International Journal of Information Security manuscript No.
(will be inserted by the editor)

Automated Attack Detection in REST APIs through OpenAPI to Colored Petri Nets Transformation

Ailton Santos Filho[a,1], **Ricardo J. Rodríguez**[b,2], **Eduardo L. Feitosa**[c,1]

[1]Institute of Computing, Federal University of Amazonas (UFAM), Av. General Rodrigo Octavio Jordão Ramos, 1200, Coroado I, Manaus - AM, Brazil, 69067-005
[2]Instituto de Investigación en Ingeniería de Aragón (I3A),Universidad de Zaragoza, Spain, Edificio I+D+i, Calle Mariano Esquillor s/n, Zaragoza, Spain, 50018

Received: date / Accepted: date

Abstract The Representational State Transfer (REST) architectural style 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 successful BOLA attacks with an accuracy greater than 95% in the proposed scenarios.

[a]e-mail: `assf@icomp.ufam.edu.br`
[b]e-mail: `rjrodriguez@unizar.es`
[c]e-mail: `efeitosa@icomp.ufam.edu.br`

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 [15]. 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* [35].

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 [27]. Many popular web services, such as Twitter, 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 [29], which 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) [28] and *cross-site scripting* (XSS) [13], although other attacks, such as *denial of service* and authentication or session management, are also feasible [37]. 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 [29] to Petri nets [30], 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 3.0.3) and model its parts using Colored Petri nets (CPNs) [22], an extension of the classical Petri net formalism with data, time, and hierarchy. These models are built into a single CPN that is suitable for analysis with specific tools such as CPN Tools [33]. 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 2019 security risk, related to broken access control [4]. Using a JSON event log (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 as follows. Section 2 gives some background on 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 Section 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.

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

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

2 Background

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

2.1 OpenAPI Specification

OpenAPI Specification (OAS) [29] 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 JSON object, which can be represented in JSON or YAML format.

The extended Backus-Naus form [20] of an OAS-compliant OpenAPI document is (partially) shown in Listing 1. It is made up of a set of fields that describe a REST API. There are two types of fields: *fixed fields*, which have a declared name; and *patterned* fields, which declare a regex pattern for the field name. A patterned field must have a unique name within the containing object. These fixed fields are `openapi`, `info`, `servers`, `paths`, `components`, `security`, `tags`, and `externalDocs`. The patterned field we are most interested in is `paths`, which is responsible for listing the available paths and operations for the API.

Each `Path` object contains a patterned field that is of type `Path Item`. This object is responsible for describing the HTTP operations available on that particular path. The operations are named as the HTTP method (in plain text, as defined by RFC7231 [14]) and is of type `Operation`. It describes a single API operation on a path and provides a summary, description, and unique identifier, further describing the operation parameters, payloads, and possible server responses. In particular, the list of parameters applicable for all the operations described in this path are in the `parameters` fixed field, which is an array containing `Parameter` objects. This array can also contain `Reference` objects, which reference other components in the specification, internally and externally. These parameters can be later redefined at the operation level (via the `Operation` object).

Each `Parameter` object describes a single operation parameter as a combination of at least one name and one location (fixed fields `name` and `in`, respectively). The fixed field `schema`, of type `Schema`, defines the structure and the type of the parameter (it can be a number, a string, a boolean value, an array, or an object).Another fixed field of the `Operation` object is `requestBody`. This object is responsible for describing the body of the HTTP request and the formats for that particular `Operation` object through its `content` fixed field, of type `Media Type`. This object provides the schema that defines the content of the request and examples for the media type identified by its key, which is a standard RFC6838 media type value [16].

Listing 1: An excerpt from the Backus-Naus form of the OpenAPI specification (in YAML).

```
oas-document ::= openapi info paths { optional
    -fields }
optional-fields ::= servers | components |
    security |
                        tags | externalDocs
paths ::= "paths:" path+
path ::=  path-item [ summary]  [ description]
     [ servers ] [ $ref ] { parameters }
path-item ::= ["get:" operation ] ["post:"
    operation ] ["delete:" operation ] ["
    options:" operation ] ["head:" operation ]
     ["patch:" operation ] ["trace:"
    operation ]
operation ::= operationId responses [ request-
    body ] { parameters }
parameters ::= parameter | reference
parameter ::= name in required [ schema ] ...
request-body ::= "requestBody:" content+ [
    description ]  [ required ]
content ::=  media-type  [ schema ] [ example
    ] [ examples ] [ encoding ]
media-type ::= "application/json:" | "
    application/html:" ...
responses :: =  [default] HTTP-status+
HTTP-status ::= HTTP-status-code { response |
    reference }
response ::= description headers* { content }
    { links }
HTTP-status-code ::= "200:" | "400:" ...
links ::= link | reference
link ::= operation-ref operation-id [ request-
    body ] [ description ] [ server ] {
    parameters }
```

