

Automated Vulnerability Assessment Approach for Web API that Considers Requests and Responses

Yuki Ishida*, Masaki Hanada**, Atsushi Waseda**, and Moo Wan Kim***

* Graduate School of Informatics, Tokyo University of Information Sciences, Japan

** Department of Informatics, Tokyo University of Information Sciences, Japan

*** TA Tech., Japan

h22001iy@edu.tuis.ac.jp, mhanada@rsch.tuis.ac.jp, aw207189@rsch.tuis.ac.jp, ykim5jp@ybb.ne.jp

Abstract— In recent years, Web Application Programming Interfaces (Web APIs) have been extensively used in numerous web applications. However, the number of attacks exploiting Web API vulnerabilities has been rapidly increasing. The Open Web Application Security Project (OWASP) published guidelines known as the OWASP API Security Top 10 to mitigate the risks associated with these vulnerabilities. The guidelines identify the top 10 most critical security risks in Web APIs and provide remediation guidance to help developers. Although developers are required to address these vulnerabilities according to these guidelines, traditional vulnerability assessment tools may not perform adequately when used to assess Web API vulnerabilities. Manually addressing these is difficult because there are a large number of endpoints and parameters in Web APIs using traditional vulnerability assessment tools. To address this issue, we propose a method for automatically conducting Web API vulnerability assessments by utilizing references, requests, and responses for Web APIs. In the evaluation experiment, we showed that the proposed method can detect authorization-related vulnerabilities in the Web APIs of vulnerable testing environments and well-known Content Management Systems, such as Wordpress, Ghost CMS, and Joomla.

Keywords— Web API, Vulnerability Assessment, Automation Analysis, Security

I. INTRODUCTION

IN recent years, the widespread adoption and use of Web Application Programming Interfaces (Web APIs) have greatly enhanced the convenience of web systems. However,

the vulnerabilities associated with Web APIs are increasingly subject to attacks. Akamai Inc. reported that more than 11 billion attacks occurred between January 2020 and June 2021 [1]. Specifically, in June 2021, 113.8 million attack traffic events were observed in a single day, and attacks targeting Web APIs are on the rise. The proactive elimination of Web API vulnerabilities is very important to guarantee a safe and secure cyberspace for general users.

To mitigate the risks associated with these vulnerabilities, the Open Web Application Security Project (OWASP) published guidelines known as the OWASP API Security Top 10 [2]. The guidelines identify the top 10 most critical security risks in Web APIs and provide remediation guidance to help developers. Although developers are required to address these vulnerabilities according to such guidelines, it is difficult to address them manually because of the large number of endpoints and parameters in Web APIs.

To solve these issues, traditional web applications have employed a variety of tools, such as OWASP ZAP (Zed Attack Proxy), for vulnerability assessment. However, because these tools are primarily designed for traditional web applications and content, they may not perform adequately when used to assess Web API vulnerabilities. Specifically, developers are required to manually configure many parameters of traditional tools (e.g., OWASP ZAP) for vulnerability assessment because traditional tools do not consider the logic and characteristics of Web APIs. A lack of understanding of the logic and characteristics of Web APIs causes omissions or mismatches in endpoints or parameters. Consequently, this issue leaves risks of Web API vulnerabilities.

In this study, we propose a method for automatically conducting a Web API vulnerability assessment by utilizing references, requests, and responses for Web APIs to detect authorization-related Web API vulnerabilities. The proposed method first generates endpoints and parameters using Web API references, and requests for validating the Web API are subsequently sent to the generated endpoints. Next, new endpoints and parameters are generated using the responses from the endpoints, and vulnerability assessment requests are sent to the generated endpoints. Finally, when the HTTP status codes of the responses from the endpoints satisfy certain conditions, the proposed method determines that they are valid endpoints and/or parameters and do not contain vulnerabilities.

Manuscript received on Sep. 27, 2023. This work is a follow-up on the accepted and presented paper titled "An Automated Vulnerability Assessment Approach for WebAPI that Considers Requests and Responses," at the 24th International Conference on Advanced Communication Technology (ICTACT2022).

Yuki Ishida, He is a Ph.D. student in the Graduate School of Informatics, Tokyo University of Information Sciences, 4-1, Onaridai, Wakaba, Chiba, Japan (e-mail: h22001iy@edu.tuis.ac.jp).

Masaki Hanada is with Department of Informatics, Tokyo University of Information Sciences, 4-1, Onaridai, Wakaba, Chiba, Japan (corresponding author, phone: +81-43-236-1224, e-mail: mhanada@rsch.tuis.ac.jp).

Atsushi Waseda is with Department of Informatics, Tokyo University of Information Sciences, 4-1 Onaridai, Wakaba-ku, Chiba, Japan (e-mail: aw207189@rsch.tuis.ac.jp).

Moo Wan Kim is with TA-Tech, Japan (e-mail: ykim5jp@ybb.ne.jp)

TABLE I
Example of RPC API and REST API

Operation	Type of API	HTTP Method	Endpoint	Request Parameters (JSON Format)
Read	RPC	POST	/getItem	{"id": "1"}
	REST	GET	/items/1	None
Create	RPC	POST	/createItem	{"name": "Pen", "price": 100}
	REST	POST	/Items	{"name": "Pen", "price": 100}

In the evaluation experiment, we showed that the proposed method can detect authorization-related vulnerabilities in the Web APIs of vulnerable testing environments and Content Management Systems (CMSs).

The remainder of this paper is organized as follows. Section II presents an overview of Web APIs. Section III presents the OWASP API Security Top 10. Traditional vulnerability assessment tools are presented in Section IV. Section V describes the details of the proposed method. Section VI presents the experimental environment. Section VII presents the experimental results. Finally, Section VIII concludes the paper.

II. WEB APIS

A Web APIs is an API that is accessed using the HTTP protocol and typically uses either Extensible Markup Language (XML) or JavaScript Object Notation (JSON) formats to encode data. Web APIs are extensively utilized to enhance the communication between web systems and a variety of web applications. Currently, despite the lack of a universal standardized format for Web APIs, Remote Procedure Call API (RPC API) and Representational State Transfer API (REST API) have been widely used as representative architectural styles in Web API design.

A. RPC API

The RPC API is based on the remote procedure call (RPC). The representative implementations of the RPC API are XML-RPC and JSON-RPC; whereas JSON-RPC uses a lightweight JSON format to encode data, XML-RPC uses a heavier XML format.

B. REST API

The REST API is a simple method for accessing Web resources. The REST API endpoint is a URL that utilizes HTTP methods such as POST, GET, POST, PUT, and DELETE to execute the CRUD (Create, Read, Update, and Delete) operations. It primarily focuses on providing resources from the server to clients. Similar to the RPC API, the REST API can use either XML or JSON to encode data.

Table I presents an example of HTTP requests to perform read and create operations in RPC API and REST API.

In this study, we targeted the REST API because it has become a mainstream Web API, and the Web APIs of many CMSs are provided by the REST API.

TABLE II
List of OWASP API Security Top 10

Risk ID	Name of Vulnerability items	Category
API 1:2019	Broken Object Level Authorization	Category 1
API 2:2019	Broken User Authentication	Category 1
API 3:2019	Excessive Data Exposure	Category 2
API 4:2019	Lack of Resources & Rate Limiting	Category 3
API 5:2019	Broken Function Level Authorization	Category 1
API 6:2019	Mass Assignment	Category 2
API 7:2019	Security Misconfiguration	Category 3
API 8:2019	Injection	Category 4
API 9:2019	Improper Assets Management	Category 3
API 10:2019	Insufficient Logging & Monitoring	Category 4

III. OWASP API SECURITY TOP 10

OWASP is a nonprofit foundation that works to improve the security of web applications. OWASP published guidelines known as OWASP API Security Top 10 to mitigate the risks associated with Web API vulnerabilities; these guidelines aimed to provide guidance on the most important security risks to consider when developing and exposing APIs. The guidelines include recommended countermeasures and outline scenarios in which Web API vulnerabilities may occur.

