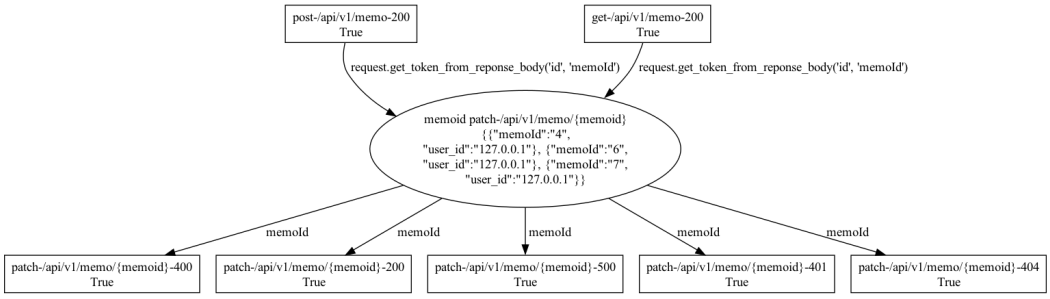
token na CPN, com o *memoId=7*. Após isso, a linha 14 é processada, se tratando do arquivamento na nota de *id* 7, através da requisição *PATCH /api/v1/memo/7*, como essa é uma requisição legítima e não causa mudança na CPN, não foram geradas novas figuras ou mensagens de erro, finalizando o teste do uso legítimo.



(a)



(b)



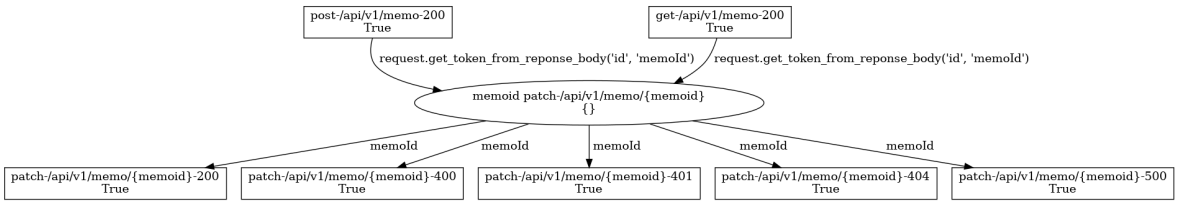
(c)

Figura 6.5: Resultado da execução do algoritmo sobre o log de eventos legítmos do Memos.

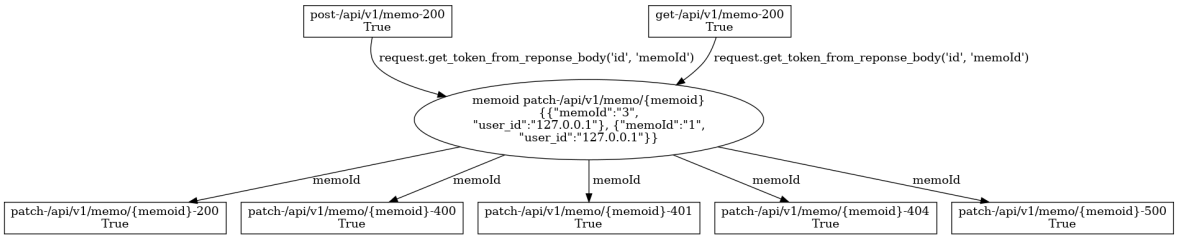
Ao executar a solução proposta com o arquivo de log do segundo cenário [14], onde um usuário malicioso realiza o login e arquiva a nota de outro usuário,

