NAUTILUS: Automated RESTful API Vulnerability Detection

Gelei Deng[1], Zhiyi Zhang[2], Yuekang Li[1†], Yi Liu[1], Tianwei Zhang[1],
Yang Liu[1], Guo Yu[3], and Dongjin Wang[4]

[1]Nanyang Technological University, [2]CodeSafe Team, Qi An Xin Group Corp,
[3]China Industrial Control Systems Cyber Emergency Response Team,
[4]Institute of Scientific and Technical Information, China Academy of Railway Sciences

{gdeng003, yi009}@e.ntu.edu.sg, *{yuekang.li, tianwei.zhang, yangliu}@ntu.edu.sg*,
zhangzhiyi@qianxin.com, yuguo@cics-cert.org.cn, wangdongjin@rails.cn

Abstract

RESTful APIs have become arguably the most prevalent endpoint for accessing web services. Blackbox vulnerability scanners are a popular choice for detecting vulnerabilities in web services automatically. Unfortunately, they suffer from a number of limitations in RESTful API testing. Particularly, existing tools cannot effectively obtain the relations between API operations, and they lack the awareness of the correct sequence of API operations during testing. These drawbacks hinder the tools from requesting the API operations properly to detect potential vulnerabilities.

To address this challenge, we propose NAUTILUS, which includes a novel specification annotation strategy to uncover RESTful API vulnerabilities. The annotations encode the proper operation relations and parameter generation strategies for the RESTful service, which assist NAUTILUS to generate meaningful operation sequences and thus uncover vulnerabilities that require the execution of multiple API operations in the correct sequence. We experimentally compare NAUTILUS with four state-of-art vulnerability scanners and RESTful API testing tools on six RESTful services. Evaluation results demonstrate that NAUTILUS can successfully detect an average of 141% more vulnerabilities, and cover 104% more API operations. We also apply NAUTILUS to nine real-world RESTful services, and detected 23 unique 0-day vulnerabilities with 12 CVE numbers, including one remote code execution vulnerability in Atlassian Confluence, and three high-risk vulnerabilities in Microsoft Azure, which can affect millions of users.

1 Introduction

Representational state transfer (REST) has become one of the most popular standards for web service interactions [33, 34]. It has been adopted by many well-known web service providers, such as Google [7], Microsoft [6], Wordpress [12], etc., to expose their digital services and assets via RESTful APIs. As RESTful APIs gain popularity, they become a common attack vector for the digital services and assets behind. According to a survey by Salt Security [13], 91% of the respondents experienced API security incidents in 2021. This survey also discloses that vulnerability is the most commonly encountered security issue. Thus, securing RESTful APIs is particularly important for service providers, and early detection of vulnerabilities is an important task to protect the web services.

Penetration testing is a popular technique adopted by many service providers to fulfill this task [10, 11]. This technique is also known as ethical hacking, which launches authorized simulated cyberattacks to find vulnerabilities in the service under test (SUT). Penetration testing can be performed manually or with automated tools. Compared with manual testing, using automated tools can not only save human effort but also yield stable testing results regardless of the experience and knowledge of the human tester. Currently, there are two widely used automated penetration testing tools for RESTful APIs, namely Open Web Application Security Project Zed Attack Proxy (ZAP) [14] and Web Application Attack and Audit Framework (w3af) [5]. To test a RESTful service, both ZAP and w3af utilize dictionaries of predefined attack payloads to request and check every single API of SUT. Although these two tools have successfully discovered many bugs in several RESTful services [28], they can only detect vulnerabilities that involve just one RESTful API operation. However, according to our empirical study of 609 vulnerabilities, 499(82%) of them require multiple RESTful API operations to trigger. This is consistent with the studies conducted by OWASP [3] and Rapid7 [4], both of which report that RESTful API vulnerabilities are fundamentally web application vulnerabilities that can be exploited through API endpoints in multiple steps. Therefore, a technique that can generate sequences of RESTful API operations for vulnerability detection is urgently needed.

Recently, several techniques [23, 32, 46] have been proposed to automatically generate sequences of RESTful API operations for bug detection. These techniques take standard

†Corresponding author.

API specifications, such as the OpenAPI [41] specification (OAS) as input. In particular, they learn the dependencies among the API operations to build correct API operation sequences. Although these testing solutions can generate meaningful API operation sequences to be consumed by the SUT, they are not suitable for vulnerability identification in RESTful APIs due to three reasons. ❶ The API operation sequences generated by existing techniques are not dedicated for detecting vulnerabilities. For penetration testing, we should concentrate on testing potentially vulnerable operations. ❷ The information extracted from the OAS documents is not enough to render diverse yet correct requests as test cases. Furthermore, OAS documents commonly contain syntax errors [32], which make the retrieved information less credible. ❸ Existing testing techniques lack the appropriate payloads for API requests to simulate attacks as well as the test oracle to check if an attack is successful or not. They only observe responses with 5xx HTTP status codes to detect bugs and are not aware of injection or authorization-related vulnerabilities. Due to these challenges, there exists a huge research gap regarding the automated detection of multi-API vulnerabilities for RESTful services.

