International Journal of Information Security (2025) 24:83
https://doi.org/10.1007/s10207-024-00970-5

REGULAR CONTRIBUTION

Automated broken object-level authorization attack detection in REST APIs through OpenAPI to colored petri nets transformation

Ailton Santos Filho[1] · **Ricardo J. Rodríguez**[2] · **Eduardo L. Feitosa**[1]

Abstract

The representational state transfer architectural style (REST) specifies a set of rules for creating web services. In REST, data and functionality are considered resources, accessed, and manipulated using a uniform, well-defined set of rules. RESTful web services are web services that follow the REST architectural style and are exposed to the Internet using RESTful APIs. Most of them are described by OpenAPI, a standard language-independent interface for RESTful APIs. RESTful APIs are continuously available on the Internet and are therefore a common target for cyberattacks. To prevent vulnerabilities and reduce risks in web systems, there are several security guidelines available, such as those provided by the Open Web Application Security Project (OWASP) foundation. A common vulnerability in web services is broken object level authorization (BOLA), which allows an attacker to modify or delete data or perform actions intended only for authorized users. For example, an attacker can change an order status, delete a user account, or add unauthorized data to the server. In this paper, we propose a transformation from OpenAPI to Petri nets, which enables formal modeling and analysis of REST APIs using existing Petri net analysis techniques to detect potential security risks directly from the analysis of web server logs. In addition, we also provide a tool, named `Links2CPN`, which automatically performs model transformation (taking the OpenAPI specification as input) and BOLA attack detection by analyzing web server execution traces. We apply it to a case study of a vulnerable web application to demonstrate its applicability. Our results show that it is capable of detecting BOLA attacks with an accuracy greater than 95% in the proposed scenarios.

Keywords RESTful web services · OpenAPI · Colored Petri nets · Security analysis · Vulnerabilities · Broken access control · Web application security

1 Introduction

A common software architectural style for creating web services today is *representational state transfer* (REST), which specifies a set of rules (constraints) on web services [17]. In REST, data and functionality are considered resources and are accessed and manipulated by using a uniform and well-defined set of rules. REST is constrained to a client/server architecture (the client is the one requesting the resources, while the server has the resources itself) and is designed to use a stateless communication protocol (typically, HTTP). Web services that follow the REST architectural style are known as *RESTful web services* [41].

RESTful web services expose their services to the Internet using Application Programming Interfaces (called RESTful or REST APIs). These API types are also commonly used when exposing internal interfaces in microservice architectures [30]. Many popular web services, such as X, Facebook, or Instagram, to name a few, have a REST API to allow users and developers connect and interact with their services in a simple and fast way. When creating an REST API, it is important to follow industry standards as a way to ease development and increase customer adoption. Today, most REST APIs are described with OpenAPI [32], which

✉ Ricardo J. Rodríguez
rjrodriguez@unizar.es

Ailton Santos Filho
assf@icomp.ufam.edu.br

Eduardo L. Feitosa
efeitosa@icomp.ufam.edu.br

[1] Institute of Computing, Federal University of Amazonas (UFAM), Av. General Rodrigo Octavio Jordão Ramos, 1200, Coroado I, Manaus, AM 69067-005, Brazil

[2] Instituto de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza, Edificio I+D+i, Calle Mariano Esquillor s/n, 50018 Zaragoza, Spain

Published online: 21 February 2025

defines a standard language-independent interface for RESTful APIs. Many frameworks for building REST APIs (such as Falcon, Flask, or Tornado, to name a few) include OpenAPI support.

As Web services are continuously available on the Internet, they are a common target for cyberattacks [8]. Most attacks fall into the category of *Structured Query Language injection* (SQLi) [31] and *cross-site scripting* (XSS) [15], although other attacks, such as *denial of service* and authentication or session management, are also feasible [43]. In this regard, there are several major players in the information security industry that provide secure design and programming guidelines to prevent vulnerabilities and reduce risks in web systems. One of these major players is the *Open Web Application Security Project* (OWASP) foundation,[1] which provides a security methodology used as a benchmark for web application security audits. In particular, they periodically publish the top 10 most critical application security risks, outlining mechanisms to minimize them.[2]

Although the textual specification of a REST API is not well suited for formal methods (such as computer aided verification), the standardization of its specification opens an exciting path for automated tools to analyze and test REST APIs for correctness. Following this direction, in this paper we investigate the automatic transformation of a REST API specified by OpenAPI [32] to Petri nets [33], which is a mathematical model commonly used to represent distributed, concurrent, or parallel systems. Obtaining a formal model as a Petri net allows us to take advantage of all existing Petri net analysis techniques and detect possible security risks directly in the specification.

The contributions of this paper is twofold. First, we propose a transformation from OpenAPI to Petri net. To do this, we study the latest OpenAPI specification that targets REST APIs (namely, version 3.1.0) and model its parts using Colored Petri nets (CPNs) [23]. CPNs are an extension of the classical Petri net formalism with data, time, and hierarchy, providing a well-defined formalism to represent systems with concurrency and data flow. OpenAPI specifications, on the other hand, describe the structural and behavioral aspects of REST APIs. By transforming OpenAPI into a CPN model, we leverage the CPN formalism to systematically analyze and detect potential vulnerabilities, such as broken object level authorization (BOLA). This transformation ensures that key API components, such as endpoints, operations, and access controls, are preserved in a formal model, enabling robust analysis. Specifically, these models are built into a single CPN that is suitable for analysis with specific tools such as CPN tools [39]. In addition, we provide a tool, dubbed `Links2CPN`, that automatically performs the model transformation. Second, we apply our tool on different case studies of vulnerable web applications to show its applicability. In particular, we focus on the first OWASP Top 10 2023 security risk, related to broken access control [3] and manipulation of data sent within the requests. Using an event log based on the Common Logfile Format (CLF),[3] with the addition of the authorization header, as well as the request and response bodies (obtained from a webserver that runs a web app that conforms to the given REST API specification) and its corresponding CPN model, we show how our tool can easily detect this vulnerability in the CPN obtained from the OpenAPI specification by analyzing the event log.

The rest of this paper is organized as follows. Section 2 gives some background on REST APIs, the OpenAPI specification and CPN. Section 3 presents the running example that is used throughout the paper. We then describe our methodology and our tool, `Links2CPN`, in Sect. 4. Section 5 introduces evaluation and the limitations of our approach. We first test our approach on the running example, then on a user-based evaluation, and finally on a real-world vulnerable software to validate it. Section 6 discusses related work. Section 7 concludes the paper along with future work.

2 Background

In this section, we first provide some background on REST APIs, the OpenAPI specification, and then Colored Petri nets.

2.1 REST APIs

REST is a style of software architecture style introduced in [17], which defines constraints for Web applications to respond to service requests from their clients, namely: *Client–Server*, *Stateless*, *Cache*, *Uniform Interface*, *Layered System* and *Code-On-Demand*. Web services that follow all of the constraints are called RESTful Web services, and their programmatic interfaces are known as REST APIs [41].

Specifically, the *client–server* constraint establishes a separation of duties, ensuring that client and server applications evolve independently without any dependencies on each other. *Stateless* means that each client request must include all information required for the server to process it, without relying on any stored context from previous interactions. *Cache* implies that caching must be applied to resources when appropriate, and those resources must explicitly indicate their caching capability. The *uniform interface* states that implementations must be decoupled from the services they provide. To achieve this, 4 interface constraints are defined: resource identification, resource manipulation through rep-

[1] Accessible in https://owasp.org/.

[2] Accessible in https://owasp.org/www-project-top-ten/.

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

resentations, self-descriptive messages, and hypermedia as the engine of application state. The *layered system* allows an architecture to be composed of hierarchical layers by restricting the behavior of components such that each component interacts only with the immediate layer it is connected to, with no visibility into layers beyond. Finally, *code-on-demand* allows client functionality to be extended by downloading and executing code. The latter is the only restriction that is optional.

In [24], the general rules that REST APIs follow are presented through examples. Nouns are used instead of verbs for resources. For example, instead of `/getBasketById`, `/basket/6` is used. Each resource is identified by an URL. For instance, `/basket` represents a basket collection resource, while `/basket/6` represents a specific item. Create, read, update, and delete operations are directly mapped to the HTTP methods `POST`, `GET`, `UPDATE` and `DELETE`, respectively.

According to [24], the most basic resource that is required for every API is documentation. One of the most widely used standards for documenting and describing APIs is the OpenAPI specification [38], which is described in detail below.

2.2 OpenAPI specification

OpenAPI specification (OAS) [32] defines a widely accepted, vendor-independent, language-independent open specification for the description of RESTful APIs. It allows both humans and computers to discover and understand the capabilities of a service without the need to access source code, additional documentation, or even inspect of network traffic. An OAS-compliant OpenAPI document is itself a structured object, which can be represented in JSON or YAML format. According to OAS manteiners [35], it is preferable to write the API documentation (i.e., the specification) and then write the code (i.e., *Design-first* approach). However, it is possible to first implement the API and then create the documentation using tools that transform code comments and annotations into documentation, or even by writing it manually (*Code-first* approach).

An OpenAPI document, compliant with the OpenAPI Specification (OAS), uses a structured set of fields to describe a REST API. These fields are categorized into *fixed fields*, which have predefined names, and *pattern fields*, which use regular expressions for naming. Key fixed fields include `openapi`, `info`, `servers`, `paths`, `components`, `security`, `tags`, `jsonSchemaDialect`, `webhooks`, and `externalDocs`. Among these, the `paths` field is of particular interest to our work, as it lists the available endpoints of the API and their operations.

Each path entry corresponds to a `PathItem` object, which describes the HTTP methods [16] (e.g., GET, POST) and their details, encapsulated in `Operation` objects. These objects define the operation parameters, request payloads, server responses, and additional metadata. Parameters, specified within the `parameters` field, detail input requirements such as name, location, and type, while the `requestBody` field describes the expected structure and format of the HTTP request payload.

For responses, the `responses` field map HTTP status codes [16] to their corresponding results, including descriptions, payload structures, and examples. The response object can optionally include `links` describing how responses relate to other operations, making it easier to navigate between API functionalities.

This structured approach allows OpenAPI to comprehensively represent interactions and data flows within a REST API, providing a foundation for tools and methodologies to analyze the API behavior.