The `Operation` object has a single mandatory fixed field, `responses`, which defines the list of responses expected from an operation. In particular, it maps a HTTP response code (as defined by RFC7231 [14]) to the expected response. The response can be a `Response` or a `Reference` object. A `Response` object describes a single operation response, containing a required field called `description` to textually detail the meaning of the response in the context of this operation, as a way to help developers understand better how to react to this response. It can also contain a *content* field, which is a map containing descriptions of possible response payloads (similar to the same field in requests).

The `Response` object can include a fixed field called `links`, which maps the operation links that can be followed from this particular response. Map elements can be a `Link` or `Reference` object. A `Link` object represents a possible design-time link for a response. The presence of a link provides a known relationship and traversal mechanism between responses and other operations in the web service. Note that its presence does not guarantee the caller's ability to invoke it successfully (for instance, authentication or authorization restrictions). A linked operation can be identified using the *operationRef* field (a relative or absolute URI reference to an operation) or the *operationId* field (the name of an existing resolvable operation). This object can also describe how the return values of one operation can be used as input parameters and request body for other operations via the *parameters* and *requestBody* fields, respectively.

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[3], as in [24], and detected only 9 using `links` (listed in [38]). 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. Therefore, similar to [5, 18], we assume that development teams can update their OpenAPI specifications if they want to detect BOLA vulnerabilities with our approach.

2.2 Colored Petri Nets (CPNs)

Colored Petri nets [22] are a well-known formalism for the design and analysis of concurrent systems. CPNs are supported by *CPN Tools* [33], 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 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 [22].

Petri nets [30] 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.

[3]Accessible in `https://apis.guru/`.

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 [22] 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 = INTx-INT`, 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 [17]. According to [1], CPNs are the most widely used Petri net-based formalism that can deal with issues related to data and time.

More formally:

Definition 1 [22] A Colored Petri Net (CPN) is a tuple $\langle P,T,A,V,G,E,\pi\rangle$, where[4]:

- 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*[5], which assigns a Boolean expression to each transition, i.e., $Type(G(t)) = Bool$.
- $E : A \longrightarrow EXPR_V$ 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 $M : P \longrightarrow \mathscr{B}(\Sigma)$, such that $\forall p \in P,\ M(p) \in \mathscr{B}(\Sigma_p)$, 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 [22], 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.
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

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

[5]$EXPR_V$ denotes expressions built using the variables in V, with the same syntax supported by *CPN Tools*.

and checks their shopping cart. This example is taken from the OWASP Juice Shop project[6], 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 Figure 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[7]. 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 Figure 1, the server will respond with the another user's shopping basket information.

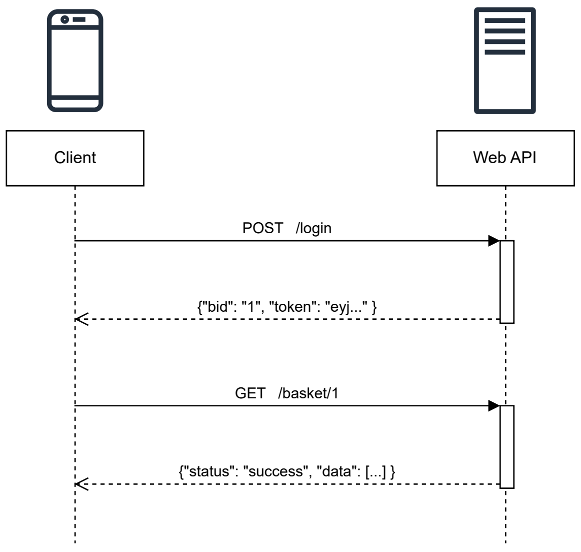

Client
Web API
POST /login
{"bid": "1", "token": "eyj..." }
GET /basket/1
{"status": "success", "data": [...] }

Fig. 1: A simple WebAPI interaction.

The excerpt of OpenAPI specification that we will use as running example is shown in Listing 6 (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.