Table II lists the OWASP API Security Top 10, which can be classified into four primary categories. The *category 1* in Table II pertains to access control vulnerabilities, such as the unauthorized use of Web APIs by general users, owing to inadequate authorization settings intended for administrators. The *category 2* includes vulnerabilities related to data handling. For example, a Web API server may transmit data to the client while assuming that filtering will occur on the Web application side, or it may include undefined data within the request body. The *category 3* highlights vulnerabilities that may occur during the development and operational phases; this includes exposure to problematic APIs utilized during development and issues related to the security configurations of the Web API server. The *category 4* encompasses vulnerabilities typical of traditional web threats, such as injection attacks and incorrect logging configurations.

This study focuses on authorization-related vulnerabilities as outlined in *API 1:2019* and *API 5:2019*, both of which hold significant importance on this list. Additionally, we investigated the endpoints associated with *API 9:2019*. In this vulnerability, endpoints not defined in the API references are exposed because of a lack of proper authentication and authorization.

A. API 1:2019 Broken Object Level Authorization

API 1:2019 is an authorization-related vulnerability. Traditional web applications utilize a mechanism called *session* that stores users' authentication status for both web servers and clients. However, a Web API does not typically manage users' authentication status (i.e., stateless), and user management is often achieved by embedding an identifying flag at the endpoint. For example, the endpoint for accessing the information of a user with ID 1 is */user/1*. In such cases, if a Web API server fails to properly authorize access to user information, a malicious user can manipulate the user ID at the endpoint to access other users' information.

B. API 5:2019 Broken Function Level Authorization

API 5:2019 is also a vulnerability associated with the authorization process. Vulnerabilities occur when proper authorization is not set for each endpoint in the Web API server used by general and privileged users. Malicious users can attack a system by exploiting vulnerabilities in privileged accounts.

C. API 9:2019 Improper Assets Management

API 9:2019 is associated with improper data handling. Exposed debug endpoints and deprecated API versions can increase potential security risks. For example, if web systems have been updated to use a new API and the old endpoint remains operational, it could provide an accessible point for a malicious user.

IV. VULNERABILITY ASSESSMENT TOOL

Several tools are available for detecting Web API vulnerabilities. However, these vulnerability assessment tools cannot detect the ten vulnerabilities described in the OWASP API Security Top 10. The reason for this failure is that they attempted to detect vulnerabilities by sending predetermined requests and could not specify the authentication information for each target Web API. In addition, various request body data (hereinafter referred to as "parameters") used for vulnerability assessment are set by a user who uses these tools, and if the user cannot set appropriate parameters, the assessment becomes difficult. The features of the three representative vulnerability assessment tools and the vulnerability items of the OWASP API TOP 10 that can be assessed are described below.

A. Automatic API Attack Tool

Automatic API Attack Tool [3] can flexibly generate vulnerability detection test cases according to the Web APIs to be vulnerability-tested by loading the Web API reference in JSON or YAML formats. For example, for an endpoint that uses a parameter of "INT type" as "id," this tool will change the value of "id" to a numeric value of another type, such as long or double, and send a vulnerability detection request. This tool can detect vulnerabilities in API 1:2019, API 5:2019, and API 9:2019 of the OWASP API Security Top 10. However, Web API references in JSON or YAML formats describing the Web API information are required, and vulnerability assessments may require considerable work and time.

B. Vooki, Rest API Scanner

Vooki, Rest API Scanner [4] is a vulnerability assessment tool that sends requests to detect vulnerabilities in API 3:2019, API 7:2019, and API 8:2019 of the OWASP API Security Top 10. However, all endpoints, headers, and parameters used in the Web API must be configured by the user of this tool. If the necessary information for vulnerability assessment cannot be set, vulnerabilities may not be detected, even if they exist.

C. OWASP ZAP

The *OWASP Zed Attack Proxy (OWASP ZAP)* [5] is one of the most representative tools used for vulnerability scanning. Primarily, OWASP ZAP is designed to scan vulnerabilities in web systems. However, it can read references written in the OpenAPI format and conduct vulnerability scans of Web APIs by incorporating an add-on or utilizing a command-line tool.

D. Problem of Traditional Tools

One of the major issues with these traditional vulnerability assessment tools is that the endpoints must be explicitly defined by developers. If any endpoint is omitted, this may lead to inaccurate vulnerability assessments. Therefore, developers must perform frequent maintenance and ensure that all endpoints are correctly and comprehensively defined.

V. PROPOSED METHOD

As we mentioned in Section IV, these traditional vulnerability assessment tools require developers to provide endpoint information to the vulnerability tools, which then scan the given endpoint. Since there are many endpoints and parameters in Web APIs, it is difficult for developers to configure them manually. In addition, the endpoints and parameters continue to change because web systems undergo continuous improvements after completion.

In this study, we propose a method for automatically conducting a Web API vulnerability assessment by utilizing references, requests, and responses for Web APIs to detect authorization-related Web API vulnerabilities. First, in the proposed method, if authentication information is required, developers (or vulnerability diagnosticians) set the authentication information to the HTTP request (e.g., HTTP header) according to the API references, and each validation check starts.

The proposed method was executed in three steps.

Step 1: Obtaining References

We collected information about Web API references to automatically generate request queries for vulnerability assessment.

Step 2: Validation Check of API References

We obtained the response information by constructing a request query using the Web API information obtained from Step 1. The constructed request query is sent to the endpoints to validate the Web APIs and the response (i.e., JSON data) is stored.

Step 3: Vulnerability Detection

To generate a query for vulnerability assessment, we define the string to be used as a *candidate key* and extract the values of the key, as well as from the JSON data stored in Step 2. Vulnerability assessment queries are generated from the *candidate key*, and the vulnerability detection process begins. If the HTTP status code corresponding to each vulnerability assessment query is 200, the proposed method indicates the existence of a vulnerability.