The approach we present here for the transformation from OpenAPI to CPNs requires having an OpenAPI specification with `links`. Unfortunately, this field is not widely used in general, which may be a limitation to the importance of our approach. In this regard, we have manually analyzed 1955 OpenAPI specifications from the open source API directory APIs.guru,[4] as in [27], and detected only 9 using `links` (listed in [44]). However, we need a way to formally relate responses and other operations in the web service under analysis so thus we can relate the components of the Petri net that we iteratively create when parsing the OpenAPI specification. Furthermore, the OpenAPI document must accurately represent the corresponding API, including error cases, otherwise, the performance of the approach will be degraded. Therefore, similar to [5, 20], we assume that development teams can update their OpenAPI specifications if they want to detect BOLA vulnerabilities with our approach. Alternatively, if they follow the *code-first* approach, newer tools already support creating OAS with `links` from comments or annotations in the source code [36, 46]. Similarly, when *design-first* approach is followed, artificial intelligence (AI) tools can be helpful to automate the work of creating and improving OpenAPI specifications [26, 48]. In this sense, according to a survey of 40, 261 API developers and professionals [38], 60% of them said they were using AI tools in API development, and 28% said they were specifically using such tools to generate API documentation.

2.3 Colored Petri nets (CPNs)

Colored Petri nets [23] are a well-known formalism for the design and analysis of concurrent systems. CPNs are supported by *CPN tools* [39], which is a tool that allows us to easily create, edit, simulate, and analyze CPNs. The following assumes that the reader is familiar with the basics of Petri

[4] Accessible in https://apis.guru/.

nets. First, we give an informal introduction to Petri nets and Colored Petri nets. Next, we provide a formal definition of the CPN formalism. For a full description of the CPN formalism, the reader is referred to [23].

Petri nets [33] are a mathematical and graphical formalism that easily represent common characteristics of computing systems, such as branching, sequencing, or concurrency, to name a few. Roughly speaking, a Petri net is a bipartite graph of *places* and *transitions* joined by *arcs*, describing the flow of a system with concurrency and synchronization capabilities. Graphically, places are represented by circles, transitions by rectangles, and arcs are represented by directed arrows. An arc may have an integer inscription, indicating the *weight* of the arc. A place can contain *tokens*, graphically represented by black dots (or by a number) within the place and denoted as the *marking* of the place. When all input places of a transition t are marked with a number of tokens equal or greater than their weights, t is said to be *enabled*. An enabled transition can *fire*, resulting in a new marking obtained by removing tokens from input places and setting tokens to output places. The number of tokens removed/set in each place corresponds to the weight of the arc that connects each place with the transition.

A CPN [23] is an extension of Petri nets, where places have an associated color set (a data type) that specifies the set of token colors allowed at that place. That is, each token at a place in a CPN has an attached data value (color) that matches the corresponding color set of the place. For instance, a place can have as its color set the integer set `INT`, the Cartesian product untimed color set `INT2 = INTxINT`, or a singleton color set (`UNIT`), which contains a single empty value denoted by *unit*. Other complex data types can also be defined by using data types constructors, such as *list*, *union*, and *record*. Together, the number of tokens and the token colors in the individual places represent the *state* of the system [19]. According to [47], CPNs are the most widely used Petri net-based formalism that can deal with issues related to data and time.

More formally:

Definition 1 [23] A Colored Petri net (CPN) is a tuple $\langle P, T, A, V, G, E, \pi \rangle$, where[5].:

- P is a finite set of *places*, with colors in a set Σ. We denote the color set of a place p by Σ_p.
- T is a finite set of *transitions* ($P \cap T = \emptyset$).
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of directed *arcs*. PT-arcs are those connecting places with transitions ($P \times T$), while TP-arcs connect transitions with places ($T \times P$).
- V is a finite set of *typed variables* in Σ, i.e., $Type(v) \in \Sigma$, for all $v \in V$.
- $G : T \longrightarrow EXPR_V$ is the *guard function*.[6].
- $E : A \longrightarrow EXPR_V$ is the *arc expression function*, which assigns an expression to each arc. Arc expressions evaluate to multisets of the set of colors of the place connected to the arc. For any transition $t \in T$, the arc expressions of the PT-arcs connected to t are called *PT-arc expressions of* t (respectively, for TP-arcs).
- $\pi : T \longrightarrow \mathbb{N}$ is the *priority function*, which assigns a priority level to each transition. The priority level of a transition t_i has a higher priority level than a transition t_j iff $\pi(t_i) > \pi(t_j)$.

Definition 2 (*Marking*) Given a CPN $N = \langle P, T, A, V, G, E, \pi \rangle$, a *marking* M of N is defined as a function $M : P \longrightarrow \mathcal{B}(\Sigma)$, such that $\forall p \in P,\ M(p) \in \mathcal{B}(\Sigma_p)$, i.e., the marking of p must be a multiset of colors in Σ_p (which can be empty).

Definition 3 (*Marked CPN*) A *marked CPN* (MCPN) is then defined as a pair $\langle N, M \rangle$, where N is a CPN and M is a marking of it.

We define the semantics for MCPNs as in [23], taking into account that transitions have associated priorities. In this paper, we assume that all transitions have a priority level of 1 (i.e., $\forall t \in T, \pi(t) = 1$). We first introduce the notion of *binding*, then the *enabling condition*, and finally the *firing rule* for MCPNs.

Definition 4 (*Bindings*) Let $N = \langle P, T, A, V, G, E, \pi \rangle$ be a CPN. For any transition t, $Var(t)$ denotes the set of variables that appear in the PT-arc expressions of t. So, a *binding* of a transition $t \in T$ is a function b that maps each variable $v \in Var(t)$ into a value $b(v) \in Type(v)$. $B(t)$ will denote the set of all possible bindings for $t \in T$. For any expression $e \in EXPR_V$, $e\langle b \rangle$ will denote the evaluation of e for the binding b. A *binding element* is then defined as a pair (t, b), where $t \in T$ and $b \in B(t)$. The set of all binding elements is denoted by BE.

Definition 5 (*Enabling condition*) Let $\langle N, M \rangle$ be a MCPN. We say that a binding element $(t, b) \in BE$ is *enabled* at marking M when the following conditions are fulfilled:

1. The guard of t evaluates to true for binding b: $G(t)\langle b \rangle = true$.
2. For all $p \in {}^\bullet t$, $E(p, t)\langle b \rangle$ is included in $M(p)$, and these tokens on $M(p)$ have a timestamp less than or equal to the current time, i.e., we have in $M(p)$ enough available tokens to fire t with the binding b.

[5] We use the classical Petri net notation to denote the precondition ${}^\bullet x$ and postcondition $x^\bullet$ of both places and transitions: $\forall x \in P \cup T$: ${}^\bullet x = \{y \mid (y, x) \in A\}$; $x^\bullet = \{y \mid (x, y) \in A\}$

[6] $EXPR_V$ denotes expressions built using the variables in V, with the same syntax supported by *CPN Tools*, which assigns a Boolean expression to each transition, i.e., $Type(G(t)) = Bool$

3. There is no other binding element $(t', b') \in BE$ fulfilling the previous conditions such that $\pi(t') < \pi(t)$.

Definition 6 *(Firing rule)* Let $N = \langle P, T, A, V, G, E, \pi \rangle$ be a CPN, M a marking of N, and $(t, b) \in BE$ an enabled binding element at marking M.

The firing of (t, b) has the following effects on M:

- For any $p \in^{\bullet} t$, the tokens in $E(p, t)\langle b \rangle$ are removed from $M(p)$.
- For any $p \in t^{\bullet}$, the tokens in $E(t, p)\langle b \rangle$ are produced in $M(p)$.

3 Running example

This section introduces the running example used in the rest of paper to illustrate how our OpenAPI to CPN transformation approach. Let us consider as an example a simple Web application (client and server) where a user logs in and checks their shopping cart. This example is taken from the OWASP Juice Shop project,[7] a cybersecurity education project with an insecure Web application. This project is commonly used in security training, awareness demos, security competitions, and to test security tools.

The UML sequence diagram in Fig. 1 illustrates the interaction between the REST API client and the server. In the beginning, the client application initiates communication by sending a POST request with the credentials to the `/login` endpoint of the server application. The server application responds with the information related to that user, such as the basket ID (*bid*) and *token*. With the information received, the client application sends a new request to the server for the `/basket/{bid}` endpoint and again, the server application responds with the requested information.

The REST API in this example is vulnerable to *Broken Object Level Authorization*, which is ranked #1 in the OWASP API Security Top 10 2019.[8] Exploiting this vulnerability is quite simple. An attacker can simply send a request to the `/basket/{bid}` endpoint with a different bid than the one given by the server in a previous request. For example, if the client requests `/basket/2` in Fig. 1, the server will respond with the another user's shopping basket information.

The excerpt of OpenAPI specification that we will use as running example is shown in Listing 5 (see the appendix). In the rest of this paper, we use this running example to show how our model transformation approach can detect Broken Object Level Authorization (BOLA) attacks.

[7] Accessible in https://owasp.org/www-project-juice-shop/.

[8] Accessible in https://owasp.org/www-project-api-security/.

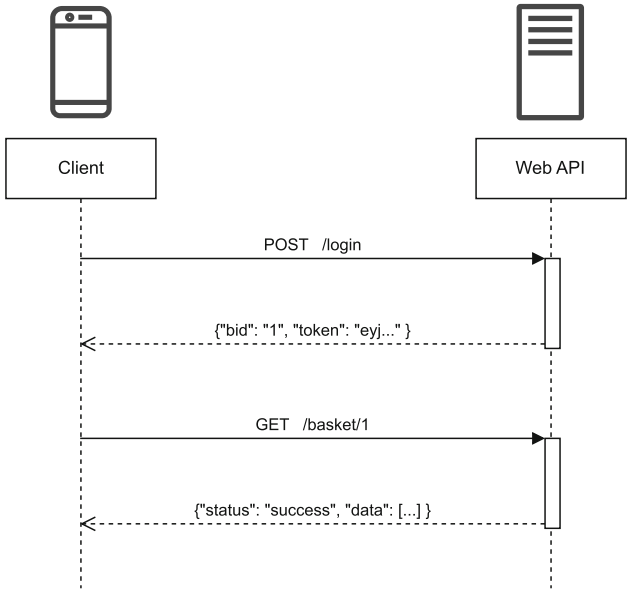


Fig. 1 A simple WebAPI interaction

4 Methodology and the tool `Links2CPN`

In this section, we first briefly introduce the methodology we follow to detect BOLA attacks against REST APIs. Then we introduce `Links2CPN`, a Python3-based tool that analyzes information system event logs and detects BOLA attacks using our methodology.