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

4 Methodology and the Tool `Links2CPN`

In this section, we first briefly introduce the methodology we follow to detect BOLA attacks in OpenAPI specifications. 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 Section 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 [1], depending on the input data and the questions that need to be addressed, an appropriate model and abstraction level must be chosen. In this paper, we consider that CPN provides a suitable abstraction and model to represent REST APIs because: (i) control flow and data flow need to be modeled together; (ii) the literature shows that REST APIs can be modeled as CPNs [10, 25, 26]; and (iii) there are conformance checking algorithms for CPN [7]. Below, we present a model transformation from a valid OpenAPI document to a CPN and illustrate it with 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 $\mathscr{C} = \langle P, T, A, V, G, E, \pi \rangle$ that represents `doc`.

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

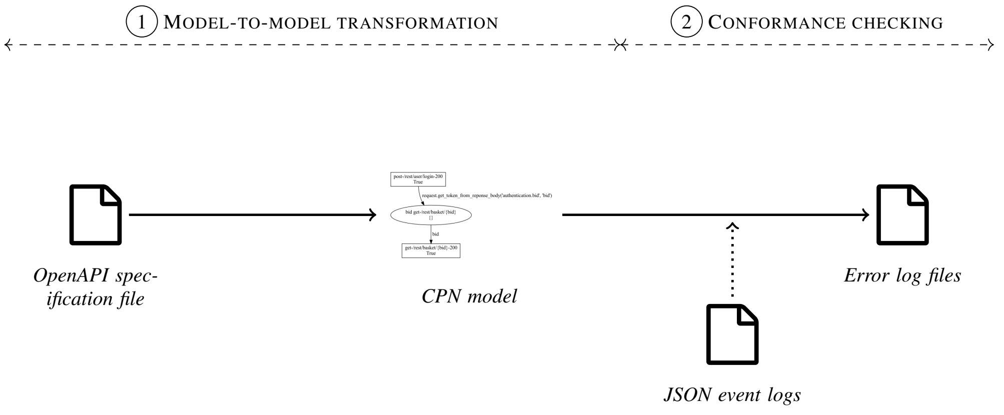


Fig. 2: Methodology to detect BOLA attacks in OpenAPI specifications.

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

```
Input: An OpenAPI specification doc.
Output: A Colored Petri net $\mathscr{C} = \langle P, T, A, V, G, E, \pi \rangle$.
1  /* Create all the transitions related to
      paths                                    */
2  $\mathscr{C}$ = createTransitionsForPaths(doc) (see Algorithm 2);
3  /* Connect transitions related to paths
      */
4  $\mathscr{C}$ = connectTransitionsForPaths(doc, $\mathscr{C}$) (see Algorithm 3);
5  Remove disconnected transitions and places in $\mathscr{C}$;
6  return $\mathscr{C}$;
```

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

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

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

```
Input: An OpenAPI specification doc and a Colored Petri
       net $\mathscr{C} = \langle P, T, A, V, G, E, \pi \rangle$.
Output: An updated Colored Petri net
        $\mathscr{C} = \langle P, T, A, V, G, E, \pi \rangle$.
1  foreach path $\mathscr{P} \in$ doc do
2      foreach operation $\mathscr{O} \in \mathscr{P}$ do
3          operationId ← Get operationId from $\mathscr{O}$;
4          HTTPMethod ← Get HTTPmethod from $\mathscr{O}$;
5          foreach response $\mathscr{R} \in \mathscr{O}$ do
6              Let $t \in \mathscr{T}$ be the transition with id
                 operationId;
7              if $\exists$ links $\mathscr{L} \in \mathscr{R}$ then
8                  operationId' ← Get operationId
                   from $\mathscr{L}$;
9                  parameter ← Get parameters from
                   $\mathscr{L}$;
10                 Let $t' \in \mathscr{T}$ 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 $\mathscr{C}$
```

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. 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 de-

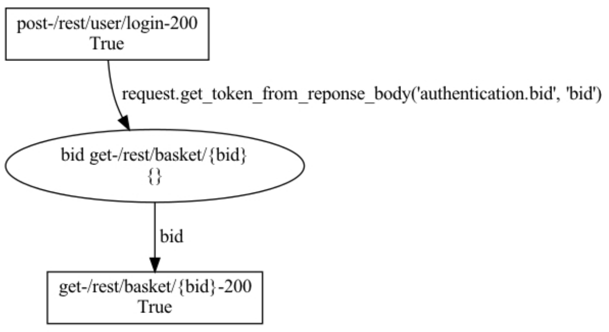

post-/rest/user/login-200
True
request.get_token_from_reponse_body('authentication.bid', 'bid')
bid get-/rest/basket/{bid}
{}
bid
get-/rest/basket/{bid}-200
True

Fig. 3: Transformation from OpenAPI specification to CPN.

picted in Figure 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`[8]. The API of `Memos` had a BOLA vulnerability related to note archiving operation that was found and fixed in 2022 [19]. 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[9].