[14]Disponível publicamente em `https://github.com/ailton07/memos-with-BOLA/blob/main/memos_login_and_attack_log.log`

obtemos o resultado apresentado na Figura 6.6. A Figura 6.6 *a)* mostra o estado inicial da CPN, sendo o resultado imediato da transformação de OpenAPI para CPN. A Figura 6.6 *b)* mostra o estado da CPN após o processamento da linha 9, uma requisição *GET* ao endpoint */api/v1/memo* que lista as notas existentes do usuário, nesse caso com os *ids* 1 e 3. Por sua vez, foram gerados na CPN tokens com *memoId=1* e *memoId=3*. Após isso, a linha 11 é processada, se tratando do arquivamento na nota de outro usuário, *id* 6, através da requisição *PATCH /api/v1/memo/6*. Uma vez que o usuário não possui na CPN um token com o *memoId=6*, é detectada a violação no fluxo de dados da aplicação. Como resultado, é exibida a mensagem de erro mostrada na Listagem 6.4. A mensagem indica um erro ao disparar a transição *PATCH /api/v1/memo/{memoid}*, apontando que ela não está habilitada para o token de *memoId=6*.



(a)



(b)

Figura 6.6: Resultado da execução do algoritmo sobre o log de eventos legítmos do Memos.

```
1 Fire error, Line 11: PATCH /api/v1/memo/6 200
2 transition not enabled for
3 {
4     "memoId ->"{
5         "memoId":"None",
6         "user_id":"127.0.0.1"
7     },
8     "request ->"{
9         "uri":"/api/v1/memo/6",
10        "method":"PATCH",
```

```
11         "user_id":"\"127.0....
12       }"
13 }
```

Listagem 6.4: Mensagem de erro recebida ao executar Caso 2 - Memos.

6.3 Discussão

Este capítulo apresentou a validação experimental e resultados da abordagem proposta para a detecção de Vulnerabilidades Lógicas de Adulteração de Parâmetros e Controles de Acesso em APIs Web REST, utilizando a especificação OpenAPI, modelagem com Redes de Petri Coloridas (CPNs) e logs de eventos. A avaliação foi realizada através de uma série de testes em cenários diferentes, progressivamente mais complexos e próximos do mundo real.

Inicialmente, demonstrou-se a capacidade da ferramenta `Links2CPN` em detectar ataques conhecidos em um ambiente controlado. Utilizando a aplicação OWASP Juice Shop, com vulnerabilidades *by-design*, reproduzimos manualmente duas explorações de *Broken Object Level Authorization* (BOLA): *View Basket* (Vulnerabilidade 1) e *Manipulate Basket* (Vulnerabilidade 2). Em ambos os casos, a ferramenta analisou os logs gerados e, com base no modelo CPN derivado da especificação OpenAPI, identificou corretamente as requisições maliciosas que violavam o fluxo de dados esperado, gerando mensagens de erro indicativas da detecção.

Em seguida, realizamos uma avaliação baseada em usuários para verificar o desempenho da abordagem em um cenário com maior variabilidade de interações e tentativas de ataque. Estudantes de computação foram convidados a explorar as mesmas vulnerabilidades no Juice Shop, gerando um conjunto de dados com mais de 4.000 linhas de logs. A análise quantitativa, comparando as detecções da ferramenta desenvolvida `Links2CPN` com uma classificação manual, revelou excelente desempenho na identificação de tentativas de ataque, alcançando 100% em Acurácia, Precisão, Recall e F1-score para ambas as vulnerabilidades. Uma análise mais aprofundada, focando apenas nos ataques bem-sucedidos, mostrou alta performance para a Vulnerabilidade 1 (mais simples), mas uma precisão menor para a Vulnerabilidade 2 (mais complexa). Isso indica que, embora a ferramenta seja capaz de identificar tentativas de ataque, distinguir entre uma tentativa falha e um ataque bem-sucedido mostra-se mais desafiador e pode requerer heurísticas adicionais, como a análise do *HTTP Status Code* das respostas ou intervenção de especialistas, especialmente em cenários de ataque mais elaborados como no da Vulnerabilidade 2. Os resultados também corroboraram a diferença de dificuldade na exploração das vulnerabilidades.

Finalmente, a abordagem foi testada em um contexto mais próximo do mundo real, utilizando a aplicação de código aberto Memos, que possuía uma vulnerabilidade BOLA publicamente documentada (CVE-2022-4814). Após preparar o ambiente (revertendo o patch de segurança, instrumentando logs e adaptando a especificação OpenAPI), executamos cenários de uso legítimo e malicioso. A ferramenta `Links2CPN` processou corretamente as interações legítimas sem gerar alertas e detectou com sucesso a requisição que explorava a vulnerabilidade BOLA no cenário de ataque.