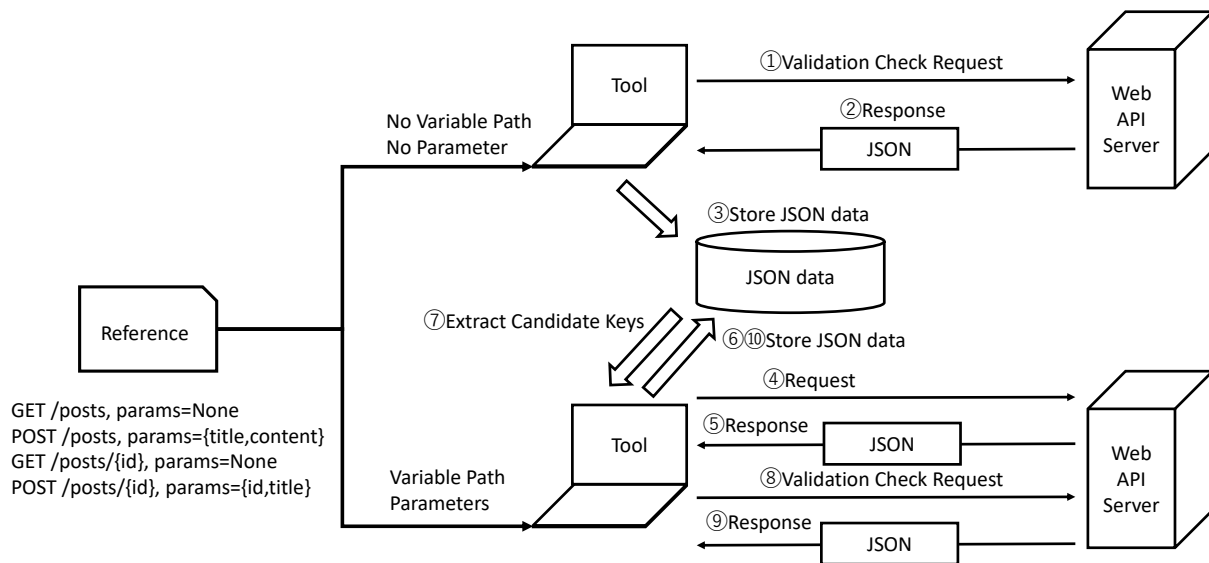


Fig. 1. API Validation Flow

 TABLE III
 Example of Web API References

No	HTTP Method	Endpoint	Request Parameters		Authentication
			Name	Type	
1	GET	/posts	None		Non-required
2	POST	/posts	title content	string string	Non-required
3	GET	/posts/{id}	None		Non-required
4	POST	/posts/{id}	id title	integer string	Non-required

A. Step 1: Obtaining References

Step 1 is to collect information about Web API references, such as the HTTP method, endpoints, parameters, and their types using **Method 1** and **Method 2**. Examples of the information about the Web API references are listed in Table III. In Table III, *{id}* is variable which can be replaced. Hereafter, the */posts* is referred to as the *basic path* and *{id}* is the *variable path*. These references were obtained using **Method 1** and **Method 2**.

Method 1: This method uses SwaggerHub [6], a resource-rich platform for Web API developers. SwaggerHub provides a wealth of references for Web APIs available in either JSON or YAML formats.

Method 2: This method employs the use of official websites. Automatic extraction of information from the HTML source becomes necessary because Web API references are provided in HTML format. In this study, we extracted information regarding Web API references from webpages written in HTML by adapting the methodology employed in a previous study [7].

If a reference does not exist in SwaggerHub (i.e., **Method 1**), the proposed method obtains the reference from the official websites of the Web APIs (i.e., **Method 2**).

B. Step 2: Validation Check of API References

Step 2 validates the Web APIs (i.e., endpoints and parameters) described in the API references. Request queries for validation are generated based on information about the API references and sent to the endpoints of the Web APIs. An overview of the validation process is presented in Figure 1. Table III shows an example of the information regarding API references.

The validation process for the Web APIs is as follows:

- 1) The validation check for endpoints without *variable path* and request parameters is first conducted. In Table III, the No.1 API, which has */posts* endpoint, is first selected, and the request query without request parameters is sent to the */posts* endpoint (① in Figure 1). After confirming the validation (i.e., HTTP Status Code is 200), response parameters (i.e., JSON data) that are returned from the endpoint are obtained and stored (② and ③ in Figure 1).

- 2) The validation check for endpoints including *variable path* without request parameters is conducted. In Table III, the No.3 API, which has */posts/{id}* endpoint, is selected. The request query is sent to the endpoint */posts* which is an endpoint one level up from */posts/{id}* (④ in Figure 1). Response parameters (i.e., JSON data) that are returned from the endpoint are obtained and stored (⑤ and ⑥ in Figure 1).

Next, the proposed method searches for a parameter with the key *id* in the response parameters. When the keys *id* are found in the response parameters (⑦ in Figure 1), the *variable path {id}* is replaced with the corresponding value for the first key *id*.

For example, if the response parameters displayed in Figure 2 are obtained, the endpoint of the next request query is */posts/38*, and this query is sent to the */posts/38* endpoint (⑧ in Figure 1). If there is no endpoint one level up or the same key, the *variable path {id}* is

```
[
  { "id": 38, "title": "ArticleName1" },
  { "id": 1, "title": "ArticleName2" }
]
```

Fig. 2. Example of response parameters from /posts endpoint in No. 3 API

```
{
  { "title": "ArticleName1", "content": "Content1" },
  { "title": "ArticleName2", "content": "Content2" }
}
```

Fig. 3. Example of response parameters from /posts endpoint in No. 2 API

replaced with the fixed value 1 (i.e., default value).

After confirming the validation (i.e., HTTP Status Code is 200), response parameters (i.e., JSON data) that are returned from the endpoint are obtained and stored (⑨ and ⑩ in Figure 1).

- 3) The validation check for endpoints, including request parameters without *variable path*, is conducted. In Table III, the No.2 API, which has /posts endpoint and the request parameter with the keys *title* and *content* is selected, and the request query without request parameters is sent to the /posts endpoint using HTTP GET Method (④ in Figure 1). Response parameters (i.e., JSON data) that are returned from the endpoint are obtained and stored (⑤ and ⑥ in Figure 1).

When the key *title* and *content* are found in the response parameters (⑦ in Figure 1), the keys *title* and *content* of the request parameter are set to the corresponding value for the keys *title* and *content*.

For example, if the response parameters displayed in Figure 3 are obtained, the endpoint and parameter are /posts and { "title": "ArticleName1", "Content1": "content1" } respectively (⑦ in Figure 1). This query is sent to the /posts endpoint (⑧ in Figure 1). If there is not the same key of the request parameters, the keys *title* and *content* of the request parameter are set to the fixed value "a" (i.e., default value).

After confirming the validation (i.e., HTTP Status Code is 200), response parameters (i.e., JSON data) that are returned from the endpoint are obtained and stored (⑨ and ⑩ in Figure 1).

- 4) The validation check for endpoints including *variable path* and request parameters is conducted. In Table III, the No.4 API, which has /posts/{id} endpoint and the request parameter with the keys *id* and *title*, is selected. Similar to the above *process 2*), when the keys *id* is found in the response parameters, the *variable path* {id} is replaced with the corresponding value for the first key *id* (④ – ⑦ in Figure 1). Additionally, similar to the above *process 3*), when the key *title* is found in the response parameters, the key *title* of the request parameter is set to the corresponding value for the key *title* (④ –

⑦ in Figure 1).

For example, if the response parameters displayed in Figure 2 are obtained, the endpoint and parameter are /posts/38 and { "id": 38, "title": "ArticleName1" } respectively (⑦ in Figure 1). This query is sent to the /posts/38 endpoint (⑧ in Figure 1). If there is no endpoint one level up or the same key in the request parameters, the *variable path* {id} is replaced with the fixed value 1 (i.e., /posts/1), and the keys *id* and *title* of the request parameter are set to the fixed values 1 and "a" (i.e., { "id": 1, "title": "a" }). After confirming the validation (i.e., HTTP Status Code is 200), response parameters (i.e., JSON data) that are returned from the endpoint are obtained and stored (⑨ and ⑩ in Figure 1).

The above four processes (*process 1*) - *process 4*)) are conducted in order from the *process 1*).

After completing the validation check for all endpoints, the vulnerability detection process detailed in **Step 3** begins.

C. Step 3: Vulnerability Detection

The objective of **Step 3** is to detect authorization-related Web API vulnerabilities. This step generates two types of vulnerability assessment queries.

- The first query aims to detect the existence of endpoints unintended for exposure by the developers. This query is basically for API 1:2019 in the OWASP API Security Top 10.
- The second query aims to detect vulnerabilities within known endpoints by replacing the *variable path* and changing parameters using the JSON data stored in **Step 2**. This query is basically for API 5:2019 in the OWASP API Security Top 10.