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.

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

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

4.2 Conformance Checking: Detecting Broken Object Level Authorization Attacks

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 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, 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 2 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[10].

Listing 2: 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..."
}
```

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

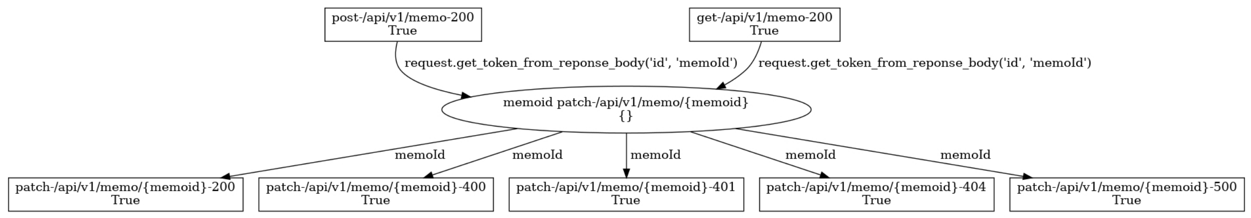

post-/api/v1/memo-200
True
get-/api/v1/memo-200
True
request.get_token_from_reponse_body('id', 'memoId')
request.get_token_from_reponse_body('id', 'memoId')
memoid patch-/api/v1/memo/{memoid}
{}
memoId
memoId
memoId
memoId
memoId
patch-/api/v1/memo/{memoid}-200
True
patch-/api/v1/memo/{memoid}-400
True
patch-/api/v1/memo/{memoid}-401
True
patch-/api/v1/memo/{memoid}-404
True
patch-/api/v1/memo/{memoid}-500
True

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

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.

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 [4]. `Links2CPN` is freely accessible in our GitHub [38] and is developed in Python 3 on top of the *Snakes* [32] 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* [11] 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 Section 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. Finally, we describe the threats to validity of our approach.

5.1 Running Example Evaluation

Similar to other works in the literature [9, 39], 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[12]. 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[13]. 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

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

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

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

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 [38]. This 30-line log file is processed and transformed using the algorithms described in Section 4.1. The obtained CPN is then verified with the replay algorithm as explained in Section 4.2.

As a result, we got the nets shown in Figure 5 and the error message shown in Listing 3. Figure 5a shows the initial state of the CPN, being the immediate result of the transformation from OpenAPI to CPN. Likewise, Figure 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, Figure 5c shows the CPN status after processing the line that represents the *GET /rest/basket/6* request. Listing 3 ilustrate 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 3: 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 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 [38]. 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 Section 4.2.

Listing 4: Error message received when executing challenge 2

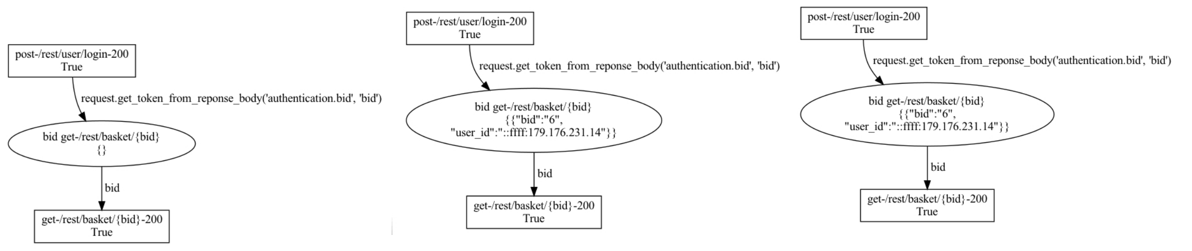
```
Fire error, Line 21: POST /api/BasketItems/
    200 11ms
transition not enabled for
{
   "BasketId␣->"{
      "BasketId":"5",
      "user_id":"::ffff:35.199.121.33"
   },
   "request␣->"{
      "uri":"/api/BasketItems/",
      "method":"POST",
      "user_id":"::ff...
   }
}
```

As a result, we got the nets shown in Figure 6 and the error message shown in Listing 4. Figure 6a shows the initial state of the CPN, being the immediate result of the transformation from OpenAPI to CPN. In addition, Figure 6b shows the CPN status after processing the log file line that represents the *POST /rest/user/login* request with response $bid = 6$. Listing 4 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 try to exploit VULNERABILITY 1 and VULNERABILITY 2 described in Section 5.1. There were 20 students enrolled in the discipline and 15 attending to the end, who were the ones invited for this evaluation.