Dessa forma, os resultados experimentais apresentados neste capítulo demonstram a viabilidade da abordagem proposta. A análise dos resultados também apontou possíveis trabalhos futuros, como a incorporação de heurísticas para identificar ataques bem-sucedidos, por exemplo, que serão discutidos na próxima seção.

Capítulo 7

Conclusões

Este capítulo apresenta as conclusões desta tese. Após a exploração da modelagem de APIs com CPNs a partir de OpenAPI e da avaliação de um método de detecção de Vulnerabilidades Lógicas baseado em verificação de conformidade, este capítulo consolida os resultados obtidos. Serão discutidas as limitações da abordagem, seguido de uma análise do atingimento dos objetivos e da formalização das contribuições. Finalmente, serão propostas direções para trabalhos futuros que podem expandir ou refinar a pesquisa apresentada.

7.1 Limitações do Estudo

Esta seção discute as limitações e ameaças à validade que identificamos neste estudo (Runeson and Höst, 2009). Apesar de a avaliação da abordagem proposta ter revelado altos valores na *Acurácia*, *Precisão*, *Recall* e F1-score para a detecção de tentativas de ataque, foi possível determinar, com os testes em ambientes controlados e reais, 4 principais limitações da solução.

A primeira limitação trata da centralização de logs. Uma vez que, para a aplicação do algoritmo de Verificação de Conformidades, as chamadas à API precisam estar registradas no mesmo arquivo de log, em cenários onde os logs das APIs são gerados e armazenados de forma descentralizada, é necessário centralizar os logs, observando a cronologia dos eventos, para a correta aplicação da abordagem proposta. No entanto, mesmo em cenários altamente distribuídos, como em arquiteturas de microserviços, existem pontos centralizados, como os *APIs Gateways*, *proxies reversos* e *balanceadores de carga*, que facilitam a geração de logs centralizados.

A segunda limitação diz respeito a respostas não esperadas da API. Caso um atacante consiga explorar com sucesso uma vulnerabilidade, mas o servidor

envie uma resposta inesperada, de acordo com o descrito na documentação, a solução pode não ser capaz de detectar o ataque. Considere, por exemplo, a Vulnerabilidade 1, onde a especificação OpenAPI descreve o endpoint `GET /rest/basket/{bid}` que responde com o *status code* 200. Caso a vulnerabilidade seja explorada com sucesso, mas a API responda com *status code* 500, a solução proposta por esse trabalho não será capaz de detectar o ataque, uma vez que a API apresentou um comportamento diferente do descrito na especificação. Erros inesperados na aplicação, porém, podem ser detectados por um profissional conduzindo análises manuais na aplicação e nos logs, uma vez que costumam chamar a atenção e serem detectados até mesmo de forma automatizada, por ferramentas de monitoramento de logs, como *SIEMs*. No mundo real é recomendável e comum a combinação de diferentes ferramentas e abordagens para alcançar a *segurança em camadas*.

A terceira limitação é relacionada a disponibilidade e completude de documentações OpenAPI. Ainda que tecnologias como bibliotecas de geração de OpenAPI a partir do código, *Infrastructure as Code* e *APIs Gateways* tenham se popularizado, tornando comum a presença de descrições das APIs, a necessidade de manutenção dessas especificações para que permaneçam atualizadas ainda é considerada uma limitação da abordagem proposta. Porém, trabalhos como (Petryshyn, 2024) têm demonstrado a aplicabilidade da inteligência artificial para construir de forma autônoma documentos OpenAPI, reduzindo a carga manual de trabalho. Além disso, documentações OpenAPI incompletas ou incorretas levam a falhas na modelagem dos fluxos da aplicação, impedindo o correto funcionamento da abordagem.