An overview of the vulnerability assessment query generation process is shown in Figure 4. To generate vulnerability assessment queries, we must first extract strings to replace the *variable path* and change the parameters from the JSON data stored in **Step 2**.

Candidate keys comprise the following two types of words:

- Words that are used as the key names and values in the JSON data.
- Words that are included in the endpoints provided by the API references.

When strings that are used as the key names and values in the JSON data are provided as text, we separate the texts into words using morphological analysis. For example, the words "id," "1," "title," and "ArticleA" are extracted from the JSON data in ④ of Figure 4.

1) *Queries for Detection of Endpoints Including Unintended Exposure*: To detect endpoints, including unintended exposure, we need to estimate unintended endpoints for developers. The *candidate keys* are further classified into two types based on morpheme analysis. One is all noun words except numerals, and the other is numerals (e.g., 0, 1, ...). For example, in ⑤ in Figure 4, the noun words "id," "title," and "Article" are extracted from the words "id," "1," "title,"

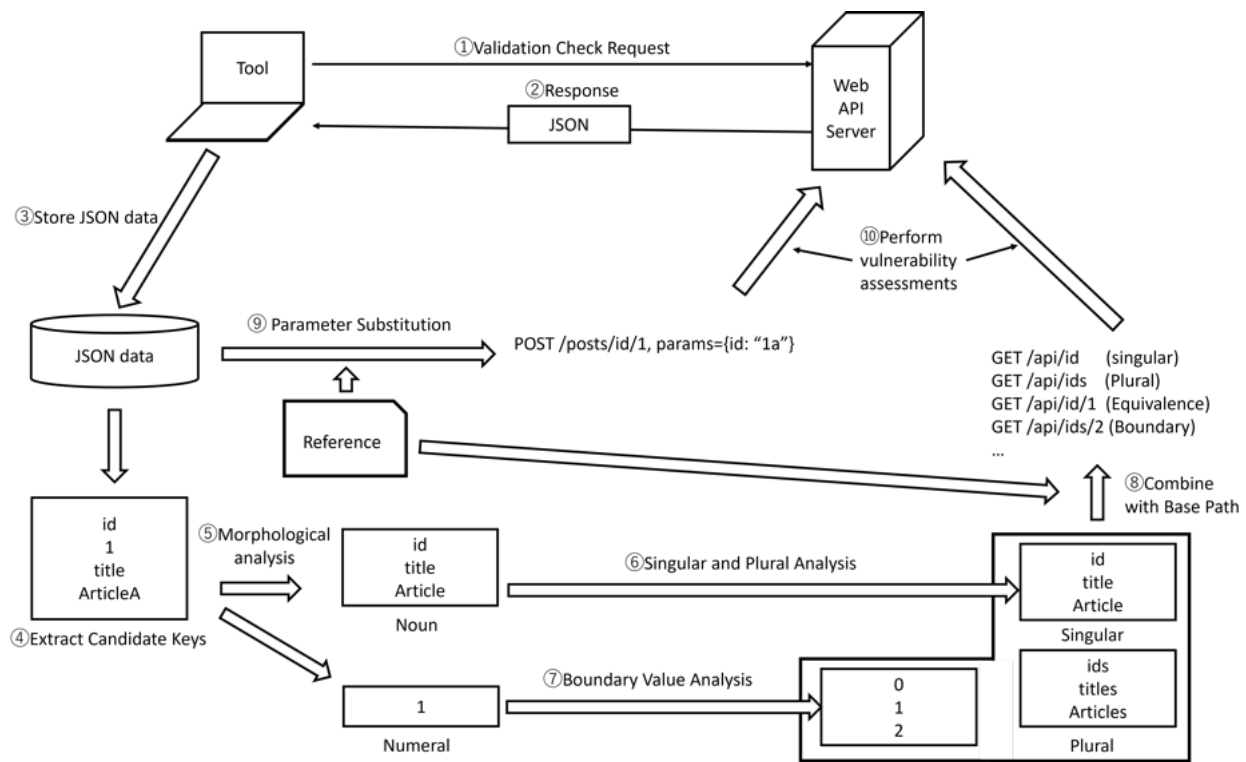


Fig. 4. Vulnerability Assessment Query Generation Flow

and "ArticleA." Additionally, the numeral 1 is extracted from the words "id," "1," "title," and "ArticleA."

For nouns, the proposed method adapts both singular and plural forms of words to detect endpoints, including unintended exposures. If only a plural form of the word is extracted, the proposed method transforms it into a singular form, and vice versa. The words "id," "ids," "title," "titles," "Article," and "Articles" are transformed from the words "id," "1," "title," and "Article" in ⑥ in Figure 4.

In the case of a numeral, given a set of numerals, the maximum value, the maximum value ± 1 , the minimum value, and the minimum value ± 1 are adapted according to the boundary value analysis to detect endpoints, including unintended exposure. For example, numerals 0, 1, and 2 are generated from numeral 1 in ⑦ of Figure 4.

Finally, by the above-mentioned processes, we can obtain the numerals 0, 1 and 2 and the noun words "id," "ids," "title," "titles," "Article," and "Articles" are generated in ⑥ and ⑦ of Figure 4.

The vulnerability assessment queries were constructed from these numerals and nouns. In the proposed method, the four combinations of vulnerability assessment queries (⑧ in Figure 4) are as follows:

- <API entry point>/<plural noun word>
- <API entry point>/<singular noun word>
- <API entry point>/<plural noun word>/<number>
- <API entry point>/<singular noun word>/<number>

The API entry point represents the base path obtained from API references. For example, in Table IV, /api is defined as API entry point. The API entry point is the /api, which is one

 TABLE IV
Example of Web API References (Non-parameter)

No	HTTP Method	Endpoint	Request Parameters		Authentication
			Name	Type	
1	GET	/api	None		Non-required
2	GET	/api/{id}	None		Non-required

level away from the /api/{id} if the variable path is included.

Finally, if the numerals 1 and 2, and the noun words "id" and "ids" are given, the endpoints of the vulnerability assessment queries are generated as follows:

- /api/id
- /api/ids
- /api/id/1
- /api/ids/1
- /api/id/2
- /api/ids/2

The vulnerability assessment queries which has these endpoints are illustrated in No.1 – No.6 of Table VI.

2) *Queries for Detection of Vulnerabilities within Known Endpoints Using Replacement and Changing:* To detect vulnerabilities in terms of parameters within known endpoints, we must estimate the resources that are not managed at the configuration or code levels. The proposed method first uses the values for the keys of the request parameters using JSON data, which are stored in Step 2, and the API references. Next, the proposed method changes the type of parameters and generates new values for the keys of the request parameters for vulnerability assessment using the JSON data and API

TABLE V
Example of Web API References (Parameters)

No	HTTP Method	Endpoint	Request Parameters		Authentication
			Name	Type	
1	POST	/posts	id	integer	Non-required
2	POST	/posts	status	string	Non-required

```
{
  "id": 5
},
{
  "status": "publish"
}
```

Fig. 5. Example of JSON data

references (9) of Figure 4).

We explain the generation of parameters for vulnerabilities using the API references listed in Table V and the JSON data illustrated in Figure 5.

If a key has a numeric type in the API references, three values for the key are generated: first, the same value as that of the numeric type is set; second, the fixed value "a" as that of the string type is set; third, the proposed method changes the numeric type to the string type keeping the numeral, and concatenates the numeral of the string type and the fixed value "a" of the string type. For example, if the key "id" is 5 (i.e., {"id":5}), the proposed method first uses {"id":5}. Second, the proposed method generates {"id":"a"} where "a" is used as the fixed value of string type in this study. Third, the proposed method generates {"id":"5a"} after concatenating "5" and "a" of the string type.