As a test environment, we set up an instance of the Juice Shop application (see Section 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

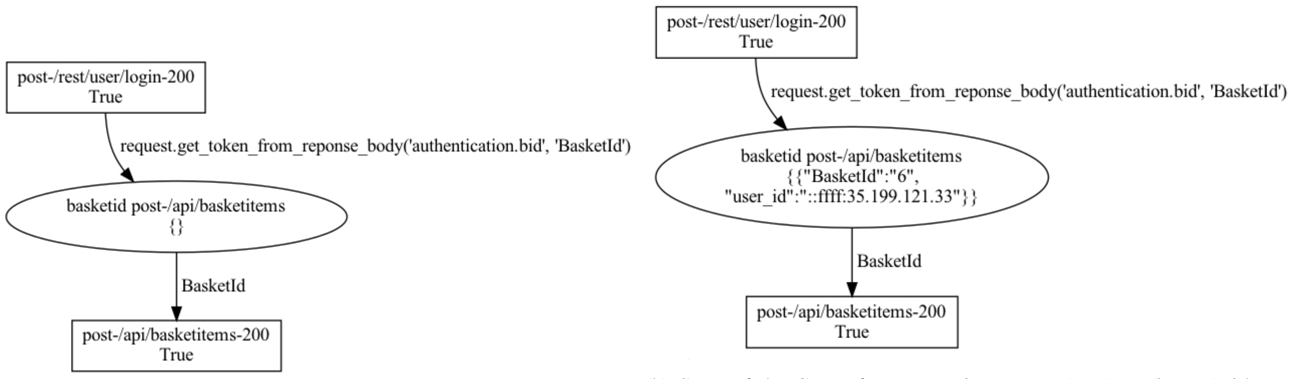

post-/rest/user/login-200
True
request.get_token_from_reponse_body('authentication.bid', 'bid')
bid get-/rest/basket/{bid}
{}
bid
get-/rest/basket/{bid}-200
True
post-/rest/user/login-200
True
request.get_token_from_reponse_body('authentication.bid', 'bid')
bid get-/rest/basket/{bid}
{{"bid":"6",
"user_id":"::ffff:179.176.231.14"}}
bid
get-/rest/basket/{bid}-200
True
post-/rest/user/login-200
True
request.get_token_from_reponse_body('authentication.bid', 'bid')
bid get-/rest/basket/{bid}
{{"bid":"6",
"user_id":"::ffff:179.176.231.14"}}
bid
get-/rest/basket/{bid}-200
True

(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.


post-/rest/user/login-200
True
request.get_token_from_reponse_body('authentication.bid', 'BasketId')
basketid post-/api/basketitems
{}
BasketId
post-/api/basketitems-200
True
post-/rest/user/login-200
True
request.get_token_from_reponse_body('authentication.bid', 'BasketId')
basketid post-/api/basketitems
{{"BasketId":"6",
"user_id":"::ffff:35.199.121.33"}}
BasketId
post-/api/basketitems-200
True

(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.

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 Section 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 Section 5.1. *Attack Attempts* refers to the number of requests that we manually classified as attacks (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) re-

Table 1: Summary of data obtained from the attacks.

	V1	V2
Total requests	517	192
Attack attempts	103	100
Successful attacks	98	20
Requests classified as attacks	101	25

fer 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, we obtain $Accuracy = Precision = Recall = F1\text{-}score = 1$, while for V2, we obtain $Accuracy = 0.609375$; $Precision = 1$; $Recall$ = 0.25; $F1\text{-}score = 0.4$. In both cases, *Precision* remains constant, indicating consistency in classifying correctly positive cases. On the contrary, *Accuracy*, *Recall*, and *F1-score* vary considerably.

To investigate the divergence in 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 $Accuracy = 0.9942$; $Precision = 0.97$; $Recall = 1$; $F1\text{-}score = 0.985$ for VULNERABILITY 1, while for VULNERABILITY 2 we obtain $Accuracy = 0.974$; $Precision = 0.8$; $Recall = 1$; $F1\text{-}score = 0.888$. These results indicate that our approach works better in detecting successfully executed attacks, rather than detecting attempted (failed and successful) attacks.

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 103 attack attempts (i.e., 95.14% success rate), while for VULNERABILITY 2, there were 20 successful attacks out of 100 attack attempts (i.e., 20% 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[14], VULNERABILITY 1 has 2 stars of difficulty, while VULNERABILITY 2 has 3 stars. This may be why 3 out of 12 students who submitted reports were unable to successfully exploit VULNERABILITY 2.

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

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 Section 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[15]. 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 [19]. Following these steps,twhen 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 [38]. This 11-line log file is processed and transformed using the algorithms described in Section 4.1 and the obtained CPN is verified with the replay algorithm as explained in Section 4.2.

Figures 7a and 7b shows the CPN obtained after the replay algorithm, while Listing 5 shows the error message obtained. In particular, Figure 7a shows the initial state of the CPN, being the immediate result of the transformation from OpenAPI to CPN, while Figure 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 5 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 5: 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
{
```