A quarta limitação diz respeito à aplicação *offline* do algoritmo de Verificação de Conformidade. Nos testes realizados, utilizamos uma análise *offline*, onde um conjunto de eventos é representado por um log de eventos, em outras palavras, um arquivo contendo eventos passados. No entanto, essa não é uma limitação da abordagem, nem do algoritmo de Verificação de Conformidade. É possível aplicar a abordagem proposta a um fluxo de eventos obtidos em tempo de execução, através da leitura de uma fila em memória, ou de um *proxy reverso* que intercepte as requisições. Sendo essa uma oportunidade futura de melhoria.

Quanto às ameaças à validade do estudo, realizamos experimentos controlados que nos permitiram verificar a ferramenta e medir as métricas relevantes para a avaliação da abordagem proposta. Como não foram levantadas relações causais nos resultados obtidos, o estudo está livre de ameaças à validade interna. Quanto à avaliação baseada em usuários, é verdade que não há uma população da qual uma amostra estatisticamente representativa tenha sido extraída. No entanto, como a utilizamos como estudos de caso, podemos concluir que os resultados são extensíveis a casos que possuem características comuns e, portanto, os achados

são relevantes. Em relação à confiabilidade, a ferramenta `Links2CPN`, os exemplos de execução e os resultados das avaliações baseadas em usuários estão publicamente acessíveis através dos links para o repositório GitHub. Dessa forma, outros pesquisadores podem realizar os mesmos experimentos e estudos posteriormente, obtendo resultados semelhantes, garantindo a replicabilidade do trabalho.

Finalmente, no que diz respeito à validade externa, a abordagem proposta é baseada em Redes de Petri, que, embora ofereça vantagens claras em termos de verificação formal e detecção determinística de vulnerabilidades estruturadas, pode se beneficiar da integração com outras técnicas, como de mineração de dados e detecção de anomalias. Esses métodos podem complementar a abordagem proposta, fornecendo meios para detectar padrões de ataque desconhecidos ou em evolução com base em anomalias estatísticas ou desvios comportamentais no sistema. Pesquisas futuras poderiam explorar um modelo híbrido, onde as redes de Petri são usadas para análise formal e detecção de ataques conhecidos, enquanto técnicas baseadas em aprendizado de máquina identificam vetores de ataque nunca antes vistos, criando assim uma abordagem mais abrangente e em camadas para a segurança de APIs REST. Contudo, a abordagem também está vinculada a uma especificação OpenAPI concreta, podendo necessitar de adaptação para especificações futuras.

7.2 Atingimento dos Objetivos e Contribuições

Na Seção 1.2 foram introduzidos os objetivos gerais e específicos dessa tese, onde o objetivo geral foi elaborar e avaliar a eficácia de um método semi-automatizado para a detecção de ataques contra Vulnerabilidades Lógicas de Adulteração de Parâmetros e Controles de Acesso em APIs Web REST, através da verificação de conformidade entre um modelo normativo de uma especificação OpenAPI, expresso em CPN, e os comportamentos exibidos pela aplicação no mundo real.

Para o atingimento desse objetivo, foi necessário i) desenvolver uma representação em Rede de Petri Colorida dos fluxos e estruturas descritos nos documentos *OpenAPI*, o que foi apresentado na Seção 4.1; ii) criar e implementar um algoritmo de transformação de documentos *OpenAPI* para CPNs, apresentado na Seção 4.2; iii) demonstrar a aplicabilidade de um algoritmo de Verificação de Conformidade para identificar divergências entre o modelo em CPN e um conjunto de requisições e respostas da API e, assim, revelar ataques às Vulnerabilidades Lógicas de Adulteração de Parâmetros e Controles de Acesso, realizado nas Seções 5 e 6.

Considerando ainda que a seção 7.1 discutiu as limitações encontradas na abordagem e introduziu oportunidades de trabalhos futuros, consideramos que os objetivos específicos propostos nessa tese foram alcançados e por consequência, o objetivo geral.