Finally, the request parameters {"id":5}, {"id":"a"} and {"id":"5a"} are generated from {"id":5}. The vulnerability assessment queries which has these parameters are illustrated in No.7 – No.9 of Table VI.

If a key has a string type in the API references, the following three values for the key are generated. First, the same value of string type is set. Second, when the value of the string type is a numeral, the numeral is converted to a string type; otherwise, the value "1" is used as the fixed value of the string type in this study. Third, the proposed method concatenates the original value of the string type and the fixed value "1" of the string type. For example, if the key "status" is "publish" (i.e., {"status":"publish"}), the proposed method first uses {"status":"publish"}. Second, the proposal method replaces "publish" with the fixed value "1" (i.e., {"status":"1"}). Third, the proposal method replaces "publish" with "publish1" (i.e., {"status":"publish1"}).

Finally, the request parameters {"status":"publish"}, {"status":"1"} and {"status":"publish1"} are generated from {"status":"publish"}. The vulnerability assessment queries which has these parameters are illustrated in No.10 – No.12 of Table VI.

3) *Vulnerability Detection*: The above-mentioned vulnerability assessment queries (Table VI) were generated, and the vulnerability detection process was initiated.

The HTTP status code of the response is used to determine the vulnerability assessment results. If the HTTP status code corresponding to each vulnerability assessment query is 200,

TABLE VI
Example of the vulnerability assessment queries

No	HTTP Method	Endpoint	Request Parameters
1	GET	/api/id	None
2	GET	/api/ids	None
3	GET	/api/id/1	None
4	GET	/api/ids/1	None
5	GET	/api/id/2	None
6	GET	/api/ids/2	None
7	POST	/posts	{"id" : 5}
8	POST	/posts	{"id" : "a"}
9	POST	/posts	{"id" : "5a"}
10	POST	/posts	{"status" : "publish"}
11	POST	/posts	{"status" : "1"}
12	POST	/posts	{"status" : "publish1"}

the proposed method indicates the existence of a vulnerability. Additionally, if the HTTP status code is 501, the proposed method indicates the existence of a vulnerability because the API server attempts to perform certain processes. When the HTTP status codes are not equal to 200 and 501, the proposed method indicates the absence of vulnerabilities.

VI. EXPERIMENT ENVIRONMENT

We conducted a preliminary experiment using actual environment to evaluate whether the proposed method can detect the OWASP API Security Top 10 vulnerabilities. We used the well-known CMSs as the experimental environments. In our selection criteria, we focus on whether the CMSs include known vulnerabilities and whether they have higher market shares [8]. Consequently, we selected three well-known CMSs and a vulnerable training environment.

- vAPI [9]
- WordPress [10]
- Ghost CMS [11]
- Joomla [12]

A. vAPI

vAPI was developed to test the vulnerabilities in the OWASP API Security Top 10. It replicates the vulnerabilities described in the OWASP API Security Top 10, serving as an application that enables users to experience attacks.

B. WordPress

WordPress is a leading CMS with a market share of 63.7% as of January 2023, according to a report by W3Techs [13]. It is utilized by global organizations, such as Microsoft, Mozilla, and Apache. However, it has been targeted by many cyber attackers owing to its widespread use. WordPress versions 4.7.0 and 4.7.1 contain a vulnerability that allows article tampering by bypassing authentication [14]. This risk was categorized as *API 1:2019* in the OWASP API Security Top 10.

C. Ghost CMS

Ghost CMS has a relatively modest market share of 0.08% compared with the other CMSs. However, it has steadily gained traction as an emerging CMS [15]. Additionally, it has been adopted by well-known websites such as Cloudflare and Duolingo. Ghost CMS versions from 4.0.0 to 4.9.4 contain vulnerabilities that allow an undisclosed endpoint to access information that typically requires administrative privileges [16]. This vulnerability was categorized as *API 5:2019* in the OWASP API Security Top 10.

D. Joomla

Joomla has a small market share of 2.7% as of January 2023, according to the report of W3Techs [17]. Despite its small market share, it is renowned for its strong emphasis on security and is frequently used on government websites.

VII. EXPERIMENT RESULTS

We conducted a preliminary experiment using well-known CMSs, such as WordPress, Ghost CMS, and Joomla. We also used a deliberately vulnerable testing environment called vAPI. For vAPI, the proposed method could detect vulnerabilities (*API 1:2019* and *API 5:2019* of the OWASP API Security Top 10) because vAPI was developed to test vulnerabilities in the OWASP API Security Top 10. WordPress has a known vulnerability (*API 1:2019* of the OWASP API Security Top 10), and Ghost CMS has a known vulnerability (*API 5:2019*). Therefore, in this study, we first evaluate whether these known vulnerabilities can be detected. Additionally, the proposed method attempts to detect vulnerabilities that are not known (i.e., vulnerabilities where CVE ID are not assigned). CVE (Common Vulnerabilities and Exposures) is a list of publicly disclosed vulnerabilities and each vulnerability is assigned a CVE ID number. Joomla does not have a known vulnerability described in the OWASP API Security Top 10, and the proposed method attempts to detect vulnerabilities that are not known (i.e., vulnerabilities where CVE ID are not assigned).

A. Vulnerability Detection for vAPI

First, we evaluated whether the proposed method could detect *API 1:2019* and *API 5:2019* from the OWASP API Security Top 10. Given that vAPI provides a webpage with API references for each endpoint, we used this information to create endpoint-specific settings.

1) *Detection for API 1:2019*: *API 1:2019* of the vAPI contains a vulnerability in the authorization process. This vulnerability allows an authenticated user to access information regarding other users. First, in the proposed method, developers (or vulnerability diagnosticians) set the authentication information to the HTTP request (e.g., HTTP header) according to the API references, and **Step 1** begins. In **Step 1**, we can obtain the API references listed in Table VII. The number of APIs in *API 1:2019* of the vAPI was 3. In **Step 2**, the No.1 API which has the */vapi/api1/user/{api1_id}* endpoint, is first selected. *{api1_id}* is not included in the response parameter from */vapi/api1/user*, which is one level up

TABLE VII
Web API References (*API 1:2019* of vAPI)

No	HTTP Method	Endpoint	Request Parameters		Authentication
			Name	Type	
1	GET	<i>/vapi/api1/user/{api1_id}</i>	None		Required
2	POST	<i>/vapi/api1/user</i>	username name course password	string string string string	Required
3	PUT	<i>/vapi/api1/user/{api1_id}</i>	username name course password	string string string string	Required

TABLE VIII
Number of Candidate Keys for *API 1:2019* of vAPI

Item	Value
The Number of String Candidate Keys	18
The Number of Numerical Candidate Keys	5

TABLE IX
Part of Candidate Keys for *API 1:2019* of vAPI

String Candidate Key	Numerical Candidate Key
site	0
id	1
entry	9
user	237235
error	2324
username	

```
GET http://[Server IP Address]/vapi/api1/user/1
Content-Type: application/json
Authorization-Token: (** mask **)
```

Fig. 6. Request for *API 1:2019* of vAPI

```
{
  "id": 1,
  "username": "michaels",
  "name": "Michael Scott",
  "course": "flag{api1_(** mask **)}"
}
```