To overcome the above limitations, we propose NAUTILUS[1], which leverages a novel design of annotations in OAS to carry out penetration tests for RESTful services. The annotation can be classified into two categories: (1) *operation annotations*: these annotations guide NAUTILUS to generate meaningful and logical operation sequences by describing the relations between API endpoints; (2) *parameter annotations*: these annotations document the proper strategy to generate concrete parameter values for each request. The annotations are designed to be both automatically processable and human-readable. Therefore, NAUTILUS can work fully automatically or involve humans in the loop. Based on the annotations, NAUTILUS uncovers vulnerabilities in the SUT with two testing stages. The first stage is *exploration*. NAUTILUS can successfully request as many API operations as possible by building proper API operation sequences and rendering them with appropriate parameter values. Specifically, it focuses on API operations with user-controllable parameters because they are more likely to contain injection vulnerabilities. By analyzing the responses from the service, NAUTILUS updates the annotations to fix the errors in the OAS and records the appropriate parameter value generation strategy. The second stage is *exploitation*. NAUTILUS constructs the operation sequence based on the exploitation pattern of the target vulnerability type, and mutates the injectable parameters with the payload dictionary. The SUT is then tested with the corresponding test oracles and reports the detected vulnerabilities.

We implemented NAUTILUS as a testing framework and evaluated it on six RESTful services. The experiment results show that NAUTILUS can outperform state-of-the-art vulnerability scanners [5, 14] and RESTful API testing tools [23, 32] with superior API operation coverage (36.9% - 174.6% increment) and numbers of detected vulnerabilities (85.8% - 202.8% increment). We further applied NAUTILUS to nine real-world RESTful API services, and detected 23 vulnerabilities. Specifically, we found three vulnerabilities in Microsoft Azure [36] and one vulnerability Atlassian Confluence [22], which can affect millions of users. Until now, all of them have been confirmed and fixed by the vendors, and ten of them have been assigned with CVE numbers.

To summarize, we make the following contributions:

- We conduct an empirical study to comprehensively analyze the patterns of RESTful API vulnerabilities, and present the key findings.
- We propose a novel design of OpenAPI specification annotations, which can benefit both automated and human-in-the-loop testing.
- We implement an automated testing tool – NAUTILUS, which can make use of the annotations to detect vulnerabilities in RESTful services.
- We compare the performance of NAUTILUS against four vulnerability scanners and RESTful API testing tools on six RESTful services and demonstrate that NAUTILUS can significantly outperform state-of-the-art techniques.
- We apply NAUTILUS to nine real-world web services, including famous commercial products, and identify 23 vulnerabilities with 12 assigned CVE IDs. We responsibly disclose the vulnerabilities to the vendors and all of the vulnerabilities are confirmed and fixed.

To facilitate future research, we will release the source code of NAUTILUS in accordance with our industrial collaborator on our website: https://sites.google.com/view/nautilus-testing.

2 Background

2.1 Key Concepts

RESTful API. The REpresentational State Transfer (REST) is a software architectural style proposed in 2000 [25] that defines the behaviors of an Internet-scale distributed hypermedia system, such as the Web. A Web API following the REST standard is called a RESTful API. Similarly, a web service that follows the REST standard is called a RESTful service. The REST architecture constrains the behavior of the system, and one of the most basic constraints is the *Uniform Interface*, which regulates users to access resources through regulated CRUD operations. Modern RESTful APIs often use the HTTP protocol as the transportation layer, and naturally the CRUD operations of RESTful APIs are mapped to the HTTP methods POST, GET, PUT, and DELETE, respectively. A RESTful service can contain many endpoints, each

[1] NAUTILUS is the name of a submarine in the science fiction – Twenty Thousand Leagues Under the Seas.

```yaml
/members/me:
  get:
    responses:
      '200':
        description: "Success"
  put:
    parameters:
      - in: header
        name: x-wp-nonce
        schema:
          type: string
        required: true
        description: "WordPress nonce"
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              context:
                type: string
                example: 'edit'
              name:
                type: string
              user_login:
                type: string
              email:
                type: string
                example:
                  'test@user.mail'
              password:
                type: string
              roles:
                type: string
                example: 'user'
    responses:
      '200':
        description: "Success"
/groups:
  post:
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              context:
                type: string
              groupname:
                type: string
              creater-id:
                type: integer
              group_description:
                type: string
    responses:
      '200':
        description: Success
/groups/{groupname}/admin/manage-members:
  get:
    parameters:
      - in: path
        name: groupname
        schema:
          type: string
    responses:
      '200':
        content:
          application/json:
            schema:
              type: object
              properties:
                data:
                  type: string
                x-wp-nonce:
                  type: string
```

Figure 1: The OpenAPI specification of BuddyPress APIs*

* For clarity, we omit some details in the YAML file.

of which is a digital location (typically with its own URL) to perform a series of pre-defined functions. These endpoints can be queried through different HTTP methods and body contents depending on the service's function, and each query is called an API operation.

OpenAPI Specification (OAS). OpenAPI (previously known as Swagger) defines a standard for describing RESTful APIs and the documentation that follows this standard is called OpenAPI specification [41]. The OpenAPI specification of target RESTful service contains the information of the object schemas as well as the API endpoints of a web service, including but not limited to the available CRUD operations, input parameters as well as expected responses. Each object has pre-defined fields and corresponding parameter types. Users can follow the specification to produce valid API operations and render them into HTTP requests to interact with the RESTful service endpoints.

Figure 1 shows a fragment of the OpenAPI specification for APIs in BuddyPress service [9], an extension to WordPress [8] blog management system. In this example, three API endpoints are specified and they are marked with grey background. We can see that each API endpoint supports one or more CRUD operations. In total, four operations are described in Figure 1 , showing their input parameters and responses. For an input parameter, it can be inside the request body (body of `put-/members/me`), in the HTTP request header (parameters of `put-/members/me`), or in the URLs of endpoints (parameter *groupname* in `get-/groups/{groupname}/admin/manage-members`). For a response, it contains the HTTP status code as well as the content body. In addition, some operation parameters and responses may involve objects that are described by schemas. For example, the admin management operation response contains an *application/json* format data with *data* field and *x-wp-nonce* field.