Figure 2 sketches the steps of our approach. First, we transform the OpenAPI specification in its corresponding Colored Petri net, applying the algorithms described in Sect. 4.1. We call this step MODEL- TO- MODEL TRANSFORMATION. We then apply process mining techniques (namely, conformance checking techniques), combining the CPN obtained in the previous step with the JSON event logs collected from the web servers running the application that implements the given OpenAPI specification as initial input. As a result, we get an error log file highlighting the detection of BOLA attacks. In what follows, we explain each step of the methodology in more detail.

4.1 Model-to-model transformation: from OpenAPI to CPN

According to [47], depending on the input data and the questions that need to be addressed, a suitable model and abstraction level should be chosen. In this sense, the Petri net structuring directly aligns with the hierarchical structure of OpenAPI. API paths are represented as flows within the Petri net, where transitions model operations and places connect these transitions, representing the flow of operations. This alignment preserves the semantics of API interactions while allowing for formal verification of properties within models.

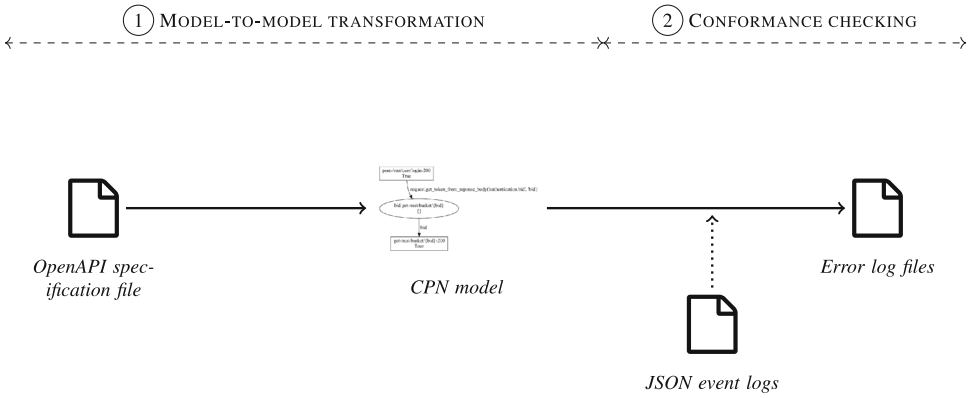


Fig. 2 Methodology to detect BOLA attacks in OpenAPI specifications

In this paper, we propose that CPNs provide a suitable abstraction and model to represent REST APIs for several reasons. First, CPNs integrate flow and data flow into a single model, capturing the dynamic nature of API interactions. Second, they formally model concurrent, distributed, and event-driven systems, which align with the nature of REST API interactions, as demonstrated in the literature [10, 28, 29]. Third, CPNs support conformance-checking algorithms [7], which are essential for verifying API behaviors. Finally, their visual representation is directly equivalent to their mathematical definition, allowing for the analysis of both graphical representation without loss of information [23].

Furthermore, by using CPNs, we introduce "coloring" to represent essential aspects such as user roles, resource identifiers, and system states. This capability is particularly important for detecting BOLA vulnerabilities. Below, we detail the process of transforming a valid OpenAPI document into a CPN and illustrate it using the following running example.

Algorithm 1 describes the steps for the model transformation. As an input, it needs an OpenAPI specification `doc`. As an output, it generates a Colored PetriNet $\mathcal{C} = \langle P, T, A, V, G, E, \pi \rangle$ that represents `doc`.

Algorithm 1: Creation of a CPN from an OpenAPI specification.

Input: An OpenAPI specification `doc`.
Output: A Colored Petri net $\mathcal{C} = \langle P, T, A, V, G, E, \pi \rangle$.

1 `/* Create all the transitions related to paths */`
2 $\mathcal{C}$ = createTransitionsForPaths(`doc`) (see Algorithm 2);
3 `/* Connect transitions related to paths */`
4 $\mathcal{C}$ = connectTransitionsForPaths(`doc`, $\mathcal{C}$) (see Algorithm 3);
5 Remove disconnected transitions and places in $\mathcal{C}$;
6 **return** $\mathcal{C}$;

First, we create all the transitions related to paths in the OpenAPI specification, as indicated by Algorithm 2. Basically, this algorithm iterates over the paths and for each operation on the path, it iterates over the responses (lines 6–17), creating a new transition t that represents such a response (line 8). The `operationId` and `HTTPMethod` of an operation, as well as its `HTTPStatusCode`, are taken as variables to make up the tagging of t. For each transition t, we also create a preset place $p = {}^{\bullet} t$. Also, the arc connecting p and t is labeled by the name of the operation parameter (if any). The second part of the algorithm is related to the connection of path-related transitions (see Algorithm 3). We iterate over the paths and for operation on the path, we iterate over the responses (lines 5–14 of Algorithm 3). Each response is analyzed and if it is linked to any other path, the transitions representing both paths are connected via the preset place of the transitions that represents the target path.

Algorithm 2: Creation of CPN transitions representing paths in an OpenAPI specification.

Input: An OpenAPI specification `doc`.
Output: A Colored Petri net $\mathcal{C} = \langle P, T, A, V, G, E, \pi \rangle$.

1 Create an empty CPN $\mathcal{C} = \langle P, T, A, V, G, E, \pi \rangle$;
2 **foreach** `path` $\mathcal{P} \in$ `doc` **do**
3 **foreach** `operation` $\mathcal{O} \in \mathcal{P}$ **do**
4 `operationId` ← Get `operationId` from $\mathcal{O}$;
5 `HTTPMethod` ← Get `HTTPmethod` from $\mathcal{O}$;
6 **foreach** `response` $\mathcal{R} \in \mathcal{O}$ **do**
7 `HTTPStatusCode` ← Get `HTTP-status-code` from $\mathcal{R}$;
8 Create a transition t, tagging it as `HTTPMethod-path-HTTPStatusCode` and with id `operationId`;
9 Add transition t to $\mathcal{C}$ (i.e., $T = T \cup \{t\}$);
10 Create a place $p = {}^{\bullet} t$;
11 Add place p to $\mathcal{C}$ (i.e., $P = P \cup \{p\}$);
12 **end foreach**
13 **if** $\exists$ `parameters` $\mathcal{U} \in \mathcal{O}$ **then**
14 `paramName` ← Get `name` from $\mathcal{U}$;
15 Label the output arc of p as `paramName`;
16 **end if**
17 **end foreach**
18 **end foreach**
19 **return** $\mathcal{C}$

Algorithm 3: Connection of CPN transitions representing paths in an OpenAPI specification.

Input: An OpenAPI specification `doc` and a Colored Petri net $\mathcal{C} = \langle P, T, A, V, G, E, \pi \rangle$.
Output: An updated Colored Petri net $\mathcal{C} = \langle P, T, A, V, G, E, \pi \rangle$.

1 **foreach** `path` $\mathcal{P} \in$ `doc` **do**
2 **foreach** `operation` $\mathcal{O} \in \mathcal{P}$ **do**
3 `operationId` ← Get `operationId` from $\mathcal{O}$;
4 `HTTPMethod` ← Get `HTTPmethod` from $\mathcal{O}$;
5 **foreach** `response` $\mathcal{R} \in \mathcal{O}$ **do**
6 Let $t \in \mathcal{T}$ be the transition with id `operationId`;
7 **if** $\exists$ `links` $\mathcal{L} \in \mathcal{R}$ **then**
8 `operationId`$'$ ← Get `operationId` from $\mathcal{L}$;
9 `parameter` ← Get `parameters` from $\mathcal{L}$;
10 Let $t' \in \mathcal{T}$ be the transition with id `operationId`$'$;
11 Create an arc connecting $t^\bullet =^\bullet t'$;
12 Label the input arc of $t^\bullet$ with `parameter` ;
13 **end if**
14 **end foreach**
15 **end foreach**
16 **end foreach**
17 **return** $\mathcal{C}$

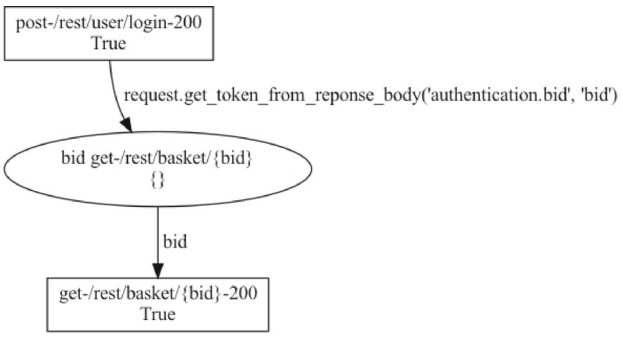


Fig. 3 Transformation from OpenAPI specification to CPN

Finally, we remove all transitions that do not have connections to places (line 5 of Algorithm 1), and we return the updated CPN (line 6).

Applying Algorithm 1 to the OpenAPI specification shown in Appendix A, we will obtain a CPN as the one depicted in Fig. 3. Note that the transition "`post-/rest/user/login/-200`" is created by the path `/rest/user/login`, the operation `post`, and the response code 200. Similarly, the transition "`get-/rest/user/bid/-200`" is created by the path `/rest/basket/{bid}`, the operation `get`, and the response code 200. The preset arc is labeled with `bid`, given the `parameter name` of the operation. Finally, both transitions are joined as the response of the path `/rest/user/login` is linked to `getBasketById`, which is represented by the transition `get-/rest/user/bid/-200`.

To show the generality of our transformation approach, we apply Algorithm 1 to a more complex example. In particular, we consider the lightweight open-source note-taking service called `Memos`.[9] The API of `Memos` had a BOLA vulnerability related to note archiving operation that was found and fixed in 2022 [21]. Figure 4 shows the CPN obtained after the automatic transformation from `Memos` OpenAPI specification to CPN performed by our algorithm. Let us remark that before applying the transformation we have slightly modified the original source code to recreate the vulnerability.[10]

Similar to the previous example, the transition `post-/api/v1/memo-200` is created by the path `/api/v1/memo`, the operation `post`, and the response code 200. Likewise, the transition `get-/api/v1/memo-200` is created by the path `/api/v1/memo`, the operation `get`, and the response code 200. Finally, transitions starting with `patch-/api/v1/memo/memoId` are created by the path `/api/v1/memo/memoId`, the operation `patch`, and the respective response code 200, 400, 401, 404, 500. These three endpoints comprise the flows where the user can list their notes, create new notes, and archive their already created notes, respectively.