Fig. 7. Response for *API 1:2019* of vAPI

from */vapi/api1/user/{api1_id}*. Therefore, the *variable path {api1_id}* is replaced by the default value of 1, and a request with authorization information is sent to the */vapi/api1/user/1* endpoint. Similarly, the No.2 API and 3 API are selected, and a request with authorization information was sent to the endpoints. After completing **Step 2**, the number of *candidate keys* is shown in Table VIII is obtained from the JSON data. Table IX presents part of the *candidate keys*. In **Step 3**, the request with the authorization information shown in Figure 6 is sent to the */vapi/api1/user/1* endpoint, which is constructed using the *candidate keys* of the *user* and 1, and the response shown in Figure 7 is obtained. Because the vAPI issues a flag when an attack is successful and an authenticated user can access information about another user (i.e., id = 1), the proposed method indicates the existence of the *API 1:2019* vulnerability. In this step, 2714 vulnerability assessment queries were sent to the vAPI server.

```
GET /vapi/api5/users
Content-Type: application/json
Authorization-Token: (** mask **)
```

Fig. 8. Request for API 5:2019 of vAPI

```
{
  "id": 1,
  "username": "admin",
  "name": "Admin User",
  "address": "flag{api5_ (** mask **)}",
  "mobilen": "8080808080"
}
```

Fig. 9. Response for API 5:2019 of vAPI

2) *Detection for API 5:2019*: API 5:2019 implementation of the vAPI also exposes a vulnerability in the authorization process. This vulnerability permits general users to access confidential endpoints at which administrator privileges are required. If this endpoint is accurately deduced, a list of users with administrative privileges can be procured. First, in the proposed method, developers (or vulnerability diagnosticians) set the authentication information to the HTTP request (e.g., HTTP header) according to the API references, and **Step 1** begins. In **Step 1**, we can obtain the API references listed in Table X. The number of APIs in API 5:2019 of the vAPI was 2. In **Step 2**, the No.1 API which has the `/vapi/api5/user/{api5_id}` endpoint, is first selected. `{api5_id}` is not included in the response parameter from `/vapi/api5/user`, which is one level up from `/vapi/api5/user/{api5_id}`. Therefore, the *variable path* `{api5_id}` is replaced by the default value of `1`, and a request with authorization information is sent to the `/vapi/api5/user/1` endpoint. Similarly, the No.2 API are selected, and a request with authorization information was sent to the endpoints. After completing **Step 2**, the number of *candidate keys* shown in Table XI is obtained from the JSON data. Table XII presents part of the *candidate keys*. In **Step 3**, the request with the authorization information shown in Figure 8 is sent to the `/vapi/api5/users` endpoint, which is constructed by the *candidate keys* of *users*, which is a plural of the singular form *user*, as shown in Figure 8. Because the vAPI issues a flag when an attack is successful, and a general user can access the confidential endpoint for which administrator privileges are needed, the proposed method indicates the existence of the API 5:2019 vulnerability. In this step, 15586 vulnerability assessment queries were sent to the vAPI server. The number of queries was relatively large because the number of generated queries rapidly increased as the number of parameters increased.

B. Vulnerability Detection for WordPress

The 4.7.0 version of WordPress has the privilege vulnerability identified as CVE-2017-1001000. This vulnerability prevails in the authorization process of the `/posts/{id}` endpoint. If a request parameter including an invalid content identifier (i.e., a string type value composed of a numeral and string) is sent to

TABLE X
Web API References (API 5:2019 of vAPI)

No	HTTP Method	Endpoint	Request Parameters		Authentication
			Name	Type	
1	GET	<code>/vapi/api5/user/{api5_id}</code>	None		Required
2	POST	<code>/vapi/api5/user</code>	username password name address mobilen	string string string string string	Required

TABLE XI
Number of Candidate Keys for API 5:2019 of vAPI

Item	Value
The Number of String Candidate Keys	6
The Number of Numerical Candidate Keys	0

TABLE XII
Part of Candidate Keys for API 5:2019 of vAPI

String Candidate Key
u
false
s
cause
user

the `/posts/{id}` endpoint, it becomes possible to overwrite the content. This vulnerability was categorized as API 1:2019 and API 2:2019 in the OWAPS API Security Top 10. We evaluated whether this vulnerability and other unknown vulnerabilities could be detected by implementing the proposed method.

First, in the proposed method, developers (or vulnerability diagnosticians) set the authentication information to the HTTP request (e.g., HTTP header) according to the API references, and **Step 1** begins. In **Step 1**, we obtain the API references listed in Table XIII. The number of APIs that required authorization was 47. Part of these APIs are shown in Table XIII. In **Step 2**, the No.1 API which has the `/wp-json/wp/v2/posts/{post_id}` endpoint, is selected. `{post_id}` is not included in the response parameter from `/wp-json/wp/v2/posts`, which is one level up from `/wp-json/wp/v2/posts/{post_id}`. Therefore, the *variable path* `{post_id}` is replaced by the default value of `1`, and a request with authorization information is sent to the `/wp-json/wp/v2/posts/1` endpoint. Similarly, other APIs are selected, and a request with authorization information is sent to the endpoints. After completing **Step 2**, the number of *candidate keys* shown in Table XIV is obtained from the JSON data. Table XV presents an example of the *candidate keys*. In **Step 3**, the request with authorization information shown in Figure 10 with the parameter `{"id": "1a", "title": "a"}` is sent to the `/wp-json/wp/v2/posts/1` endpoint and the response shown in Figure 11 is obtained. The method for generating the parameter `{"id": "1a", "title": "a"}` is described in Section V-C2. As illustrated in Figure 11, the value of the "title" are tampered to "a." This result is considered to be the vulnerability identified as CVE-2017-1001000. Additionally, other unknown vulnerabilities were not detected using the

TABLE XIII
Part of Web API References (WordPress)

No	HTTP Method	Endpoint	Request Parameters		Authentication
			Name	Type	
1	GET	/wp-json/wp/v2/posts/{post_id}	None		Required
2	POST	/wp-json/wp/v2/posts	id	integer	Required
			title	string	
			content	string	
			status	string	
3	GET	/wp-json/wp/v2/posts/{parent}/revisions/{id}	None		Required
..	..	..	..		..

TABLE XIV
Number of Candidate Keys for WordPress

Item	Value
The Number of String Candidate Keys	88
The Number of Numerical Candidate Keys	29

TABLE XV
Part of Candidate Keys for WordPress

String Candidate Key	Numerical Candidate Key
site	0
id	1
http	27079
message	27080
world	27081
setting	

```
POST /wp-json/wp/v2/posts/1
Content-Type: application/json

{"id": "1a", "title": "a"}
```

Fig. 10. Request for WordPress

```
{
  "id": 1,
  "date": "2023-07-07T20:11:50",
  "modified": "2023-07-18T22:27:16",
  "slug": "hello-world",
  "type": "post",
  "title": {
    "raw": "a",
    "rendered": "a"
  },
}
```

Fig. 11. Response for WordPress

proposed method. In this step, 5288 vulnerability assessment queries were sent to the WordPress server.

C. Vulnerability Detection for Ghost CMS

Ghost CMS versions from 4.0.0 to 4.9.4 contain a vulnerability pertinent to inadequate authorization management. This susceptibility permits the extraction of an administrator API key, including a response to a specific API, thereby enabling privilege escalation for any authenticated user. This vulnerability was categorized under *API 5:2019* of the OWASP API Security Top 10.