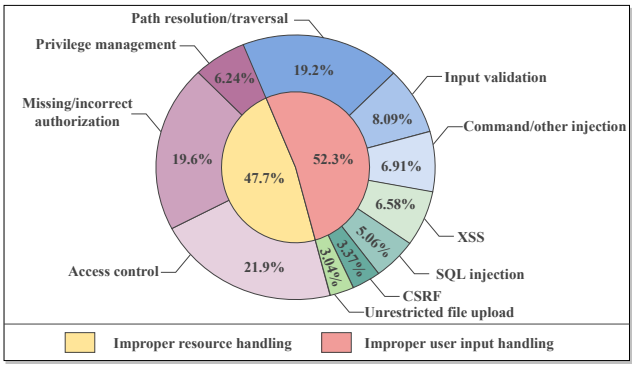


Figure 2: RESTful API vulnerability categories

2.2 RESTful API Vulnerabilities

While there are many RESTful API vulnerabilities in the wild, their exploitation patterns and root causes were not systematically summarized. Before investigating the methodology for RESTful API penetration testing, we conducted an empirical study to answer two research questions to limit the types of vulnerability for detection and understand the challenge of RESTful API vulnerability detection:

RQ1 (Scope) What are the categories of RESTful API vulnerabilities?

RQ2 (Challenge) What are the differences between triggering the RESTful API vulnerabilities and triggering the bugs/internal server errors?

In the empirical study, we collected a total number of 609 RESTful API vulnerabilities from the CVE list [37] of the National Vulnerability Database (NVD) and exploit-db [40]. We manually analyzed the vulnerabilities via the disclosed information such as the CVE descriptions, the exploits, the patches and so on.

Vulnerability Categorization RESTful API vulnerabilities can be categorized by many criteria. Here we focus on using their Common Weakness Enumeration (CWE) [38] types for categorization. Figure 2 shows the categorization result. From Figure 2, we can observe that there are two main root causes for RESTful vulnerabilities. ❶ 52.3% of vulnerabilities are caused by improper user input handling, which can be mapped to multiple CWE items including different types of injections (SQL injection, XSS, command injection, etc.). ❷ 47.7% of vulnerabilities are caused by improper resource management, including broken access control, lack of rate limits, sensitive information disclosure, etc. In this paper, we focus on detecting the vulnerabilities caused by improper user input handling due to their prevalence and significance. Specifically, they are the major types of vulnerabilities (52.3%), and lead to

exploitable scenarios, including command injection and code execution. In comparison, improper resource handling vulnerabilities are difficult to model uniformly because it is difficult to define *sensitive* data in different contexts. Therefore, we restrict our research scope to the former type of vulnerabilities and refer to them as *RESTful API vulnerabilities* unless stated otherwise and we discuss the identification of improper resource handling vulnerabilities as future work in Section 6.

Vulnerability vs. Bug Through the empirical study, we found that the detection of vulnerabilities differs from the detection of bugs in three aspects. ❶ *Attack Payload.* Each type of RESTful API vulnerability requires a corresponding type of payload for triggering. The exploit payloads are mainly injected into three positions of a RESTful request: body parameters, in-url parameters and cookies. For example, exploiting a SQL injection vulnerability (CVE-2019-10692) in the RESTful service requires a suffix of `--` to the original SQL query in the body of the request, which is a common payload pattern for SQL injection. ❷ *API Call Sequence.* For detecting a RESTful API bug, the only requirement for an API call sequence is that it can reach the buggy API properly. On the contrary, to detect RESTful API vulnerabilities, different types of vulnerabilities require different API request sequence patterns. According to our empirical study, there are two major types of patterns for API call sequences. For SQL injections, normally they only require one `GET` operation for both injecting and triggering. For other incorrect user input handling vulnerabilities, such as stored XSS vulnerabilities, they require one `POST`/`PUT` operation for injecting attack payloads followed by one `GET` operation to trigger. ❸ *Test Oracle.* RESTful API bugs and vulnerabilities requires different types of test oracles for capturing. For detecting RESTful API bugs, we only need to observe the status codes of the responses. A bug occurs when a response has a 5xx status code. For detecting RESTful API vulnerabilities, we need to use three types of manifestations: the change of status code before and after applying the attack payloads, the change of response data object structure before and after applying the attack payloads, and the semantic relation between the content of response bodies and the attack payloads.

3 Running Example

In BuddyPress version 7.2 and below, there is an injection-based privilege escalation vulnerability, which allows an attacker to escalate his/her user privilege to the administrator. This vulnerability has been recorded as CVE-2021-21389 [39]. Figure 1 shows the OpenAPI specifications for some of the APIs related to this vulnerability and Figure 3 shows the exploitation steps. We first introduce the mechanism of CVE-2021-21389 and then explain why existing bug detection and penetration testing techniques cannot reveal it.

According to Figure 3, CVE-2021-21389 requires three steps (including six API calls) to exploit. ❶ The attacker needs to signup and login properly. The login API will return a `nonce`, which is needed as an identity token to access the follow-up APIs. ❷ In order to launch the attack, the attacker first needs to send a `POST` request to the `/groups` API to create a new group. The attacker can then get `groupname` in the response which contains the data object of the newly created group. With `groupname`, the attacker can send a `GET` request to the `/groups/{groupname}/admin/manage-members` API to get the data objects of the administrators in the group, including a `x-wp-nonce`, which can be used as the identity token for group administrators. By adding the `x-wp-nonce` into the request headers, the attacker can send `PUT` requests to `/members/me` to change his/her personal information as an administrator. By appending an attack payload to the request that sets the `role` property to `administrator`, the attacker can escalate his/her privilege to the administrator. ❸ To verify successful privilege escalation, the attacker sends a `GET` request to `/members/me` and checks whether the data object in the response contains more properties than documented in the OAS.