4.2 Conformance checking: detecting broken object level authorization attacks

Conformance checking is a topic in process mining that refers to the analysis of the relation between the intended behaviour of a process and its recorded behaviour observed during execution. In practice, conformance checking is a family of techniques and algorithms that relates two main inputs, process models and event logs, providing methods to compare and analyse observed instances of a process against its model [6, 47]. An example of a comparison is whether a process is being executed as documented in a model.

Once we have a CPN modeled from an OpenAPI specification and a set of HTTP requests and responses (i.e., a JSON event logs), we use process mining techniques to verify the correctness of the request and response pairs collected in a system trace. In particular, we work with information system event logs that consist of recorded traces, describing the activities executed and the resources involved (for example, users, data objects, requests, and responses). We then use these logged traces to apply the conformance checking algorithm presented in [7]. Basically, this algorithm works like a replay algorithm. By replaying each trace of an event log on top of a CPN, this algorithm can discover control flow and data flow deviations due to unavailable resources, rule violations, and differences between modeled and actual resources. For completeness, we refer the reader to [7] to get a more detailed idea of the algorithm. Despite their simplicity,

[9] Accessible in https://github.com/usememos/memos.

[10] The vulnerable source code is accessible in https://github.com/ailton07/memos-with-BOLA/blob/main/api/v1/openapi.yaml.

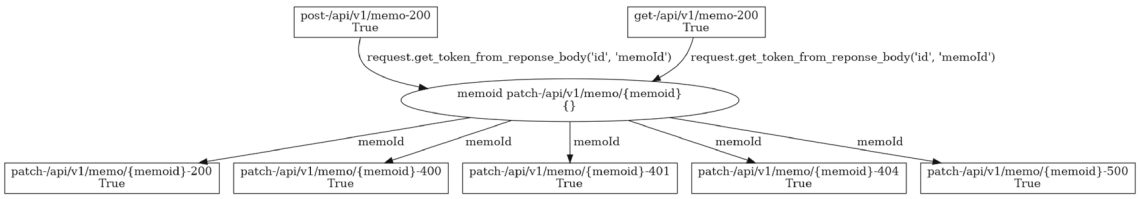


Fig. 4 Transformation from `Memos` OpenAPI specification to CPN

token-based replay algorithms have became the standard not only for conformance checking, but also for decision mining and performance analysis, among others [6].

To use it, we need to create an event log based on the request-response pairs that contains the CPN related information. In particular, this file must contain the URL, URL parameters, HTTP methods, status code, client IP and HTTP header, timestamp, request body, and response body. To make it easier to perform programmatic operations on these logs, we parse the original webserver logs to generate a JSON format file containing the required information. Listing 1 shows an example of a single event log using this JSON format. Note that this processing step can be performed on any information system event log, such as `Apache2` or `nginx`, and only minor adaptations are necessary to parse the original event log. An example of the modifications that need to be made to a Web server in order to generate event logs with the format discussed is available online.[11]

To improve the flexibility and adaptability of our tool, we are exploring ways to enhance the log processing component. Specifically, we plan to introduce a modular approach incorporating the *Strategy* behavioral pattern [18], allowing for interchangeable event log processing strategies based on the input log format. This design would allow the tool to seamlessly handle different server environments without requiring manual intervention to reformat logs. Additionally, we are considering using the *Adapter* structural pattern [18] to standardize log formats for internal processing, further decoupling the tool from the particularities of any particular server log. This architectural refinement aims to increase the applicability and ease of integration of our tool into various HTTP server configurations.

Listing 1 JSON event log example.

```
{
    "timestamp":"2022-11-01T22:35:33.107Z",
    "ip":"::1",
    "message":"GET /rest/user/whoami 200 4ms",
    "method":"GET",
    "uri":"/rest/user/whoami",
    "requestBody":{
    },
    "responseBody":{
       "user":{
          "id": "123"
       }
    },
    "statusCode":200,
    "headerAuthorization":"Bearer
        eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9..."
}
```

Given a file of event logs and the CPN associated with the corresponding OpenAPI specification, we can now run the conformance checking algorithm presented in [7]. As commented above, this replay algorithm allows us to follow the evolution of the system, checking if the firing of transitions enabled in the CPN is correct. When firing a transition is feasible, it means that the observed behavior is correct and expected. On the contrary, there are two possible reasons why the firing of a transition is not possible: (i) there are no tokens in the input places of the transition; or (ii) the token at the input places of the transition has a different value than expected. The first case corresponds to a control flow violation and is associated with an expected behavior of the system. On the other hand, the second case corresponds to a data flow violation. We assume that this behavior is associated with a BOLA attack, and therefore detecting a data flow violation at the CPN level allows us to detect BOLA attacks in the OpenAPI specification.

The performance overhead introduced by logging is minimal, and especially when considering the security improvements it offers, is acceptable in most cases. Logging each request, in fact, is a standard practice in web systems, such as Apache HTTP Servers, which use CLF by default. Our approach extends the CLF by logging the request and response body, and while this adds some overhead, it can be mitigated using techniques like asynchronous logging or API gateways with built-in logging features. Thus, while there is some overhead, it is consistent with typical logging practices and can be managed efficiently. Furthermore, given the enhanced security that this logging enables, the overhead is a reasonable trade-off.

4.3 Tool support

We develop a tool, called `Links2CPN`, to automatically perform the transformation of the OpenAPI model to Colored Petri nets and apply the conformance checking algorithm presented in [7] to detect attacks on broken access control vulnerabilities [3]. `Links2CPN` is freely accessible in our

[11] See https://github.com/ailton07/juice-shop-with-winston/blob/079ea6d65463b99c0d25a9ad116127575ce96e9a/server.ts#L305.

GitHub [44] and is developed in Python 3 on top of the *Snakes* [37] library, a general-purpose Petri net library that allows the creation, transformation, and net manipulation. Using this library we implement the model transformation and the conformance checking algorithm. In addition to *Snakes*, the tool also uses the *openapi-schema-validator* [12] library to validate and interpret OpenAPI 3.x specifications.

5 Experimental evaluation and limitations

In this section, we first test our approach on the running example presented in Sect. 3. We then run a user-based evaluation to validate our approach. Next, we evaluate a real-world BOLA-vulnerable open source software with our tool with a user-based evaluation. Finally, we describe the threats to validity of our approach.

5.1 Running example evaluation

Similar to other works in the literature [9, 45], we use the OWASP Juice Shop web application to test the accuracy of vulnerability and attack detection solutions. This approach has the advantages of providing a controlled environment without data noise, while ensuring similar characteristics to a real-world environment.

The first step is to instrument OWASP Juice Shop to generate event logs. This code can be found publicly available at GitHub.[13] We then deployed the application to an AWS EC2 instance, a cloud service that allows the creation of a virtual server to run applications on the Amazon Web Services infrastructure, making the application accessible via URL of type http://ec2-XXX-XXX-XXX-XXX.compute-1.amazonaws.com:3000, where the *X* represent the public IP of the running instance.

Now, we reproduce the attacks on the Juice Shop vulnerabilities related to BOLA. The Juice Shop was created with the following vulnerabilities related to BOLA:

> VULNERABILITY 1. *View cart*, which consists of viewing another user's shopping cart;
> VULNERABILITY 2. *Manipulate cart*, which consists of placing a product in another user's shopping cart.

The guide to successfully exploring such vulnerabilities is publicly available.[14] Following these steps, the scenario for the first challenge is that upon loading the website and logging in, a `POST` request is sent to the `/rest/user/login` API endpoint, returning the user's token to the client and user cart identifier (called *bid*). After this, the home page of the website is displayed. One of the requests sent in this process is `GET /rest/basket/6`, where the value 6 refers to the shopping cart identifier obtained in the login request. The attack consists of sending `GET` requests to `/rest/basket/` with a different shopping cart identifier than the one received in the login process after loading the home page. According to the guide, adding or subtracting 1 from its value is enough to explore the vulnerability.

The log file obtained after performing the account registration, login, and exploit attack process is publicly available on our GitHub [44]. This 30-line log file is processed and transformed using the algorithms described in Sect. 4.1. The obtained CPN is then verified with the replay algorithm as explained in Sect. 4.2.

As a result, we got the nets shown in Fig. 5 and the error message shown in Listing 2. Figure 5a shows the initial state of the CPN, being the immediate result of the transformation from OpenAPI to CPN. Likewise, Fig. 5b shows the status of the CPN after processing the line of the log file that represents the request `POST /rest/user/login` with response $bid = 6$. Finally, Fig. 5c shows the CPN status after processing the line that represents the `GET /rest/basket/6` request. Listing 2 ilustrates the processing of the line that represents the request `GET /rest/basket/7`, which is the BOLA attack we performed. This error message briefly states that there is no token available at the CPN's place to make the transition enabled.

Listing 2 Error message received when executing challenge 1

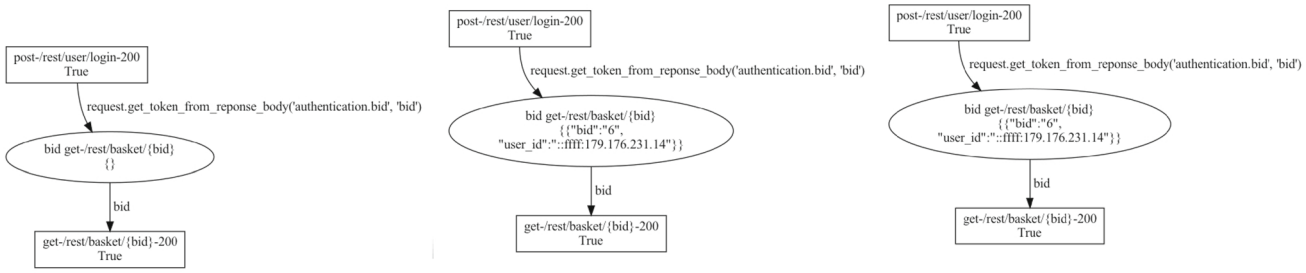
```
Fire error, line 29:  transition not
    enabled for
{
    "bid ->"{
        "bid":"None",
        "user_id":"::ffff:179.176.231.14"
    },
    "request ->"{
        "uri":"/rest/basket/7",
        "method":"GET",
        "user_id": "::ffff:1..."}
}
```

Following the same guide to successfully attacking VULNERABILITY 2, after the website loads and the user logs in, a `POST` request is sent to the `/rest/user/login` API endpoint, returning the user's token to the client's and user's cart identifier (called *bid*), similar to VULNERABILITY 1. After this, the home page of the website is displayed. By clicking on any product and adding it to the shopping basket via the *Add to Basket* button, a `POST` request is sent to the `/api/BasketItems/` API endpoint with the values *ProductId* (product code), *BasketId* (identifier of the user's basket) and *quantity* (quantity of the product to add). The attack consists of sending `POST` requests

[12] Accessible in https://github.com/python-openapi/openapi-schema-validator.