Ao final deste trabalho, as seguintes contribuições foram alcançadas:

- Um modelo visual que permita a representação de APIs REST modeladas com OpenAPI, em Redes de Petri Coloridas, permitindo análises e a aplicação do conjunto de ferramentas voltadas a CPNs;
- Um algoritmo de transformação de modelos que permita a representação de especificações *OpenAPI* em Redes de Petri Coloridas;
- Um método de detecção de ataques de Vulnerabilidade Lógica Adulteração de Parâmetros e Controles de Acesso a partir de especificações *OpenAPI* e logs de eventos da API.

7.3 Trabalhos Futuros

Nesta seção, são discutidas as oportunidades de trabalhos futuros, baseadas principalmente nas limitações apresentadas.

A abordagem proposta pressupõe a existência de uma especificação OpenAPI fidedigna com a implementação real da API, atualizada e com os atributos necessários para a análise, como em outros trabalhos anteriores. Uma oportunidade de trabalho futuro seria o uso de técnicas de inteligência artificial, como *Large Language Models* (LLMs) especialistas em desenvolvimento de código, para criar e atualizar documentos OpenAPI com base no código fonte da API e/ou nas requisições e respostas da API. Essa abordagem reduziria o risco de a documentação estar desatualizada ou com informações ausentes em relação ao código da API. Além disso, eliminaria a necessidade de desenvolver manualmente a especificação OpenAPI.

A primeira limitação apresentada na seção 7.1 diz respeito a centralização de logs, o que pode ser difícil em cenários altamente distribuídos, apesar de teoricamente possível, mas abre uma oportunidade relevante para testes em diferentes cenários, incluindo testes como API que seguem outras arquiteturas, como APIs *severless*. Uma outra linha de melhoria seria a eliminação completa da necessidade de logs de eventos, através da aplicação *online* da abordagem proposta em um ponto de entrada centralizado, como em um *API Gateway*. Por outro lado, essa nova abordagem inseriria novas complexidades, como o gerenciamento do atraso no tempo de processamento das requisições (*overhead*).

Neste trabalho, propomos uma abordagem semi-automatizada que exige a supervisão de um analista humano, em outras palavras, *"human in the loop"* (HITL). Com os avanços recentes da IA, uma tendência emergente é o uso de agentes de inteligência artificial para realizar operações que antes exigiam trabalho manual. Para reduzir a carga sobre analistas, poderia ser explorado o uso de

machine learning para classificar automaticamente as violações detectadas pela análise de conformidade da CPN, transformando a abordagem em totalmente automatizada.

Finalmente, uma análise híbrida seria uma adição a análise estrutural baseada em Redes de Petri Coloridas e algoritmos de Verificação de Conformidade. Para cobrir um espectro maior de ameaças, outras abordagens, como mineração de dados e detecção de anomalias poderiam ser agregadas a abordagem proposta, com o objetivo de identificar novos padrões de ataque.

Estas oportunidades de trabalhos futuros apresentam caminhos promissores para avançar o estado da arte na segurança de APIs, aproveitando os fundamentos estabelecidos nesta tese.

Referências Bibliográficas

Alarcon, R., Wilde, E., and Bellido, J. (2010). Hypermedia-driven restful service composition. In *International Conference on Service-Oriented Computing*, pages 111–120. Springer.

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.

Barabanov, A., , Dergunov, D., Makrushin, D., Teplov, A., , and and (2022). AUTOMATIC DETECTION OF ACCESS CONTROL VULNERABILITIES VIA API SPECIFICATION PROCESSING. *Voprosy kiberbezopasnosti*, (1(47)):49–65.

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

Bernardi, S., Alastuey, R. P., and Trillo-Lado, R. (2017). Using process mining and model-driven engineering to enhance security of web information systems. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 160–166. IEEE.

Burattin, A. and Carmona, J. (2017). A framework for online conformance checking. In *International Conference on Business Process Management*, pages 165–177. Springer.

Carmona, J., van Dongen, B., Solti, A., and Weidlich, M. (2018). Conformance checking. *Switzerland: Springer.[Google Scholar]*.