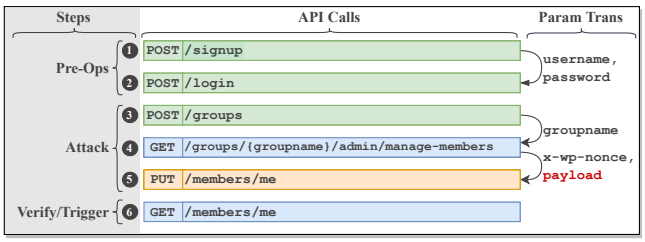


Figure 3: The running example (CVE-2021-21389)

This vulnerability cannot be detected by existing penetration testing tools such as w3af and ZAP. The reason is that they can only test one API of the SUT per time while the example vulnerability requires a sequence of six API calls to trigger and verify. Simply injecting the attack payloads via `PUT` operations through the `/members/me` API does not work due to the wrong value of `x-wp-nonce`.

This vulnerability cannot be detected by existing bug detection techniques such as RESTLER, RESTTESTGEN, and MOREST. The reason is double-fold. First, these techniques cannot add attack payloads to their requests. Second, even if attack payloads are added, these techniques lack the awareness of whether an attack is successful or not. They only capture responses with 5xx status code for bug detection while in this example, the vulnerability does not trigger any response with 5xx status code.

4 Design

4.1 Overview

Figure 4 shows the overview of NAUTILUS. We can see that the overall input is the OpenAPI specification of the SUT and

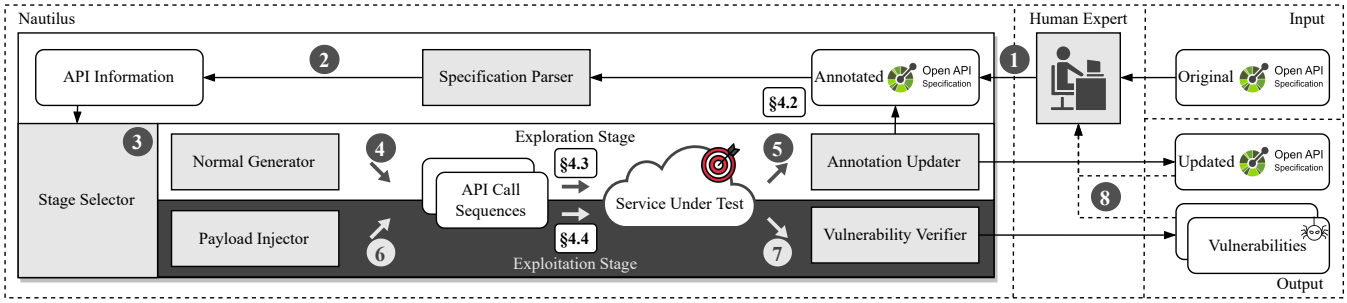


Figure 4: Overview of NAUTILUS

the overall outputs are the updated OpenAPI specification with customized annotations and the detected vulnerabilities. The workflow of NAUTILUS is as follows.

Annotation Processing. ❶ Given the original OpenAPI specification of the SUT, the human expert can ***optionally*** add some initial annotations to the specification following the specification annotation design of NAUTILUS (Section 4.2). The annotations are automatically processable and human-readable. Therefore, after NAUTILUS has generated some new annotations, the human expert can also choose to further update them manually. ❷ With the annotated specification, NAUTILUS leverages its specification parser to extract API information, including the relations among the APIs and the parameter details of each API.

Two-stage Testing. ❸ With the extracted API information, NAUTILUS generates API operation sequences to test the SUT. NAUTILUS runs testing in two stages, namely *exploration* and *exploitation*. NAUTILUS switches from the exploration stage to the exploitation stage when no endpoints are successfully requested after a predefined time threshold t [2]. Meanwhile NAUTILUS switches from the exploitation stage back to the exploration stage after the same amount of time t. ❹ Under the exploration mode, NAUTILUS aims to successfully request as many `POST`/`PUT` API endpoints as possible by generating proper API call sequences as test cases. The test case generation involves generating a correct sequence of API calls and filling up the API parameters properly (Section 4.3). ❺ In the exploration process, NAUTILUS leverages the execution feedback from the SUT to create new annotations or update existing ones, thus providing more accurate guidance for the testing. The updated annotations can be used for extracting new API information. ❻ During the exploitation stage, NAUTILUS applies the attack payloads to the successful API call sequences generated in the exploration stage to create new test cases for vulnerability detection (Section 4.4). ❼ In the exploitation stage, NAUTILUS captures the vulnerability by verifying the execution feedback of the SUT. Different types of attack payload require different verification oracles.

Result Handling. ❽ After NAUTILUS completes the testing, it can provide the updated/annotated OpenAPI specification together with the detected vulnerabilities to the human expert for further analyses, such as vulnerability clustering or specification fixing.

4.2 Specification Annotation

NAUTILUS uses a set of customized annotations to complement the information embedded in the OpenAPI specification. The design of the annotation is fully compatible with the OpenAPI 3.0 standard. Moreover, the annotations are human-readable and automatically processable. Hence, both human experts and NAUTILUS can create or update the annotations.