[13] Accessible in https://github.com/ailton07/juice-shop-with-winston.

[14] See https://pwning.owasp-juice.shop/appendix/solutions.html.



(a) Initial state of the CPN

(b) State of the CPN after processing *POST /rest/user/login* (with response $bid = 6$)

(c) State of the CPN after processing *GET /rest/basket/7*

Fig. 5 Result of running the transformation algorithms on the event log for VULNERABILITY 1

to `/api/BasketItems/` with a different shopping cart identifier than the one received at login, in addition to the one already present. According to the guide, adding or subtracting 1 from its value is enough to exploit the vulnerability.

The log file obtained after performing the account registration, login, and exploit attack process is publicly available on our GitHub [44]. Similarly to VULNERABILITY 1, this 23-line log file is processed and transformed using the algorithms described above, and the obtained CPN is verified with the replay algorithm, as explained in Sect. 4.2.

Listing 3 Error message received when executing challenge 2

```
Fire error, Line 21: POST /api/
    BasketItems/ 200 11ms
transition not enabled for
{
    "BasketId ->"{
        "BasketId":"5",
        "user_id":"::ffff:35.199.121.33"
    },
    "request ->"{
        "uri":"/api/BasketItems/",
        "method":"POST",
        "user_id":"::ff...
    }
}
```

As a result, we got the nets shown in Fig. 6 and the error message shown in Listing 3. Figure 6a shows the initial state of the CPN, being the immediate result of the transformation from OpenAPI to CPN. In addition, Fig. 6b shows the CPN status after processing the log file line that represents the `POST /rest/user/login` request with response $bid = 6$. Listing 3 illustrates the processing of the line 21, which represents the `POST /api/BasketItems/` request regarding the BOLA attack we performed. This error message briefly indicates that there is no token available at the CPN place to enable the firing of the transition.

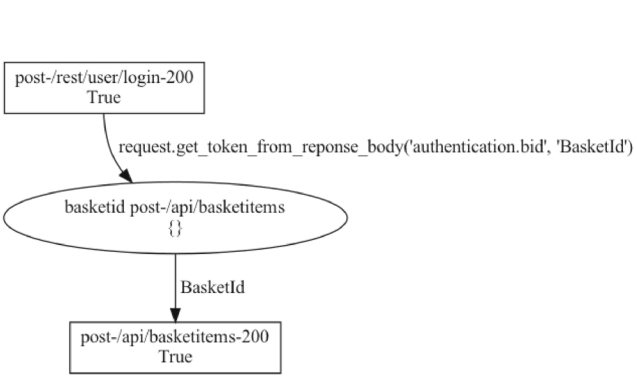
5.2 User-based evaluation

To test the proposed solution in a more realistic scenario, we have carried out a user-based evaluation. For this, we invited Computer Science students from the Federal University of Amazonas, enrolled in the Computer Network Security discipline (ICC303), to participate in a capture-the-flag (CTF) event, where the goal was to actively attempt to exploit VULNERABILITY 1 and VULNERABILITY 2 described in Sect. 5.1. There were 20 students enrolled in the discipline and 15 attending to the end, who were the ones invited for this evaluation. to participate in a capture-the-flag (CTF) event, where the goal was to actively attempt to exploit BOLA vulnerabilities.

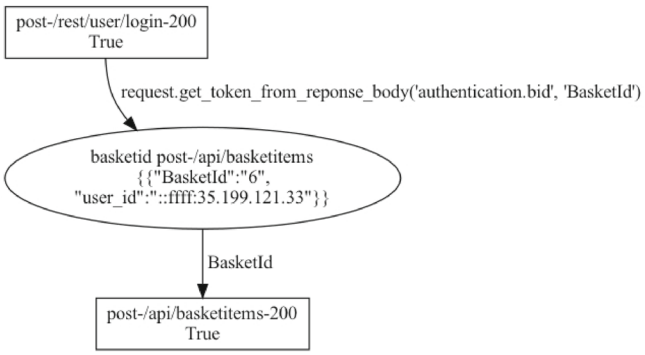
As a test environment, we set up an instance of the Juice Shop application (see Sect. 3) running on an AWS Web server (EC2) from January 25, 2023 to February 8, 2023. We first explain the basics of the BOLA vulnerability and give examples of exploitation. We then explain the Juice Shop app and instruct students to exploit both vulnerabilities in the shopping cart as they see fit, writing reports on the process. We also advise them not to copy solutions from the Internet, but to try to build their own exploits of the vulnerabilities.

In the end, we got 12 attack reports, approximately 4, 000 log lines, 2, 000 requests analyzed, 514 requests associated with VULNERABILITY 1 and 192 requests associated with VULNERABILITY 2. For each of the reports received, we manually analyze the generated logs, especially those associated with registration process, the login process, VULNERABILITY 1, and VULNERABILITY 2. Thanks to this manual analysis, we are able to calculate the number of legitimate requests, the number of of attack attempts and the number of successful attacks. Subsequently, we process the generated logs with our tool (introduced in Sect. 4) to obtain the requests classified as legitimate and as attacks, and compare them with the data obtained through our manual analysis.

As a result, we get the data shown in Table 1. For VULNERABILITY 1 (V1), *Total Requests* refers to the number of requests logged in `GET /rest/basket/`, while for VULNERABILITY 2 (V2), it refers to the number of requests logged in `POST /api/BasketItems/`. Recall the description of both vulnerabilities in Sect. 5.1. *Attack Attempts* refers to the number of requests that we manually classified as attacks



(a) Initial state of the CPN



(b) State of the CPN after processing *POST /rest/user/login* (with response $bid = 6$)

Fig. 6 Result of running the transformation algorithms on the event log for VULNERABILITY 2

Table 1 Summary of data obtained from the attacks

	V1	V2
Total requests	517	192
Attack attempts	101	88
Successful attacks	98	20
Requests classified as attacks	101	88

(failed and successfully) on the cited endpoints, while *Successful Attacks* refers to the number of requests that were manually classified as successful attacks. Finally, *Requests classified as attacks* refers to the number of requests that were classified by our tool as attacks on the cited endpoints.

After that, we processed the logs generated with the proposed solution in order to obtain the list of requests classified as legitimate and as attacks, and compare them with the data obtained through manual analysis. We then obtained the data shown below in Table 1. To VULNERABILITY 1, *Total requests* refers to the number of requests registered to `GET /rest/basket/`, while VULNERABILITY 2 refers to the number of requests registered to `POST /api/BasketItems/`. *Attack attempts* refers to the number of requests that were manually classified as attacks on the cited endpoints. *Successful attacks* refers to the number of requests that were manually classified as successful attacks on the cited endpoints. Finally, *Requests classified as attacks* refers to the number of requests that were classified by the proposed solution as attacks on the endpoints cited.

To express the performance of our approach, we calculate the standard metrics of $Accuracy = \frac{TP+TN}{TP+FP+FN+TN}$, $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, $F1\text{-}score = \frac{2 \cdot Recall \cdot Precision}{Recall+Precision}$. In this setting, false positives (FP) refer to benign requests that are wrongly classified as attacks, false negatives (FN) refer to attack requests that have not been classified as attacks, true positives (TP) refer to benign requests that are correctly classified as benign requests, and true negatives (TN) refer to attack requests that are correctly classified as attack requests.

The confusion matrix is show in Table 2(a). For V1 and V2, we obtain $Accuracy = Precision = Recall = F1\text{-}score = 1$. In both cases, *Precision*, *Accuracy*, *Recall*, and *F1-score* remains constant.

To further investigate the performance metrics between V1 and V2, we create the confusion matrix comparing successful attacks and those classified by our approach. The results are summarized in Table 2(b). In this case, we obtain $Accuracy = 0.99$; $Precision = 0.97$; $Recall = 1$; $F1\text{-}score = 0.98$ for VULNERABILITY 1, while for VULNERABILITY 2 we obtain $Accuracy = 0.64$; $Precision = 0.22$; $Recall = 1$; $F1\text{-}score = 0.37$. In both cases, Recall remains constant, indicating consistency in identifying positive cases (successful attacks). However, the relatively low Precision demonstrates it generates a substantial number of incorrect positive predictions, resulting in a high false positive rate. These results indicate that our approach requires additional heuristics to distinguish between attack attempts and successful attacks, such as considering the status code of the response or requiring manual analysis by an expert.

Our results also show that VULNERABILITY 1 and VULNERABILITY 2 actually have very different *attack attempt/attack success* rates. For VULNERABILITY 1, there were 98 successful attacks out of 101 attack attempts (i.e., 97.02% success rate), while for VULNERABILITY 2, there were 20 successful attacks out of 88 attack attempts (i.e., 22.72% success rate). One explanation for this difference in the success rate of exploiting vulnerabilities is the difference in the difficulty of exploiting the vulnerability. According to the exploit guide of Juice Shop,[15] VULNERABILITY 1 has 2 stars of difficulty, while VULNERABILITY 2 has 3 stars. This may be why

[15] Publicly available at https://pwning.owasp-juice.shop/appendix/solutions.html.

Table 2 Confusion matrices

			Predicted	
			Positive	Negative
(a) Attack attempts versus detections				
Actual		Positive	101	0
	V1	Negative	0	416
		Positive	88	0
	V2	**Negative**	0	104
(b) Successful attacks versus detections				
Actual		Positive	98	0
	V1	Negative	3	416
		Positive	20	0
	V2	Negative	68	104

3 out of 12 students who submitted reports were unable to successfully exploit VULNERABILITY 2.

5.3 Real-world software evaluation

To test the solution proposed in this work with a real application vulnerable to BOLA, we consider the `Memos` application, presented previously (see Sect. 4.1). As before, the first step is to instrument the `Memos` API to generate event logs. The `Memos` code, containing the modifications described below, is publicly available on GitHub.[16] As we have taken the latest version of `Memos` as the code base, where the BOLA vulnerability is already fixed, we had to revert this fix to make the code vulnerable again.