Carrasquel, J. C., Mecheraoui, K., and Lomazova, I. A. (2020). Checking conformance between colored petri nets and event logs. In *International Conference on Analysis of Images, Social Networks and Texts*, pages 435–452. Springer.

Cheh, C. and Chen, B. (2021). Analyzing openapi specifications for security design issues. In *2021 IEEE Secure Development Conference (SecDev)*, pages 15–22.

Collado, E. S., Castillo, P. A., and Merelo Guervós, J. J. (2020). Using evolutionary algorithms for server hardening via the moving target defense technique. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 670–685. Springer.

Cotton, I. W. and Greatorex Jr, F. S. (1968). Data structures and techniques for remote computer graphics. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 533–544.

Decker, G., Lüders, A., Overdick, H., Schlichting, K., and Weske, M. (2009). Restful petri net execution. In Bruni, R. and Wolf, K., editors, *Web Services and Formal Methods*, pages 73–87, Berlin, Heidelberg. Springer Berlin Heidelberg.

Deepa, G. (2018). *Behavior-Based Attack Generation for Detecting Web Application Vulnerabilities*. PhD thesis, National Institute of Technology Karnataka, Surathkal.

Deepa, G. and Thilagam, P. S. (2016). Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Information and Software Technology*, 74:160–180.

Deepa, G., Thilagam, P. S., Praseed, A., and Pais, A. R. (2018). Detlogic: A black-box approach for detecting logic vulnerabilities in web applications. *Journal of Network and Computer Applications*, 109:89–109.

Deng, G., Zhang, Z., Li, Y., Liu, Y., Zhang, T., Liu, Y., Yu, G., and Wang, D. (2023). NAUTILUS: Automated RESTful API vulnerability detection. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5593–5609, Anaheim, CA. USENIX Association.

Du, W., Li, J., Wang, Y., Chen, L., Zhao, R., Zhu, J., Han, Z., Wang, Y., and Xue, Z. (2024). Vulnerability-oriented testing for RESTful APIs. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 739–755, Philadelphia, PA. USENIX Association.

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.

Fortinet (2025). Web application security report 2025: Evolving threats, strategies, and best practices. `https://www.fortinet.com/content/dam/maindam/PUBLIC/02_MARKETING/08_Report/2025-Web-Application-Security-Report-Fortinet.pdf`.

Gómez, A., Rodríguez, R. J., Cambronero, M.-E., and Valero, V. (2019). Profiling the publish/subscribe paradigm for automated analysis using colored petri nets. *Software & Systems Modeling*, 18(5):2973–3003.

Google Cloud (2021). The State of API Economy 2021 Report. Technical report.

Haddad, R. and Malki, R. E. (2022). Openapi specification extended security scheme: A method to reduce the prevalence of broken object level authorization.

Hoffman, A. (2020). *Web application security : exploitation and countermeasures for modern web applications / Andrew Hoffman*. O'Reilly Media, Inc, Place of publication not identified.

Hunter, K. L. (2017). *Irresistible APIs : designing web APIs that developers will love / Kristen L. Hunter.* Manning Publications, Shelter Island, NY, 1st edition edition.

Imperva (2024). The State of API Security in 2024. Technical report, Imperva.

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

Ivanchikj, A. (2021). *RESTalk: A Visual and Textual DSL for Modelling RESTful Conversations*. Phd, Università della Svizzera italiana (USI=, Lugano.

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.

Jensen, K. (1996). Springer Berlin Heidelberg, Berlin, Heidelberg.

Jensen, K. (1997). A brief introduction to coloured petri nets. In *Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '97, page 203–208, Berlin, Heidelberg. Springer-Verlag.

Jensen, K. and Kristensen, L. M. (2009). *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media.

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

Kallab, L. (2019). *Static and Automatic Resource Composition in Web-based Environments: An Application for Buildings Energy Management*. PhD thesis, Université de Pau et des Pays de l'Adour; Univerza na Primorskem (Koper ....