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

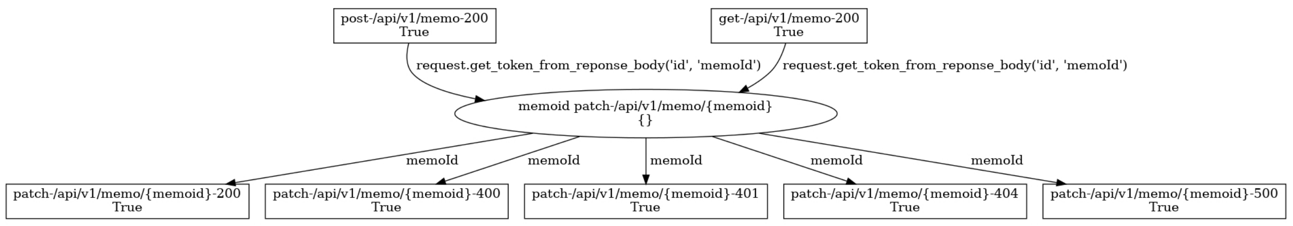
Table 2: Confusion matrices.

			Predicted	
			Positive	Negative
Actual	V1	Positive	101	0
		Negative	0	416
	V2	Positive	25	75
		Negative	0	92

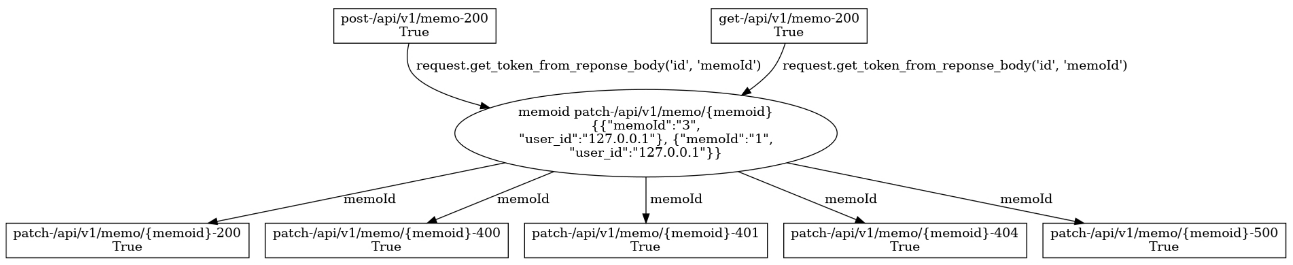
(a) Attack attempts versus detections

			Predicted	
			Positive	Negative
Actual	V1	Positive	98	0
		Negative	3	416
	V2	Positive	20	0
		Negative	5	167

(b) Successful attacks versus detections


post-/api/v1/memo-200
True
get-/api/v1/memo-200
True
request.get_token_from_reponse_body('id', 'memoId')
request.get_token_from_reponse_body('id', 'memoId')
memoid patch-/api/v1/memo/{memoid}
{}
memoId
memoId
memoId
memoId
memoId
patch-/api/v1/memo/{memoid}-200
True
patch-/api/v1/memo/{memoid}-400
True
patch-/api/v1/memo/{memoid}-401
True
patch-/api/v1/memo/{memoid}-404
True
patch-/api/v1/memo/{memoid}-500
True

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


post-/api/v1/memo-200
True
get-/api/v1/memo-200
True
request.get_token_from_reponse_body('id', 'memoId')
request.get_token_from_reponse_body('id', 'memoId')
memoid patch-/api/v1/memo/{memoid}
{{"memoId":"3", "user_id":"127.0.0.1"}, {"memoId":"1", "user_id":"127.0.0.1"}}
memoId
memoId
memoId
memoId
memoId
patch-/api/v1/memo/{memoid}-200
True
patch-/api/v1/memo/{memoid}-400
True
patch-/api/v1/memo/{memoid}-401
True
patch-/api/v1/memo/{memoid}-404
True
patch-/api/v1/memo/{memoid}-500
True