A guide to exploiting the vulnerability is described in [21]. Following these steps, when the website loads and the user logs in, a `GET` request is sent to the `/api/v1/memo` endpoint, returning the notes that belong to the logged-in user. After this, the user can create a note via a `POST` request to the `/api/v1/memo` endpoint, or archive one of the notes from their notes list via a `PATCH` request to `/api/v1/memo/{memoId}`, where `memoId` is the list of note identifiers that the user wants to archive. The attack consists of sending `PATCH` requests to `/api/v1/memo/{memoId}` with values of *memoId* that does not belong to the logged-in user.

The log file obtained after performing the account registration, subsequent login, and exploit attack process is also publicly available in the software repository of our tool [44]. This 11-line log file is processed and transformed using the algorithms described in Sect. 4.1 and the obtained CPN is verified with the replay algorithm as explained in Sect. 4.2.

Figures 7a and 7b shows the CPN obtained after the replay algorithm, while Listing 4 shows the error message obtained. In particular, Fig. 7a shows the initial state of the CPN, being the immediate result of the transformation from OpenAPI to CPN, while Fig. 7b shows the state of CPN after processing the log file line representing the request `GET /api/v1/memo` with response $memoId = 3$ and $memoId = 1$, and the CPN state after processing the log line representing a legitimate request. Likewise, Listing 4 illustrates the processing of line 11 of the log, which represents the `PATCH /api/v1/memo/6` request and corresponds to the BOLA attack we performed. This error message briefly indicates that there is no token available at the CPN site to allow the transition to be fired, thus successfully detecting the occurrence of the attack.

Listing 4 Error message displayed when verifying with the CPN obtained after the transformation the log file containing the BOLA attack carried out.

```
Fire error, line 11:  transition not
    enabled for
{
    "memoId->"{
        "memoId":"None",
        "user_id":"127.0.0.1"
    },
    "request->"{
        "uri":"/api/v1/memo/6",
        "method":"PATCH",
        "user_id": "127.0...."}
}
```

5.4 Threats to validity

This section discusses the threats to validity that we have identified for this study [42] according to construct, internal, external validity, and reliability.

Construct validity

We have conducted controlled experiments that allowed us to fine-tune our tool and measure the metrics of interest. In this sense, no problem should arise from our experiential study.

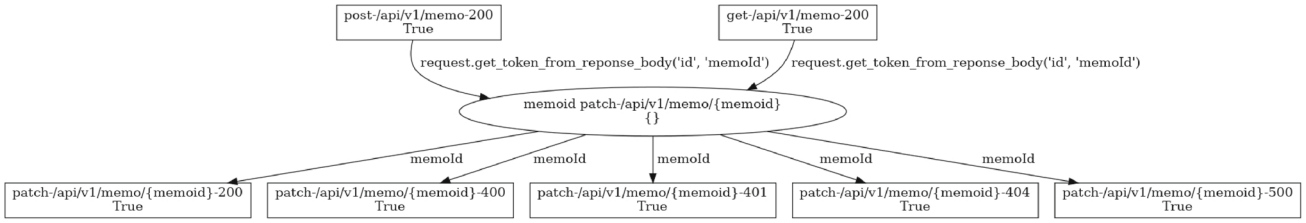
Internal validity

Since we are not examining causal relationships in the results obtained, our study is free from threats to internal validity.

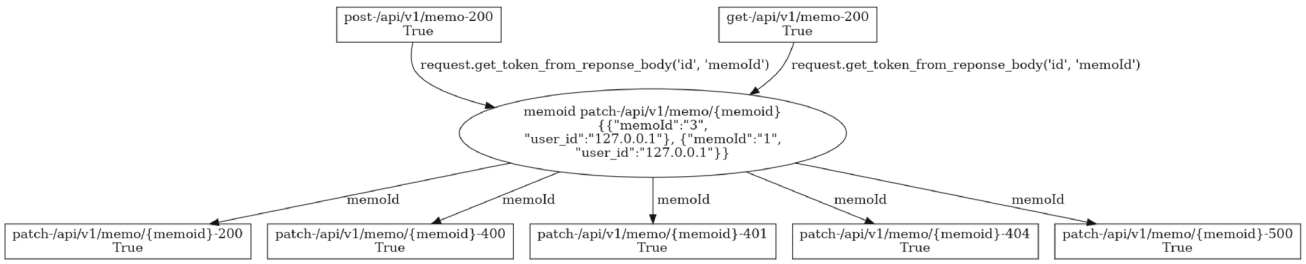
External validity

Our approach is based on Petri nets. While this approach offers clear advantages in terms of formal verification and deterministic detection of structured vulnerabilities, we recognize that it can benefit from integration with data mining and anomaly detection techniques. These methods can complement Petri nets by providing a means to detect unknown or evolving attack patterns based on statistical anomalies or

[16] Accessible in https://github.com/ailton07/memos-with-BOLA.



(a) Initial state of the CPN, result of the `Memos` OpenAPI transformation.



(b) CPN state after processing line 11.

Fig. 7 Result of running the transformation algorithms on the event log for `Memos`

behavioral deviations in the system. Future research could explore a hybrid model, where Petri nets are used for formal analysis and detection of known attacks, while machine learning-based techniques identify never-before-seen attack vectors. This would create a more comprehensive and layered approach to REST API security.

Our approach is also tied to a concrete OpenAPI specification. Therefore, our approach may need to be adapted to future OpenAPI specifications. Also, we assume that the web server logs that are necessary for the replay algorithm are centralized. This extent, however, is discouraged in network administration, where the principle of network segmentation (as a way to reduce the number of assets in a network segment, limiting lateral movements of potential attackers) is commonly recommended [12].

Similarly, `Links2CPN` currently works with the JSON file generated after parsing the web server logs. Therefore, it would be necessary to adapt our tool to analyze the log files of other web servers.

As for our user-based assessment, it is true that there is no population from which a statistically representative sample has been drawn. However, as we use it as case studies, we can conclude that the results are extensible to cases that have common characteristics and therefore the findings are relevant.

Reliability

Our tool, the running example, and the results of our user-based evaluation are publicly accessible through our GitHub repository [44]. Therefore, other researchers can perform the same study later and the results would be the same.

6 Related work

A lot of research has been done in the field of RESTful application modeling and documentation [22]. In 2015, a widely accepted standard emerged with the OpenAPI initiative effort to standardize the description of RESTful APIs. Although the OpenAPI specification can provide details about the relationship between the operations, it focuses on structural and data modeling aspects, lacking behavioral aspects [22]. We begin by reviewing the literature on modeling RESTful APIs using non-domain-specific languages. We then analyze studies that use the OpenAPI specification to detect attacks and vulnerabilities in REST APIs.

6.1 Visual modeling of REST API behavior

In this section, we review works that address the problem of using non-domain-specific languages to visually model the behavior of REST and RESTful APIs. In particular, we divide the discussion into OpenAPI specification, Petri net-based, and UML based approaches.

6.1.1 OpenAPI specification approaches

A systematization of the Insecure Direct Object Reference (IDOR) and BOLA attack techniques based on the literature review and the analysis of real cases is provided in

[5], with the purpose of proposing an approach to describe IDOR/BOLA attacks based on the properties of the OpenAPI specifications and, subsequently, develop an algorithm for detecting potential (i.e., not confirmed) IDOR/BOLA vulnerabilities. The proposed detection approach consists of providing a valid OpenAPI specification, annotating potentially vulnerable properties, and determining which attack vector techniques are applicable. If any condition of attack vector techniques is found to be met, then it is considered a potential IDOR/BOLA vulnerability that needs to be checked. Additionally, it specifies a combination of endpoints, operations, and parameters that are potentially vulnerable and can be attacked with corresponding attack vectors. After this step, an analyst must manually test and verify whether the vulnerability actually exists. This approach is applied to two experiments. The first experiment is to generate example specifications that contain at least one vulnerability for each established detection rule. The second experiment uses publicly available specifications containing potential vulnerabilities, where the exact number of potential (access control) vulnerabilities is not defined.

An extension to the OpenAPI specification, called the OAS Security Scheme, is proposed in [20]. This extension introduces new properties to function as a security control mechanism for declarative security descriptions, with the goal of providing and standardizing an authorization capability to protect resources from unauthorized access. Using this custom OpenAPI specification, a specialized authorization module can apply object-level authorization checks while the API is running and calls are made. Although this extension may encourage further studies to improve the security of the OpenAPI specification, its scalability is not fully studied.

These works consider the OpenAPI specification as a source of truth to establish a baseline of how the API should work, as our approach does. Therefore, this is a prerequisite for their correct functioning. Additionally, the documentation must be reliable, that is, correctly describe the underlying API. The approach given in [5] is complementary to ours, which can provide another vulnerability analysis as a way to detect false positives (or false negatives) from the previous approach. Although our approach is also applicable to the extension of the OpenAPI specification proposed in [20], it may require additional remodeling to capture the newly introduced security control mechanism.

6.1.2 Petri net-based approaches

In [10], the authors introduce a formal model for *process enactment* in REST systems using *Service Nets*, another class of Petri Nets. Later, in [1], this formalism is used to convert a REST description language (proposed by the authors) into a Service Net. Unfortunately, as discussed in [25], these approaches ignore internal hypermedia links, describe all tokens in XML only, and do not check the correctness of composition behavior. Also, they do not describe RESTful APIs, but REST. Unlike these approaches, our model transformation approach is based on a standard specification and is well suited for describing RESTful APIs.

In [29] and [28], the authors propose *REST Chart*, a Petri net-based XML modeling framework (model and markup language) to describe REST APIs without violating the REST constraints. Their approach is based on modeling REST APIs as a set of hypermedia representations and transitions between them, where each transition specifies the possible interactions with the resource referenced by a hyperlink in a type representation. In [25], the authors propose a formal CPN-based language for modeling and verifying RESTful service composition. They defined data types as colors, a unique definition of a resource as a service identified by a URI, which aligns with the OpenAPI specification and focuses on using CPNs properties to verify the correctness of RESTful composition behaviors. However, the authors do not discuss how their approach can be used to represent multiple users at the same time. By contrast, our model transformation approach is well suited to representing concurrent users.