Kallab, L., Mrissa, M., Chbeir, R., and Bourreau, P. (2017). Using colored petri nets for verifying restful service composition. In Panetto, H., Debruyne, C., Gaaloul, W., Papazoglou, M., Paschke, A., Ardagna, C. A., and Meersman, R., editors, *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, pages 505–523, Cham. Springer International Publishing.

Kononenko, I. and Kukar, M. (2007). *Machine learning and data mining*. Horwood publishing.

Kopecký, J., Fremantle, P., and Boakes, R. (2014). A history and future of web apis. *it - Information Technology*, 56(3):90–97.

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.

Li, L. and Chou, W. (2011). Design and describe rest api without violating rest: A petri net based approach. In *2011 IEEE International Conference on Web Services*, pages 508–515. IEEE.

Li, L. and Chou, W. (2015). Designing large scale rest apis based on rest chart. In *2015 IEEE International Conference on Web Services*, pages 631–638. IEEE.

Li, X. and Xue, Y. (2014). A survey on server-side approaches to securing web applications. *ACM Computing Surveys (CSUR)*, 46(4):1–29.

Loebens, M. d. C. (2022). Detectando ataques broken object level authorization em apis rest usando dependência produtor-consumidor. Master's thesis.

Menemencioğlu, O. and Orak, İ. M. (2017). A simple solution to prevent parameter tampering in web applications. In *Threat Mitigation and Detection of Cyber Warfare and Terrorism Activities*, pages 1–20. IGI Global.

OpenAPI Initiative (2020). Openapi specification 3.0.3. `https://swagger.io/specification/`. Accessed: 2022-02-13.

OpenAPI Initiative (2023). Best Practices. [Online; `https://learn.openapis.org/best-practices.html`]. Accessed on October 01, 2024.

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 30/10/2020.

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 30/10/2020.

OWASP (2021). Top 10 - 2021 the ten most critical web application security risks. `https://owasp.org/Top10/`. Acesso em 30/01/2022.

OWASP (2023). Owasp top 10 api security risks – 2023 - owasp api security top 10. `https://owasp.org/API-Security/editions/2023/en/0x11-t10/`. Acesso em 18/06/2023.

Petryshyn, B. (2024). *Large language models for OpenAPI definition autocompletion*. PhD thesis, Kauno technologijos universitetas.

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

Postman (2023a). 2022 state of the api report | api technologies. `https://www.postman.com/state-of-api/api-technologies/#api-technologies`.

Postman (2023b). Raml and api blueprint: where are they now? | postman blog. `https://blog.postman.com/raml-and-api-blueprint-where-are-they-now/`.

Postman (2024). 2023 State of the API Report. [Online; `https://www.postman.com/state-of-api/api-global-growth`]. Accessed on October 01, 2024.

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

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164.

Sadqi, Y. and Maleh, Y. (2021). A systematic review and taxonomy of web applications threats. *Information Security Journal: A Global Perspective*, pages 1–27.

Salt Security (2022). State of api security q1 2022 - api security trends. `https://salt.security/api-security-trends`.

Salt Security (2024). State of api security report 2024. `https://salt.security/blog/increasing-api-traffic-proliferating-attack-activity-and-lack-of-maturity-key`

Salt Security (2025). State of api security report 2024. `https://content.salt.security/state-api-report.html`.

Schoenborn, J. M. and Althoff, K.-D. (2021). Detecting sql-injection and cross-site scripting attacks using case-based reasoning and seasalt. In *LWDA*, pages 66–77.

SmartBear (2020). The State of API Report 2020 | SmartBear. Technical report.

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.

Van der Aalst, W. (2016). *Process Mining: Data Science in Action*. Springer Berlin Heidelberg, Berlin, Heidelberg.

van Zelst, S. J., Bolt, A., Hassani, M., van Dongen, B. F., and van der Aalst, W. M. (2019). Online conformance checking: relating event streams to process models using prefix-alignments. *International Journal of Data Science and Analytics*, 8(3):269–284.

Wu, X. and Zhu, H. (2016). Formalization and analysis of the rest architecture from the process algebra perspective. *Future Generation Computer Systems*, 56:153–168.

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