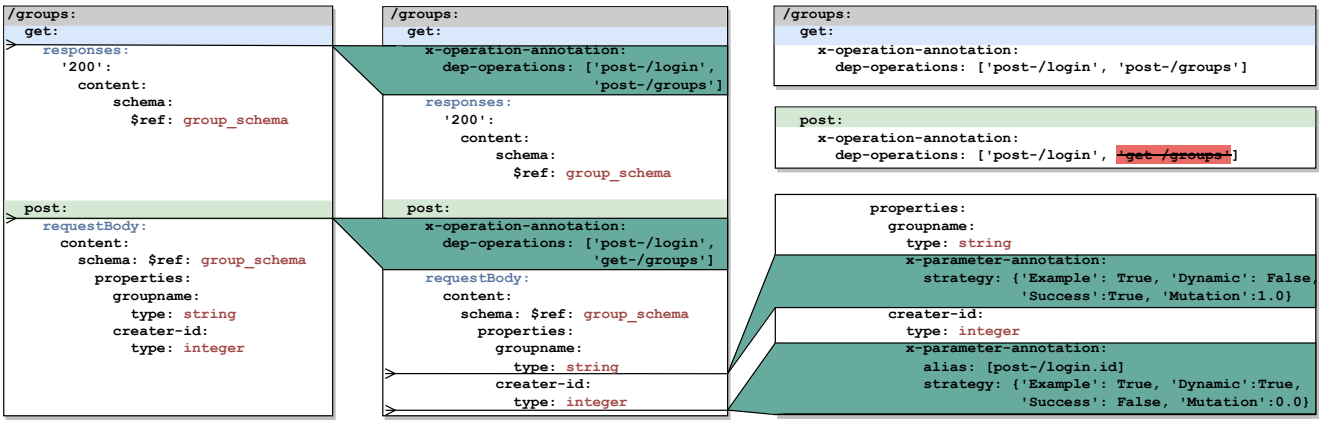
The purpose of the specification annotations is to embed more information in the OpenAPI specification. The reason is that although the OpenAPI specification can document the endpoints and parameters information, the information is not complete or accurate enough for NAUTILUS to generate valid test cases. To address this problem, we introduce two types of annotations: *operation annotation* and *parameter annotation*. The former helps NAUTILUS to construct meaningful API operation sequences, while the latter improves the effectiveness of parameter value generation.

4.2.1 Operation Annotation

Operation annotations are designed to guide the generation of valid API operation sequences that comply with the business logic of the service. For this purpose, we design the following three types of fields for the operation annotations:

Dep-operation. The *dep-operation* field annotates an operation with its dependent operations that should be executed in advance. This field is in the form of a ***list*** of API operations, where each operation is a uniquely identified string in the format `{request method}-{endpoint name}`. The dependencies among operations can be classified into three categories: parameter-wise data dependencies, CRUD dependencies and logical dependencies. ❶ The parameter-wise data dependency refers to the case where the variables in the response of one operation is used as the request parameters of the other operation. For example, in Figure 5 b), `get-/groups` is marked as a dependent operation for `post-/groups` because the response of the former matches the request body of the latter

[2]The default value of t is 30 minutes.



a) Original Specification b) Initial Annotated Specificaiton c) Final Annotated Specification

Figure 5: Annotation updates on the running example*

* For simplicity, we omit unnecessary fields in the specification.

by both referring to the group schema. ❷ The CRUD dependency is to enforce the CRUD restrictions and link up operations according to their CRUD relations. For example, in Figure 5 c), `post-/groups` is marked a dependent operation for `get-/groups` since following the CRUD relation, a group should be created first before it can be read. Notice that using parameter-wise data dependencies and CRUD dependencies may yield contradictory operation dependencies, like shown by the running example. Therefore, we have a dynamic update mechanism to resolve the conflicts (Section 4.2.3). ❸ The logical dependency refers to the dependency introduced by the internal logic of the SUT. For example, in both Figure 5 b) and c), `post-/login` is the dep-operation for both `get-/groups` and `post-/groups`. The reason is that `post-/login` can return a `nonce`, which is a required parameter in the header used as the identity token for accessing other API endpoints.

Term-operation. The *term-operation* field annotates the operations that terminate the current session with the SUT. Similar to dep-operation, this field is a ***list*** of API operations and the format of the API operations is the same. The operations stored in this field should only be executed after other operations. Endpoints like `logout` and `change_password` belong to this category. During testing, NAUTILUS never inserts the term-operations in the middle of operation sequences.

Alias. The *alias* field annotates the aliases of parameter names across the operations. This field is a ***list*** of strings, each of which is in the format of `{operation}.{parameter name}` and points to the parameter from the specific operation. This design aims to address the naming issue in OAS. In practice, we find that poorly maintained OAS have one parameter with different names in different operations. Since the aliases are across multiple operations, we put the alias field under operation annotations. This field is useful for rendering API requests as it helps to correctly match the parameter values across different operations in the same sequence (Section 4.3).

4.2.2 Parameter Annotation

Parameter annotations are designed to guide NAUTILUS to generate and link up parameter values of the API operations by addressing the drawbacks of existing solutions. Existing RESTful API testing techniques [23, 32, 46] generate parameter values via two basic strategies: *random* and *previous success*. The *random* strategy generates random values based on the type of data specified in the API specification (e.g., a random integer if the value type is `integer`). The *previous success* strategy generates the value of a parameter using the value of the last successful request. Both strategies have clear drawbacks. On the one hand, the *random* strategy is inefficient because requests with non-compliant parameter values will be rejected by the RESTful service without clear feedback, which does not provide sufficient guidance to the next parameter value generation. On the other hand, the *previous success* strategy does not bring enough diversity to the test cases, thus cannot explore the API service effectively. To address both randomness and correctness of the parameter values, we design the following four fields to describe the parameter generation strategies:

Example. This field is a ***boolean*** value. If the value is `True`, NAUTILUS will use the parameter values documented in the *example* field of the OAS. Normally, the parameter values provided by the examples are correct, which can help NAUTILUS to successfully request the corresponding endpoints.