First, in the proposed method, developers (or vulnerability diagnosticians) set the authentication information to the HTTP request (e.g., HTTP header) according to the API references, and **Step 1** begins. In **Step 1**, we can obtain the API references listed in Table XVI. The number of APIs that required authorization was 29. Part of these APIs are shown in Table XIII. In this experiment, we used version 2 of the API of Ghost CMS, and the *variable path* {*version*} was replaced with v2. In **Step 2**, the No.1 API, which contains the */ghost/api/v2/admin/posts* endpoint, is selected, and a request with authorization information is sent to the */ghost/api/v2/admin/posts* endpoint. Similarly, other APIs are selected, and a request with authorization information is sent to the endpoints. After completing **Step 2**, the number of *candidate keys* shown in Table XVII is obtained from the JSON data. Table XVIII presents part of the *candidate keys*. In **Step 3**, a request with authorization information, as shown in Figure 12 is sent to the */ghost/api/v2/admin/users* endpoint, and the response shown in Figure 13 is obtained. Consequently, 11 endpoints were identified as vulnerabilities. The 11 endpoints are listed in Table XIX. The list of users was procured using the administrator endpoint. Among the 11 endpoints, the */admin/site/* endpoint was defined as accessible by general users in the API references. Therefore, 10 endpoints were identified as vulnerabilities. In particular, the */admin/session/* and */admin/integrations/* endpoints were not described in the API references. The result was considered a vulnerability, identified as CVE-2021-39192. In this step, 2108 vulnerability assessment queries were sent to the Ghost CMS server.

D. Vulnerability Detection for Joomla

In the Joomla Web API, no obvious vulnerabilities associated with the OWASP API Security Top 10 were reported. However, the proposed method attempted to detect vulnerabilities that are not known (i.e., vulnerabilities where CVE ID are not assigned).

First, in the proposed method, developers (or vulnerability diagnosticians) set the authentication information to the HTTP request (e.g., HTTP header) according to the API references, and **Step 1** begins. Similar to the detections for vAPI, WordPress, and Ghost CMS, **Steps 1, 2, and 3** were processed. After completing **Step 2**, the number of *candidate keys* shown in Table XX were obtained from the JSON data. Table XXI presents part of the *candidate keys*. In **Step 3**, a request with the authorization information shown in Figure 14 is sent to the */api/index.php/v1/extensions*

TABLE XVI
Part of Web API References (Ghost CMS)

No	HTTP Method	Endpoint	Request Parameters		Authentication
			Name	Type	
1	GET	/ghost/api/{version}/admin/posts		None	Required
2	GET	/ghost/api/{version}/admin/posts/{id}		None	Required
3	POST	/ghost/api/{version}/admin/posts	title	string	Required
4	GET	/ghost/api/{version}/admin/settings		None	Required
5	GET	/ghost/api/{version}/admin/posts/{parent}/revisions/{id}		None	Required

```
GET /ghost/api/v2/admin/users/
Content-Type: application/json
Authorization: Ghost (** mask **)
```

Fig. 12. Request for Ghost CMS

```
{
  "users": [ {
    "id": "1",
    "name": "Yuki Ishida",
    "slug": "yuki",
    "email": "** mask **",
    "profile_image": null,
    "cover_image": null,
    "bio": null,
    "website": null,
    "location": null,
    "facebook": null,
    "twitter": null,
    "accessibility": null,
    "status": "active",
    "meta_title": null,
    "meta_description": null,
    "tour": null,
    "last_seen": "2023-07-22T10:53:11.000Z",
    "created_at": "2023-05-06T08:14:32.000Z",
    "updated_at": "2023-07-22T10:53:11.000Z",
    "url": "http://[Server IP Address]/404/"
  }, ... ],
}
```

Fig. 13. Response for Ghost CMS

```
POST /api/index.php/v1/extensions
Content-Type: application/json
Authorization: Bearer (** mask **)
```

Fig. 14. Request for Joomla

endpoint, and the response shown in Figure 15 was obtained.

Consequently, 3 unpublished endpoints were identified as vulnerabilities because the endpoints were not described in the API references. The 3 unpublished endpoints are listed in Table XXII. This vulnerability was categorized under *API 9:2019* of the OWAPS API Security Top 10. In particular, the */api/index.php/v1/extensions* endpoint illustrated in Figure 15 shows a catalog of extended functions installed within Joomla. This vulnerability implies the potential risk of exploiting the system configuration information if privilege escalation is perpetrated owing to other vulnerabilities.

TABLE XVII

Number of Candidate Keys for Ghost CMS

Item	Value
The Number of String Candidate Keys	974
The Number of Numerical Candidate Keys	887

TABLE XVIII

Part of Candidate Keys for Ghost CMS

String Candidate Key	Numerical Candidate Key
sell	0
pages	1
tags	40633578
upload	40633579
site	40633580
session	

TABLE XIX

Vulnerability Assessment Results for Ghost CMS

Endpoint	HTTP Status Code
/admin/pages/	200
/admin/tags/	200
/admin/posts/	200
/admin/roles/	200
/admin/users/	200
/admin/settings/	200
/admin/site/	200
/admin/users/1/	200
/admin/themes/	200
/admin/session/	200
/admin/integrations/	200

E. System Impact of Vulnerability Assessment using the Proposed Method

We evaluated the impact of each CMS in terms of system load. Table XXIII lists the number of endpoints recorded in the API reference for each CMS, the number of *candidate keys* extracted from JSON data, and the number of vulnerability assessment queries.

Compared with the number of queries between WordPress and Ghost CMS, although the number of endpoints in WordPress is greater than that in Ghost CMS, the number of *candidate keys* in WordPress is lower than that in Ghost CMS. One reason for this is that there are more types of strings in keys and values in WordPress is more than that in Ghost CMS. However, although the number of *candidate keys* in WordPress was less than that in Ghost CMS, the number of vulnerability queries in WordPress was greater than that in Ghost CMS. One of the reasons for this is that the number of keys and values in WordPress is greater than that in Ghost CMS. Specifically, although the number of endpoints and *candidate keys* in the

```

{
  "links": {
    "self": "http://[Server IP Address]/api/index.php/v1/
    extensions",
    "next": "http://[Server IP Address]/api/index.php/v1/
    extensions?page%5Boffset%5D=20&page%5
    Blimit%5D=20",
    "last": "http://[Server IP Address]/api/index.php/v1/
    extensions?page%5Boffset%5D=220&page%5
    Blimit%5D=20"
  },
  "data": [
    {
      "type": "manage",
      "id": "91",
      "attributes": {
        "name": "Authentication - Joomla",
        "type": "plugin",
        "folder": "authentication",
        "client_id": 0,
        "status": 2,
        "version": "3.0.0",
        "id": 91
      }
    }, ... ]
  }
}
    
```

Fig. 15. Response for Joomla

 TABLE XX
 Number of Candidate Keys for Joomla

Item	Value
The Number of String Candidate Keys	134
The Number of Numerical Candidate Keys	57

 TABLE XXI
 Part of Candidate Keys for Joomla

String Candidate Key	Numerical Candidate Key
site	0
featured	1
next	25079
page	25080
privacy	25081
category	

 TABLE XXII
 Vulnerability Assessment Results for Joomla

Endpoint	HTTP StatusCode
/api/index.php/v1/redirects	200
/api/index.php/v1/contacts	200
/api/index.php/v1/extensions	200

vAPI(API 5) was the lowest, the number of vulnerability queries was the highest. Although this number is considered tolerable during the development phase, there will be a need to devise an approach that does not generate obvious invalid vulnerability requests, such as reductions in *candidate keys*.

VIII. CONCLUSION

In this study, we proposed a vulnerability assessment method for addressing *API 1:2019*, *API 5:2019*, and *API 9:2019* vulnerabilities in the OWASP API Security Top 10.