6.1.3 UML-based approaches

Other approaches are based on the *Unified Modeling Language* (UML) [34], a widely adopted standard notation for modeling software systems. In [2], the authors propose an approach to describe RESTful and resource-oriented Web services using UML collaboration diagrams. Their model describes the type of resource, the relationships between them, and the control flow. However, the data flow is not considered. In [40], the authors provide a methodology for designing REST web service interfaces using a UML class diagram to represent the resource mode and a UML state machine diagram (with state invariants) to represent the behavioral model of a REST web service. Their model focuses on states that use the POST, PUT, or DELETE requests to trigger flow between states, while the GET method is used to check for state invariants. In [14], the authors proposed *WAPIML*, a software tool for converting the OpenAPI specification to an annotated UML class diagram, editing the UML model, and converting it back to the OpenAPI specification.

6.1.4 FSM-based approaches

In [49], the authors present a finite-state machine with epsilon transitions for modeling RESTful systems. This model follows some REST constraints (i.e., uniform interface, stateless client–server operation, and code-on-demand execution), but is not suitable for modeling RESTful systems.

6.2 OpenAPI in detecting attacks and vulnerabilities

To the best of our knowledge, there are still no studies that use OpenAPI as a normative model to detect attacks against REST APIs. However, some works employ OpenAPI specifications to perform security testing, thereby revealing vulnerabilities. Below, we discuss how these studies leverage OpenAPI and the challenges they face.

The work in [20] proposes an extension to the OpenAPI specification, called OAS ESS (OpenAPI specification extended security scheme). This extension aims to include declarative security controls for objects, which can then be used at runtime by custom authorization modules to enforce authorization checks and mitigate BOLA attacks. Although promising, this approach requires further testing on real-world API implementations. In addition, the implementation of the authorization module depends on the underlying technology of the API; in this work, it was implemented for the FastAPI framework. Furthermore, this approach requires developers to manually update OpenAPI specifications to include new constructs and adapt tools that interact with the specification, which could hamper its wider adoption.

In [5], the authors present an algorithm and tool that automatically analyze OpenAPI specifications and generate a list of potential vulnerabilities and attack vectors. The approach analyzes an OpenAPI specification and applies heuristics related to BOLA vulnerabilities, such as HTTP methods and parameter locations. These heuristics create a custom annotated OpenAPI specification to determine which attack vector techniques might be applied. If an attack vector condition is met, the associated endpoints, operations, and parameters are flagged as potentially vulnerable and require further validation by a human analyst. Although tested in two experimental scenarios, the algorithm generates some false positives and false negatives, indicating that further refinement is needed.

Similarly, [11] proposes the use of an annotated OpenAPI specification that encodes operation relationships and parameter generation strategies for RESTful APIs. However, unlike previous work, the focus here is on penetration testing. The annotated specification is processed by an automated testing tool called `NAUTILUS`, which detects vulnerabilities by creating API operation sequences and populating them with parameter values. Relationships between operations are automatically inferred using attribute naming conventions in request and response descriptions, along with heuristics. However, this can sometimes lead to conflicting operation dependencies. The approach was compared against four vulnerability scanners and RESTful API testing tools on six RESTful services, demonstrating its ability to detect vulnerabilities such as access bypass, directory traversal, and SQL injection. However, the OpenAPI specifications used in the tests were manually annotated. The authors suggest integrating the annotation process into the OAS creation phase, which is typically performed by developers.

Finally, an API inspection framework for API security testing, called VOAPI, is proposed in [13]. This framework was designed to identify vulnerabilities such as SSRF, unrestricted upload, path traversal, command injection, SQL injection, and XSS. The framework focuses on analyzing API specifications to identify functions associated with vulnerabilities and perform targeted security testing on those functions. For example, file upload interfaces may pose a risk for malicious uploads, while proxy interfaces may lead to SSRF vulnerabilities. In particular, the authors analyzed a dataset of API-related CVEs to establish connections between keywords, corresponding API functionalities, and vulnerability types. They then used algorithms from [4] to analyze the API specification and extract possible vulnerability types and API operations. For each API operation to be tested, an algorithm was applied to create a sequence of API requests that reached a valid state to execute the operation under test. This algorithm is similar to [11] but includes an additional heuristic, the *producer-consumer relationship*. A test case generator then focused on potential vulnerabilities and a feedback-based vulnerability checker verified whether operations were actually vulnerable. The approach was tested on seven real-world RESTful APIs and compared to five other tools designed to detect similar vulnerabilities. While the test sequence generation was compared to other automated methods, it was not evaluated in scenarios where the links in the API specification explicitly indicated valid request sequences. According to the authors, the test sequence generation achieved 62.7% coverage of the tested endpoints.

Comparison with related work. To the best of our knowledge, there are no solutions in the literature that directly target the detection of BOLA attacks in REST API systems. While several tools and methods focus on general web attack detection and automated testing, none specifically addresses the challenges posed by BOLA vulnerabilities. Our work contributes to filling this gap by offering a novel approach to detect BOLA attacks using colored Petri nets derived from OpenAPI specifications. In addition, as shown, many works focus on modeling REST systems that are also described in non-standard ways. Rather, our approach is to model the data flow of the RESTful system described through the OpenAPI specification. While some solutions aim to automate the description of data flows in APIs, none have been able to fully capture all flows, achieving at best a coverage of about 60%. Finally, unlike other works, we provide a tool to make it easy to use our approach.

7 Conclusions and future work

This paper presents a transformation from OpenAPI to Petri nets. We have developed a tool that supports this transformation, called `Links2CPN`, which is publicly available and freely accessible in our GitHub repository. Our evaluation showed that `Links2CPN` can detect broken object level authorization attacks (the first OWASP Top 10 2023 security risk in web applications) in web server logs with more than 95% accuracy.

There are still some limitations that can be addressed in future work. First, our tool assumes the existence of a well-formed OpenAPI specification with the attributes required for analysis, like other previous works [5, 20]. Our ongoing efforts aim to somewhat relax this assumption and require instead source code annotations to automatically parse and build the input model needed for the transformation. Second, logs must be centralized. This can be difficult in heavily distributed applications, but it may be feasible in practice. Similarly, in the case of multiple log files referring to the same period, a chronological sorting of the logs is required before processing.

We also intend to conduct a broader evaluation incorporating more realistic traffic distributions, including a higher proportion of legitimate requests. This will allow us to evaluate the performance of our approach in environments that more closely resemble typical application usage, complementing the results obtained from our more attack-focused experiments. In addition, we plan to extend our approach by incorporating fuzzing techniques to generate HTTP interactions, turning it into a penetration testing tool for comprehensive REST API security assessments. Finally, our tool only performs offline analysis. A big step to protect the security of RESTful web services would be to integrate our tool directly into the web server, analyzing requests in real time. We envision eBPF technology as a possible solution to do this.

Appendix A: Excerpt of OpenAPI used as a running example

Listing 5 Excerpt of OpenAPI Used as a Running Example

```
openapi: 3.0.0
servers:
  # Added by API Auto Mocking Plugin
  - description: SwaggerHub JuiceShop
    API
    url: https://virtserver.swaggerhub.
       com/ailton07/JuiceShop/1.0.0
info:
  description: JuiceShop API
     Description.
  version: "1.0.0"
  title: JuiceShop API Description
  contact:
    email: you@your-company.com
  license:
    name: Apache 2.0
    url:  http://www.apache.org/
       licenses/LICENSE-2.0.html
tags:
  - name: View Basket
    description: View another user s
       shopping basket
paths:
  /rest/user/login:
    post:
      tags: ["View Basket"]
      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:
                 getBasketById
              parameters:
                id: $response.body#/
                   authentication.bid
  /rest/basket/{bid}:
    get:
      tags: ["View Basket"]
      operationId: getBasketById
      parameters:
      - name: bid
        in: path
        required: true
        schema:
          type: string
      responses:
         200 :
          description: search results
             matching criteria
          content:
            application/json:
              schema:
                type: object
                $ref:  #/components/
                   schemas/
                   GetBasketByIdResponse

         400 :
          description: Error resposen
             status code 400
```

```
        401 :
          description: Error resposen
              status code 401
        500 :
          description: Error resposen
              status code 500
components:
  schemas:
    LoginUserRequest:
      type: object
      properties:
        email:
          type: string
        password:
          type: string
    LoginUserResponse:
      type: object
      properties:
        authentication:
          type: object
          properties:
            token:
              type: string
            bid:
              type: integer
            umail:
              type: integer
    GetBasketByIdResponse:
      type: object
      properties:
        status:
          type: string
        data:
          type: object
          properties:
            id:
              type: integer
            coupon:
              type: string
            createdAt:
              type: string
              format: date
            updatedAt:
              type: string
              format: date
            UserId:
              type: integer
              example: 96
            Products:
                type: array
                items:
                  $ref:  #/components/
                      schemas/Product
    Product:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        description:
          type: string
        price:
          type: number
          example: 1.10
        deluxePrice:
          type: number
          example: 0.99
        image:
          type: string
        createdAt:
          type: string
          format: date
        updatedAt:
          type: string
          format: date
        deletedAt:
          type: string
          format: date
        BasketItem:
          type: object
          properties:
            id:
              type: integer
            quantity:
              type: integer
            createdAt:
              type: string
              format: date
            updatedAt:
              type: string
              format: date
            BasketId:
              type: integer
            ProductId:
              type: integer
```

Acknowledgements The authors would like to express their sincere gratitude to the reviewers for their valuable comments and constructive suggestions, which have significantly contributed to improving the clarity, depth, and overall quality of this manuscript. The research of Ricardo J. Rodríguez was supported in part by the grant TED2021-131115A-I00, funded by MCIN/AEI/10.13039/501100011033, by the Recovery, Transformation and Resilience Plan funds, financed by the European Union (Next Generation), by the Spanish National Cybersecurity Institute (INCIBE) under *Proyecto Estratégico CIBERSEGURIDAD EINA UNIZAR*, and by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-23R).

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Research Data Policy and Data Availability Statements The data that support the findings of this study are openly available in a software repository of GitHub [44].

Declarations

Conflict of interest The authors declare that they have no Conflict of interest.

Ethical approval All procedures performed in studies involving human participants were in accordance with the ethical standards of the institutional research committee and with the 1964 Helsinki declaration and its later amendments or comparable ethical standards.

Informed consent Informed consent was obtained from all individual participants included in the study.