(b) CPN state after processing line 11

```
    "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 [36] 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 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 [11].

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 [38]. 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 [21]. 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 [21]. 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 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 [18]. 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 [18], it may require additional remodeling to capture the newly introduced security control mechanism.

6.2 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 [2], this formalism is used to convert a REST description language (proposed by the authors) into a Service Net. Unfortunately, as discussed in [23], 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 [25] and [26], 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 [23], 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.

6.3 UML-based Approaches

Other approaches are based on the *Unified Modeling Language* (UML) [31], a widely adopted standard notation for modeling software systems. In [3], 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 [34], 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 [12], 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.

6.4 FSM-based Approaches

In [40], 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.

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. In addition, 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 successful Broken Object Level Authorization attacks (the first OWASP Top 10 2019 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, 18]. Our ongoing efforts aim to somewhat relax this assumption and instead require instead source code annotations to 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. Lastly, 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.

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 [38].

Compliance with ethical standards

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.

Acknowledgements The research of Ricardo J. Rodríguez was supported by the grant TED2021-131115A-I00 funded by MCIN/AEI/10.13039/501100011033 and by European Union NextGenerationEU/PRTR, 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).

References

1. van der Aalst W (2016) Process Mining: Data Science in Action. Springer Berlin Heidelberg, Berlin, Heidelberg,
2. Alarcon R, Wilde E, Bellido J (2011) Hypermedia-Driven RESTful Service Composition. In: Maximilien EM, Rossi G, Yuan ST, Ludwig H, Fantinato M (eds) Service-Oriented Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 111–120

3. Alowisheq A, Millard DE, Tiropanis T (2011) Resource oriented modelling: Describing restful Web Services using collaboration diagrams. In: Proceedings of the International Conference on e-Business, IEEE, pp 1–6
4. Anumotu S, Jha K, Balhara A, Chawla P (2023) Security Issues and Vulnerabilities in Web Application. In: Kumar R, Pattnaik PK, R S Tavares JM (eds) Next Generation of Internet of Things, Springer Nature Singapore, Singapore, pp 103–114
5. Barabanov A, Dergunov D, Makrushin D, Teplov A (2022) Automatic detection of access control vulnerabilities via API specification processing. CoRR abs/2201.10833, `2201.10833`
6. Carmona J, van Dongen B, Solti A, Weidlich M (2018) Conformance Checking. Springer International Publishing,
7. Carrasquel JC, Mecheraoui K, Lomazova IA (2021) Checking Conformance Between Colored Petri Nets and Event Logs. In: van der Aalst WMP, Batagelj V, Ignatov DI, Khachay M, Koltsova O, Kutuzov A, Kuznetsov SO, Lomazova IA, Loukachevitch N, Napoli A, Panchenko A, Pardalos PM, Pelillo M, Savchenko AV, Tutubalina E (eds) Analysis of Images, Social Networks and Texts, Springer International Publishing, Cham, pp 435–452
8. Clay J (2022) Recent Cyberattacks Increasingly Target Open-source Web Servers. [Online; `https://www.trendmicro.com/en_ae/research/22/b/recent-cyberattacks-open-source-web-servers.html`], accessed on February 23, 2023.
9. Collado ES, Castillo PA, Merelo Guervós JJ (2020) Using Evolutionary Algorithms for Server Hardening via the Moving Target Defense Technique. In: Castillo PA, Jiménez Laredo JL, Fernández de Vega F (eds) Applications of Evolutionary Computation, Springer International Publishing, Cham, pp 670–685
10. Decker G, Lüders A, Overdick H, Schlichting K, Weske M (2009) RESTful Petri Net Execution. In: Bruni R, Wolf K (eds) Web Services and Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 73–87
11. Diogenes Y, Ozkaya E (2019) Cybersecurity – Attack and Defense Strategies, 2nd edn. Packt Publishing