Dynamic. This field is a ***boolean*** value. If the value is `True`, NAUTILUS will get the corresponding parameter values from previous successfully requested operations in the same sequence. The mechanism of deciding the parameter is from which previous operations is almost the same as RESTLER [23], where parameter names and schemas are used

Table 1: Parameter generation strategy (●: True ○: False)

Example	Dynamic	Success	Generation Strategy
●	●	●	Dynamic
●	●	○	Dynamic
●	○	●	Success + Mutation
●	○	○	Example + Mutation
○	●	●	Dynamic
○	●	○	Dynamic
○	○	●	Success + Mutation
○	○	○	Random Generation

to determine whether two parameters should be linked up. The only difference is that NAUTILUS also uses the alias information of the parameters.

Success. This field is a ***boolean*** field. If it is `True`, NAUTILUS will use the parameter values of the last successful request. Otherwise, NAUTILUS will try to generate new parameter values randomly.

Mutation. This field is a ***float number*** with a range of 0.0 to 1.0, which represents the parameter's mutation degree. The higher the value is, the service can intake more flexibly mutated parameter value. Specifically, we adopt the normalized edit distance or similarity (NES) from [35] to measure the mutation degrees. Given the original parameter string s with length l, we generates the mutation string s_m with length l_m while maintaining the NES between the two strings below boundary t. The NES can be calculated by $e^{\frac{d}{d-max(l,l_m)}}$, where d is the Levenshtein Distance [31] between the two strings. We also record the largest NES of the mutated parameter that is still properly handled by the target RESTful service.

The final parameter generation strategy is an interplay of the values of four fields. Table 1 shows the relation between the generation strategy and the field values. NAUTILUS generates the parameter value based on its annotation at runtime. In general, the dynamic field has the highest priority and the example field has the lowest priority. Note that some field value combinations in Table 1 may never appear in practice. For instance, the example field and the dynamic field should never both take the value of `True` for well-documented OASs.

4.2.3 Annotation Updates

Annotation Initialization and Manual Update. At the beginning of testing, NAUTILUS provides an utility to generate the initial annotations based on the parameter-wise dependencies and heuristics, such as using keyword matching to identify operations as *login* or *logout*. Figure 5 b) shows an example of the initially added annotations of NAUTILUS. The human expert can then choose to update the annotated OAS documentation based on his/her domain knowledge. As far as the manually added annotations follow the required formats, NAUTILUS can work with them seamlessly. Note that the human expert can also skip this manual update step to let NAUTILUS work fully automatically.

Dynamic Annotation Update. During testing, NAUTILUS updates the annotations dynamically according to the execution feedback of the SUT. For ***operation annotations***, NAUTILUS updates them in the following three scenarios. ❶ One of the most common cases is where the OAS does not correctly document the response body of an operation. In this case, NAUTILUS needs to analyse the actual response body after successfully requested an endpoint to fine-tune parameter-wise dependencies and update the corresponding dep-operation annotations. In the running example (Figure 1), the response body of `get-/members/me` is not documented. NAUTILUS will update the parameter-wise dependencies related to the response body of this operation once it receives the actual successful response during the execution. ❷ Another common case is updating the parameter aliases. In the running example, the `get-/members/me` operation returns current user object if successfully requested. The object has an `id` property, which is the ID of the current user. The `post-/groups` operation requires a parameter called `creator-id`. In BuddyPress, users can only use its own `id` to create new groups. Therefore, although `creator-id` and `id` have different names, they refer to the same value across the two operations. After some trail and errors during the testing, NAUTILUS can recognize that only when the values of `creator-id` and `id` are equal, the `post-/groups` can be requested successfully. Hence, NAUTILUS will mark them as aliases. ❸ The last case is about removing the infeasible dep-operations. If the operation after a dep-operation cannot be executed successfully after Θ tries (the default value of Θ is 10), the dep-operation will be removed from the annotation. For example, in Figure 5 b) and c), `get-/groups` will be removed from the dep-operation list of `post-/groups` during testing. The reason is that BuddyPress does not allow duplicate group names. So getting the information of an existing group and using the same information to create a new group will fail. For ***parameter annotations***, the updates are mainly about adjusting the value of the success field and the mutation field. If an operation has been successfully requested, its success field will be updated to `True`. As for the mutation field, the key idea is to increase the value of the mutation field upon successful requests and to decrease the value upon failed ones. The rationale is that if the request is successful, NAUTILUS can loose the restrictions to try out more aggressive mutations to increase the diversity of the parameters. However, if the request fails, NAUTILUS needs to apply mutations with smaller granularity to guarantee the correctness of the parameters.

4.2.4 Annotation Primitives

Our annotation primitives are constructed based on the specification extension feature of OpenAPI [2]. OpenAPI also allows users to add additional fields to pa-

rameters, namely *x-parameter*. With this feature, OpenAPI can be extended to support representations beyond RESTful services. For instance, Microsoft Azure APIs [36] contain custom fields including `x-ms-paths` and `x-www-form-urlencoded` to interact with internal services through encoded web form, which are not supported by conventional RESTful services. In NAUTILUS, the annotation primitives are denoted as `x-operation-annotation` and `x-parameter-annotation`. Since the annotation primitives are designed based on the official feature of OpenAPI, they are fully compatible with any OAS document.