References

1. Alarcon, R., Wilde, E., Bellido, J.: Hypermedia-driven RESTful service composition. In: Maximilien, E.M., Rossi, G., Yuan, S.T., Ludwig, H., Fantinato, M. (eds.) Service-Oriented Computing, pp. 111–120. Springer, Heidelberg (2011)
2. Alowisheq, A., Millard, D.E., Tiropanis, T.: Resource oriented modelling: describing restful Web Services using collaboration diagrams. In: Proceedings of the International Conference on e-Business, IEEE, pp 1–6 (2011)
3. Anumotu, S., Jha. K., Balhara, A., Chawla, P.: Security issues and vulnerabilities in web application. In: Kumar, R., Pattnaik, P.K., Tavares, J.M. (Eds.) Next Generation of Internet of Things, Springer Nature Singapore, Singapore, pp 103–114 (2023)
4. Atlidakis, V., Godefroid, P., Polishchuk, M.: RESTler: Stateful REST API Fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 748–758. https://doi.org/10.1109/ICSE.2019.00083 (2019)
5. Barabanov, A., Dergunov, D., Makrushin, D., Teplov, A.: Automatic detection of access control vulnerabilities via API specification processing. CoRR abs/2201.10833, arXiv:2201.10833 (2022)
6. Carmona, J., van Dongen, B., Solti, A., Weidlich, M.: Conformance Checking. Springer International Publishing, Berlin (2018). https://doi.org/10.1007/978-3-319-99414-7
7. Carrasquel, J.C., Mecheraoui, K., Lomazova, I.A.: Checking Conformance Between Colored Petri Nets and Event Logs. In: van der Aalst, W.M.P., Batagelj, V., Ignatov, D.I., Khachay, M., Koltsova, O., Kutuzov, A., Kuznetsov, S.O., Lomazova, I.A., Loukachevitch, N., Napoli, A., Panchenko, A., Pardalos, P.M., Pelillo, M., Savchenko, A.V., Tutubalina, E. (eds.) Analysis of Images, pp. 435–452. Social Networks and Texts, Springer International Publishing, Cham (2021)
8. Clay, J.: Recent cyberattacks increasingly target open-source Web Servers. https://www.trendmicro.com/en_ae/research/22/b/recent-cyberattacks-open-source-web-servers.html. Accessed on February 23, 2023 (2022)
9. Collado, E.S., Castillo, P.A., Merelo Guervós, J.J.: Using evolutionary algorithms for server hardening via the moving target defense technique. In: Castillo, P.A., Jiménez Laredo, J.L., Fernández de Vega, F. (eds.) Applications of Evolutionary Computation. Springer International Publishing, Cham, pp 670–685 (2020)
10. Decker, G., Lüders, A., Overdick, H., Schlichting, K., Weske, M.: RESTful Petri Net Execution. In: Bruni, R., Wolf, K. (eds.) Web Services and Formal Methods, pp. 73–87. Springer, Heidelberg (2009)
11. Deng, G., Zhang, Z., Li, Y., Liu, Y., Zhang, T., Liu, Y., Yu, G., Wang, D.: NAUTILUS: Automated RESTful API vulnerability detection. In: 32nd USENIX security symposium (USENIX Security 23), USENIX Association, Anaheim, CA, pp. 5593–5609. https://www.usenix.org/conference/usenixsecurity23/presentation/deng-gelei (2023)
12. Diogenes, Y., Ozkaya, E.: Cybersecurity – Attack and Defense Strategies, 2nd edn. Packt Publishing, Birmingham (2019)
13. Du, W., Li, J., Wang, Y., Chen, L., Zhao, R., Zhu, J., Han, Z., Wang, Y., Xue, Z.: Vulnerability-oriented testing for RESTful APIs. In: 33rd USENIX Security Symposium (USENIX Security 24), USENIX Association, Philadelphia, PA, pp. 739–755, https://www.usenix.org/conference/usenixsecurity24/presentation/du (2024)
14. Ed-douibi, H., Cánovas Izquierdo, J.L., Bordeleau, F., Cabot, J.: WAPIml: towards a modeling infrastructure for Web APIs. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 748–752. https://doi.org/10.1109/MODELS-C.2019.00116 (2019)
15. Emmons, T., McReynolds, S., Lauro, T., Kimhy, E.: Akamai web application and API threat report. https://www.akamai.com/resources/research-paper/akamai-web-application-and-api-threat-report. Accessed on 23 Feb 2023 (2022)
16. Fielding, R., Reschke, J.: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. [Online; https://www.rfc-editor.org/rfc/rfc7231]. Accessed 23 Feb 2023 (2014)
17. Fielding, R.T.: Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine (2000)
18. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Boston (1994)
19. Gómez, A., Rodríguez, R.J., Cambronero, M.E., Valero, V.: Profiling the publish/subscribe paradigm for automated analysis using colored petri nets. Softw. Syst. Model. **18**(5), 2973–3003 (2019). https://doi.org/10.1007/s10270-019-00716-1
20. Haddad, R., Malki, R.E.: OpenAPI specification extended security scheme: A method to reduce the prevalence of Broken Object Level Authorization. https://arxiv.org/abs/2212.06606. Accessed on Oct 19 2023 (2022). arXiv:2212.06606
21. huntr: IDOR to archive victims memo vulnerability found in memos. https://huntr.dev/bounties/e65b3458-c2e2-4c0b-9029-e3c9ee015ae4/, Accessed on 19 Oct 2023 (2022)
22. Ivanchikj, A.: RESTalk: a visual and textual DSL for modelling RESTful conversations. phdthesis, Università della Svizzera italiana (2021)
23. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems, 1st edn. Springer, Berlin (2009)
24. Jin, B., Sahni, S., Shevat, A.: Designing Web APIs: Building APIs That Developers Love, 1st edn. O'Reilly Media, Inc (2018)
25. Kallab, L., Mrissa, M., Chbeir, R., Bourreau, P.: Using colored petri nets for verifying restful service composition. In: Panetto, H., Debruyne, C., Gaaloul, W., Papazoglou, M., Paschke, A., Ardagna, C.A., Meersman, R. (eds.) On the Move to Meaningful Internet Systems. OTM 2017 Conferences, Springer International Publishing, Cham, pp. 505–523 (2017)
26. Kim, M., Stennett, T., Shah, D., Sinha, S, Orso, A.: Leveraging large language models to improve REST API testing. In: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, Association for Computing Machinery, New York, NY, USA, ICSE-NIER'24, p 37-41, https://doi.org/10.1145/3639476.3639769 (2024)
27. Kus, D.A., Koren, I., Klamma, R.: A link generator for increasing the utility of openapi-to-graphql translations. CoRR abs/2005.08708. arXiv:2005.08708 (2020)
28. Li, L., Chou, W.: Designing large scale REST APIs based on REST chart. In: 2015 IEEE International Conference on Web Services, IEEE, pp. 631–638 (2015)

29. Li, L., Chou, W. Design and describe REST API without violating REST: A Petri net based approach. In: 2011 IEEE International Conference on Web Services, IEEE, pp. 508–515 (2011)
30. Madden, N. (2020) API security in action. Manning Publications
31. Marashdeh, Z., Suwais, K., Alia, M.: A Survey on SQL Injection attack: detection and challenges. In: 2021 International Conference on Information Technology (ICIT), pp 957–962, https://doi.org/10.1109/ICIT52682.2021.9491117 (2021)
32. Miller, D., Harmon, J., Whitlock, J., Hahn, K., Gardiner, M., Ralphso, M., Dolin, R., Ratovsky, R., Tam, T.: OpenAPI Specification v3.1.0. https://spec.openapis.org/oas/v3.1.0. Accessed on 09 Oct 2024 (2021)
33. Murata, T.: Petri Nets: properties, analysis and applications. Proc. IEEE **77**, 541–580 (1989)
34. OMG: Unified Modelling Language: Superstructure. Object Management Group, version 2.4, formal/11-08-05 (2011)
35. OpenAPI Initiative: Best Practices. https://learn.openapis.org/best-practices.html. Accessed 01 Oct 2024 (2023)
36. OpenAPI Initiative: Code Generators. https://tools.openapis.org/categories/code-generators.html. Accessed on 01 Oct 2024 (2024)
37. Pommereau, F.: SNAKES: A flexible high-level petri nets library (tool paper). In: Application and Theory of Petri Nets and Concurrency: 36th International Conference, PETRI NETS 2015, Brussels, Belgium, June 21–26, 2015, Proceedings 36, Springer, pp. 254–265 (2015)
38. Postman (2024) 2023 State of the API report. https://www.postman.com/state-of-api/api-global-growth, Accessed 01 Oct 2024
39. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: CPN tools for editing, simulating, and analysing coloured Petri nets. In: Proceedings of the 24th International Conference on Applications and Theory of Petri Nets, Springer, pp. 450–462 (2003)
40. Rauf, I.: Design and validation of stateful composite RESTful web services. PhD thesis, Turku Centre for Computer Science (2014)
41. Richardson, L., Ruby, S.: RESTful Web Services, 1st edn. O'Reilly Media Inc, Sebastopol (2007)
42. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**(2), 131–164 (2009). https://doi.org/10.1007/s10664-008-9102-8
43. Salt Security (2022) State of API Security Report Q3 2022. https://content.salt.security/state-api-report.html. Accessed on 23 Feb 2023
44. Santos Filho A (2023) Links2CPN - OpenAPI Links to CPNs. https://github.com/ailton07/openapi-links-to-CPNs. Accessed on 27 Feb 2023
45. Schoenborn, J.M., Althoff, K.D. (2021) Detecting SQL-injection and cross-site scripting attacks using case-based reasoning and SEASALT. In: LWDA, pp. 66–77
46. Swagger-PHP (2024) Link. https://zircote.github.io/swagger-php/reference/attributes.html#link. Accessed on 01 Oct 2024
47. van der Aalst, W.: Process Mining: Data Science in Action. Springer, Berlin (2016). https://doi.org/10.1007/978-3-662-49851-4_1
48. White, J., Hays, S., Fu, Q., Spencer-Smith, J., Schmidt, D.C.: ChatGPT Prompt patterns for improving code quality. In: Refactoring, Requirements Elicitation, and Software Design, pp. 1–108. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-55642-5_4
49. Zuzak, I., Budiselic, I., Delac, G.: Formal modeling of RESTful systems using finite-state machines. In: Auer, S., Díaz, O., Papadopoulos, G.A. (eds.) Web Engineering, pp. 346–360. Springer, Heidelberg (2011)