12. Ed-douibi H, Cánovas Izquierdo JL, Bordeleau F, Cabot J (2019) 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,
13. Emmons T, McReynolds S, Lauro T, Kimhy E (2022) Akamai Web Application and API Threat Report. [Online; `https://www.akamai.com/resources/research-paper/akamai-web-application-and-api-threat-report`], accessed on February 23, 2023.
14. Fielding R, Reschke J (2014) Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. [Online; `https://www.rfc-editor.org/rfc/rfc7231`], accessed on February 23, 2023.
15. Fielding RT (2000) Architectural styles and the design of network-based software architectures. phdthesis, University of California, Irvine
16. Freed N, Klensin J, Hansen T (2013) Media Type Specifications and Registration Procedures. [Online; `https://www.rfc-editor.org/rfc/rfc6838`], accessed on February 23, 2023.
17. Gómez A, Rodríguez RJ, Cambronero ME, Valero V (2019) Profiling the Publish/Subscribe Paradigm for Automated Analysis Using Colored Petri Nets. Software and Systems Modeling 18(5):2973–3003,
18. Haddad R, Malki RE (2022) OpenAPI Specification Extended Security Scheme: A method to reduce the prevalence of Broken Object Level Authorization. [Online; `https://arxiv.org/abs/2212.06606`], accessed on October 19, 2023., `2212.06606`
19. huntr (2022) IDOR to archive victims memo vulnerability found in memos. [Online; `https://huntr.dev/bounties/e65b3458-c2e2-4c0b-9029-e3c9ee015ae4/`], accessed on October 19, 2023.
20. International Organization for Standardization (1996) ISO/IEC 14977:1996 Information technology – Syntactic metalanguage – Extended BNF. [Online; `https://www.iso.org/standard/26153.html`], accessed on February 23, 2023.
21. Ivanchikj A (2021) RESTalk: a visual and textual DSL for modelling RESTful conversations. phdthesis, Università della Svizzera italiana
22. Jensen K, Kristensen LM (2009) Coloured Petri Nets: Modelling and Validation of Concurrent Systems, 1st edn. Springer Berlin, Heidelberg
23. Kallab L, Mrissa M, Chbeir R, Bourreau P (2017) Using Colored Petri Nets for Verifying RESTful Service Composition. In: Panetto H, Debruyne C, Gaaloul W, Papazoglou M, Paschke A, Ardagna CA, Meersman R (eds) On the Move to Meaningful Internet Systems. OTM 2017 Conferences, Springer International Publishing, Cham, pp 505–523
24. Kus DA, Koren I, Klamma R (2020) A link generator for increasing the utility of openapi-to-graphql translations. CoRR abs/2005.08708, URL `https://arxiv.org/abs/2005.08708`, `2005.08708`
25. Li L, Chou W (2011) 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

26. Li L, Chou W (2015) Designing large scale REST APIs based on REST chart. In: 2015 IEEE International Conference on Web Services, IEEE, pp 631–638
27. Madden N (2020) API security in action. Manning Publications
28. Marashdeh Z, Suwais K, Alia M (2021) A Survey on SQL Injection Attack: Detection and Challenges. In: 2021 International Conference on Information Technology (ICIT), pp 957–962,
29. Miller D, Harmon J, Whitlock J, Hahn K, Gardiner M, Ralphso M, Dolin R, Ratovsky R, Tam T (2020) OpenAPI Specification v3.0.3. [Online; `https://spec.openapis.org/oas/v3.0.3`], accessed on Feburary 23, 2023.
30. Murata T (1989) Petri Nets: Properties, Analysis and Applications. In: Proceedings of the IEEE, vol 77, pp 541–580
31. OMG (2011) Unified Modelling Language: Superstructure. Object Management Group, version 2.4, formal/11-08-05
32. Pommereau F (2015) 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
33. Ratzer AV, Wells L, Lassen HM, Laursen M, Qvortrup JF, Stissing MS, Westergaard M, Christensen S, Jensen K (2003) 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
34. Rauf I (2014) Design and validation of stateful composite RESTful web services. phdthesis, Turku Centre for Computer Science
35. Richardson L, Ruby S (2007) RESTful Web Services, 1st edn. O'Reilly Media, Inc
36. Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14(2):131–164,
37. Salt Security (2022) State of API Security Report Q3 2022. [Online; `https://content.salt.security/state-api-report.html`], accessed on February 23, 2023.
38. Santos Filho A (2023) Links2CPN - OpenAPI Links to CPNs. [Online; `https://github.com/ailton07/openapi-links-to-CPNs`], accessed on February 27, 2023.
39. Schoenborn JM, Althoff KD (2021) Detecting SQL-Injection and Cross-Site Scripting Attacks Using Case-Based Reasoning and SEASALT. In: LWDA, pp 66–77
40. Zuzak I, Budiselic I, Delac G (2011) Formal Modeling of RESTful Systems Using Finite-State Machines. In: Auer S, Díaz O, Papadopoulos GA (eds) Web Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 346–360

Appendix A: Excerpt of OpenAPI Used as a Running Example

Listing 6: Excerpt of OpenAPI Used as a Running Example

```yaml
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'
```