4.3 Exploration Stage

In the exploration stage, NAUTILUS aims to successfully request as many API operations as possible with properly built API operation sequences. Algorithm 1 describes how NAUTILUS builds the API operation sequence for an API operation. As shown in Algorithm 1, NAUTILUS first checks if the API operation is *interesting* or not. Unlike existing RESTful testing techniques such as RESTLER, RESTTESTGEN and MOREST, NAUTILUS generates operation sequences by enumerating through only the interesting API operations instead of all operations. The interesting API operations refer to the operations that can possibly get attacked by requests with well-crafted payloads. Therefore, API operations which can accept user inputs and add/update backend data of the SUT are considered interesting. Specifically, NAUTILUS focuses on API operations with `POST`/`PUT` HTTP methods or API operations with the `GET` HTTP method and have in-URL parameters. For example, the `get-/members/me` operation in the running example (Figure 1) is not interesting because it does not accept user input and cannot be attacked. Thus, NAUTILUS will not try to build API call sequences for this operation. If an API operation is considered interesting, NAUTILUS will check the dep-operations of this operation and build the API call sequence by recursively including all the required API calls (line 6 – line 13 in Algorithm 1).

After the successful generation of API operation sequences, NAUTILUS renders the concrete HTTP requests one API operation after another. For each request, NAUTILUS generates its parameters according to the parameter annotations (Section 4.2.2) following the strategy in Table 1. The requests cannot be generated all at once because the parameters of some API calls are from the responses of previous API calls.

Finally, NAUTILUS will send the requests to the SUT to verify whether the target operation can be requested successfully and update the annotations according to the execution feedback (Section 4.2.3). Note that the target operation is considered successfully requested as long as it returns a response with 2xx status code.

The key difference of the exploration strategy between NAUTILUS and existing solutions [23, 32] is that NAUTILUS is aware of the most appropriate parameter values that can lead to successful requests. In particular, NAUTILUS identifies the best parameter generation strategy based on the execution feedback and records it in the parameter annotation, while prior solutions mainly rely on pre-defined strategies. After the exploration stage, the annotations are updated, further guiding the efficient vulnerability identification in the next stage.

Algorithm 1: API operation sequence generation

Input: a: A RESTful API operation
Output: $\mathcal{S}$: the API operation sequence based on a

```
1  def sequence_generation(a):
2      S ← [];
3      if is_interesting(a) then
4          concat_sequence(S, a);
5      return S;
6  def concat_sequence(S, a):
7      if a.dep_operations = ∅ then
8          return;
9      for d ∈ a.dep_operations do
10         if d ∈ S then
11             S.remove(d);
12         S ← [d, S];
13         concat_sequence(S, d);
14 def is_interesting(a):
15     return
         a.http_method ∈ {POST, PUT} ∨ (a.http_method =
         GET ∧ a.in_url_params ≠ ∅);
```

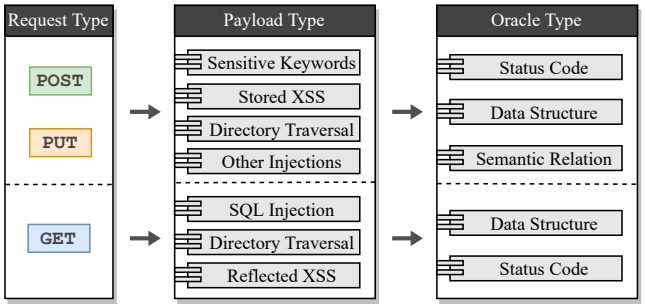


Figure 6: The mapping between the request types and the required payloads/oracles

4.4 Exploitation Stage

Once NAUTILUS cannot successfully request new endpoints for certain time, it switches into the exploitation stage. Or, if all endpoints are successfully requested, NAUTILUS will stay in the exploitation stage. In the exploitation stage, NAUTILUS aims to detect as many vulnerabilities as possible. The workflow of the exploitation stage is as follows. ❶ NAUTILUS collects the API operation sequences which can successfully request the interesting API operations. These API operation sequences will be used as the basis for vulnerability detection. ❷ When rendering the requests for an API operation sequence, for the dependent API operations of the interesting

API, NAUTILUS leverages the success parameter annotations to reuse the parameter values in the last successful requests. In contrast, for the interesting operations, NAUTILUS will use predefined attack payloads to replace its parameters. As elaborated in Section 2, RESTful services can contain various categories of vulnerabilities, each of which can only be revealed with certain type of payloads. Thus, we propose a vulnerability-specific mutation strategy to uncover certain types of vulnerability in the target RESTful service. The mapping between the most common types of the interesting API and the types of attack payloads are summarized in Figure 6. Below we elaborate the key technical steps involved.

4.4.1 Payload-based Mutation

Our general strategy is to mutate the normal request parameters with the vulnerability-specific payloads. Instead of arbitrarily injecting payloads into the parameters, NAUTILUS focuses on operations that have a greater possibility of containing vulnerabilities and construct the sequence accordingly in the following steps. ❶ NAUTILUS identifies the user-controllable parameters that are possible to get injected. The user-controllable parameters are the parameters whose dynamic fields have the value of `False`, which means that the value is not inherited from previous responses. ❷ NAUTILUS then identifies the operations that contain the injectable parameters as the candidate operations to test. It selects one operation from the candidate operations and picks one parameter from it as the mutation target. ❸ If the target operation is `GET`, NAUTILUS will reuse the corresponding API operation sequence generated in the exploration stage. When rendering requests, the injectable parameters, typically in-url parameters, are mutated using the payload dictionary. Specifically, the target parameter is replaced by the payload value to formulate the final request. The other parameter values are generated on the basis of the annotated strategy normally. ❹ If the target operation is `POST`/`PUT`, NAUTILUS will randomly pick a `GET` operation which has CRUD relation with the target operation and append it to the API operation sequence. The rationale of adding the `GET` operation is to obtain more information from SUT to serve as test oracles.