TABLE XXIII

Vulnerability Assessment Results in terms of System Load

	vAPI1	vAPI5	WordPress
The number of endpoints	3	2	47
The number of candidate keys	23	6	116
The number of queries	2714	15586	5288
The number of http status 200	5	15554	184
The number of http status 500	2704	22	0
The number of invalid requests	5	10	5104

	Ghost	Joomla
The number of endpoints	29	47
The number of candidate keys	1861	191
The number of queries	2108	657
The number of http status 200	11	12
The number of http status 500	0	46
The number of invalid requests	2097	599

The proposed method conducts a dynamic vulnerability assessment using requests to validate the Web APIs according to API references and vulnerability requests that take advantage of their corresponding responses.

In this experiment, we confirmed the following:

- *API 1:2019* and *API 5:2019* vulnerabilities in the OWASP API Security Top 10 can be exactly detected using the vAPI.
- In WordPress, the known vulnerabilities of *API 1:2019* and *API 5:2019* (i.e., CVE-2017-1001000) can be exactly detected.
- In Ghost CMS, the known vulnerabilities of *API 5:2019* (i.e., CVE-2021-39192) can be exactly detected. Additionally, 10 endpoints are identified as vulnerabilities. Specifically, the */admin/session/* and */admin/integrations/* endpoints are not described in the API reference.
- In Joomla, the unknown vulnerabilities of *API 9:2019* can be detected. Additionally, 3 endpoints are identified as unpublished endpoints.

In addition, we evaluated the impact on each CMS in terms of system load. The number of assessment queries of the proposed method depends on the type of strings in the keys and values and the number of keys. Therefore, there is a need to develop an approach that does not generate obvious invalid vulnerability requests, such as a reduction in the number of *candidate keys*.

With the application of the proposed method, we anticipate an effective vulnerability assessment of Web APIs during the developmental phase and prior to production.

REFERENCES

- [1] Akamai, "State of the internet / report — api: The attack surface that connects us all — akamai," <https://www.akamai.com/site/en/documents/state-of-the-internet/soti-security-api-the-attack-surface-that-connects-us-all.pdf>, (Accessed on 07/10/2023).
- [2] Open Worldwide Application Security Project, "Owasp top 10 api security risks – 2019 - owasp api security top 10," <https://owasp.org/API-Security/editions/2019/en/0x11-t10/>, (Accessed on 07/10/2023).
- [3] imperva, "imperva/automatic-api-attack-tool: Imperva's customizable api attack tool takes an api specification as an input, generates and runs attacks that are based on it as an output," <https://github.com/imperva/automatic-api-attack-tool>, (Accessed on 07/13/2023).
- [4] VegaBird Technologies, "Vooki - web application and api vulnerability scanner — vooki infosec," <https://www.vegabird.com/vooki/>, (Accessed on 07/13/2023).

- [5] Open Worldwide Application Security Project, "OWASP ZAP," <https://www.zaproxy.org/>, (Accessed on 07/25/2023).
- [6] SwaggerHUB, "SwaggerHUB," <https://app.swaggerhub.com/search>, (Accessed on 07/25/2023).
- [7] Takai Masanari and Sakaguchi Tetsuo, "Automatic generation of program libraries for accessing web apis,," *IPSI Information Fundamentals and Access Technologies (IFAT)*, vol. 2012-IFAT-108, no. 1, pp. 1–8, 09 2012, (in Japanese).
- [8] Q-Success: World Wide Web Technology Surveys, "Market share yearly trends for content management systems, July 2023," https://w3techs.com/technologies/overview/content_management, (Accessed on 07/15/2023).
- [9] roottusk, "roottusk/vapi: vapi is vulnerable adversely programmed interface which is self-hostable api that mimics owasp api top 10 scenarios through exercises," <https://github.com/roottusk/vapi>, (Accessed on 07/16/2023).
- [10] WordPress Foundation, "Blog tool, publishing platform, and cms – wordpress.org," <https://wordpress.org/>, (Accessed on 07/16/2023).
- [11] Ghost Foundation, "Ghost: The creator economy platform," <https://ghost.org/>, (Accessed on 07/16/2023).
- [12] Open Source Matters, Inc., "Joomla content management system (cms) - try it! it's free!" <https://www.joomla.org/>, (Accessed on 07/16/2023).
- [13] Q-Success: World Wide Web Technology Surveys, "Usage statistics and market share of wordpress, July 2023," <https://w3techs.com/technologies/details/cm-wordpress>, (Accessed on 07/15/2023).
- [14] National Institute of Standards and Technology, "Nvd - cve-2017-1001000," <https://nvd.nist.gov/vuln/detail/CVE-2017-1001000>, (Accessed on 07/17/2023).
- [15] Q-Success: World Wide Web Technology Surveys, "Usage statistics and market share of ghost, July 2023," <https://w3techs.com/technologies/details/cm-ghost>, (Accessed on 07/15/2023).
- [16] National Institute of Standards and Technology, "Nvd - cve-2021-39192," <https://nvd.nist.gov/vuln/detail/CVE-2021-39192>, (Accessed on 07/17/2023).
- [17] Q-Success: World Wide Web Technology Surveys, "Usage statistics and market share of joomla, July 2023," <https://w3techs.com/technologies/details/cm-joomla>, (Accessed on 07/15/2023).



Yuki Ishida was born in Japan 1991, received the B.E. and M.S. degrees in Informatics from Tokyo University of Information Sciences, Japan, in 2014 and 2016, respectively. Upon graduation in 2016, he joined Digital Arts, Inc., where he contributed to the development of security products. In 2019, he transitioned to SecureBrain Corporation, also in Japan, with a primary focus on research and development in the field of cybersecurity. Since 2022, he has been concurrently enrolled in the doctoral program at the Graduate School of Informatics at

Tokyo University of Information Sciences in Japan. His research interests encompass cybersecurity and network quality control. He holds memberships in IEEE, IEICE, and IPSJ.



Masaki Hanada was born in Japan 1973, received the B.E. degree in resources engineering from Waseda University in 1996, the M.S. degree in information science from Japan Advanced Institute of Science and Technology (JAIST) in 1999, and the M.S. and D.S. degrees in global information and telecommunication studies from Waseda University in 2003 and 2007, respectively. He worked at Waseda University and Tokyo University of Science. After joining Tokyo University of Information Sciences as an Assistant Professor in 2011, he has

been a Professor in the Department of Information Systems, Tokyo University of Information Sciences, since 2019. His research interests include network QoS control, network resource control and management, and network security. He is a member of the IEEE, IEICE and IPSJ.



Atsushi Waseda was born in Japan 1977, received the B.E. degree in communication engineering from the University of Electro-Communications in 2000. He received his M.S. and Ph.D. in information science from Japan Advanced Institute of Science and Technology (JAIST) in 2002, and 2007, respectively. He worked at the National Institute of Information and Communications Technology and KDDI research inc. After joining Tokyo University of Information Sciences as an Assistant Professor in 2019. His research interests include information security, quantum security and privacy protection. He is a member of the IEICE and IPSJ.



Moo Wan Kim was born in Korea 1951, received B.E., M.E. and Ph.D degree in electronic engineering from Osaka University, Osaka, Japan in 1974, 1977 and 1980, respectively. He joined Fujitsu Lab. in 1980 and had been engaged in research and development on multimedia communication systems, Intelligent Network, ATM switching system and operating system. In 1998 he joined Motorola Japan and had been engaged in research and development on CDMA2000 system. In 2000 he joined Lucent Japan and had been engaged in research and development on W-CDMA system, IMS and Parlay. In 2005 he joined Tokyo University of Information Sciences and had been engaged in research on Ubiquitous Network. In 2022 he joined TA Tech. as a lecture.