4.4.2 Payload and Oracle

The types of test oracles used for detecting vulnerabilities are related to the types of payloads and the relations are illustrated in Figure 6.

Status Code. Some vulnerabilities can cause changes in the status codes returned by the SUT. For instance, a successful login bypass results in 200 status code, while normally the server shall return 400 if the wrong credentials are provided.

Data Structure. Some vulnerabilities can cause the operation to return data objects with different data structures from what is described in the OAS. For example, SQL injection changes the response data structure because unexpected table contents are returned after successful exploitation.

Semantic Relation. Some vulnerabilities may falsely execute the payload content and add semantic relations between the contents of the payloads and responses. For instance, some command injection vulnerabilities can cause the parameters to be executed instead of being parsed as strings, and the execution result is predictable (e.g., the payload is '1+1' and the response is '2').

5 Implementation and Evaluation

We implement NAUTILUS based on Python 3.9.0 with 6,500 lines of code and conduct experiments to evaluate the performance of NAUTILUS. Our evaluation targets the following questions:

RQ1 (Vulnerability Detection) How is the vulnerability identification capability of NAUTILUS?

RQ2 (Coverage) How is the operation exploration capability of NAUTILUS?

RQ3 (Ablation Study) How do the annotation strategies affect the performance of NAUTILUS separately?

RQ4 (Real-world Targets) Can NAUTILUS identify vulnerabilities in real-world applications, including those industrial products?

5.1 Evaluation Setup

Evaluation Baselines. We compare our solution with both open-source vulnerability scanners and existing research works on RESTful API testing. It is worth noting that these tools are designed for different purposes. Vulnerability scanners are designed to assess web applications and APIs. Given requests to API endpoints, they send mutated requests to those endpoints and report the potential vulnerabilities directly. Conversely, RESTful API test tools aim to achieve better coverage and bug reporting, and they do not have vulnerability reporting capability. To conduct a fair comparison and evaluation, we select vulnerability scanners and RESTful API testing tools that are extensible to have custom payloads so that we can use the same payload data on all tools. In the end, we select the following four tools.

(1) **Zed Attack Proxy** (ZAP) [14] is an open-source blackbox web vulnerability fuzzer developed by OWASP. It is mainly used for blackbox vulnerability assessment and penetration testing. In this evaluation, we use ZAP's OpenAPI add-on and disable unrelated web exploration functions such as web crawling modules.

(2) **w3af** [5] is an open-source web application attack and audit framework. Similar to the previous setting, we use the w3af in-built module *crawl.open_api* and disable unrelated functions.

(3) **RESTLER** [23] is an open-source blackbox RESTful API testing technique developed by Microsoft. It dynam-

Table 2: Benchmark Applications

Subjects	LoC	Endpoints	Description	Version	Developer
NodeGoat [42]	24933	20	Educational	1.4	OWASP
Juice Shop [17]	109244	50	Educational	12.5	OWASP
VAmPI [24]	2695	13	Educational	-	Community
SeoPanel [18]	62277	20	SEO Management	4.0	Independent
Navigate [16]	57192[3]	12	CMS	1.8	Independent
Gila [15]	49391	24	CMS	2.1.0	Independent

ically builds operation sequences by appending new API operations according to execution feedback.

(4) **MOREST** [32] is a state-of-art blackbox RESTful API testing technique which constructs operation sequences through the dynamically updated RESTful-service Property Graph (RPG).

We modify the above tools to use the same FUZZDB payload dictionary [26]. It covers various types of vulnerabilities and is adopted by various testing tools and industrial solutions [14, 27, 30]. For ZAP and w3af, we update the payload dictionary to their corresponding modules. For RESTLER and MOREST, we extend their mutation modules so their value generation strategy use the payload from dictionary instead of random value generation. Note that we do not make changes on their test sequence generation strategy.

Evaluation Benchmarks. We select real-world web applications as our evaluation benchmarks using three criteria: (1) open-source for exploitation reproducibility, (2) actively maintained to validate security findings, (3) complete OAS or RESTful documentation available as required input for all baselines. The result is a selection of six web applications, as presented in Table 2. Three of the benchmark applications (OWASP NodeGoat [42], OWASP JuiceShop [17] and VAmPI [24]) are deliberately vulnerable applications with seeded vulnerabilities for education purposes. The other three applications are open-source software with both web interface and well-documented RESTful API endpoints. All of these applications are implemented based on their default documentation, and their details are demonstrated in the following Table 2.

Benchmark OpenAPI Specifications. We use the same OASs for all the evaluation baseline solutions to test the evaluation benchmarks. For NAUTILUS, we do not include additional manual annotations, and only use an automatic script to generate the initial operation annotations through keyword mapping on operation names (login, logout, checkout, etc.). The automation script is open-sourced on our project website [1].

Evaluation Criteria. We use two criteria for the evaluation of NAUTILUS and the baselines to answer the aforementioned research questions.

(1) **Vulnerabilities**: The number of vulnerability is crucial criteria for security testing. Without loss of generality, we focus SQL injection, XSS and improper access control, while our solution can also identify other injection-based vulnerabilities with proper payloads. In Section 5.5, we present industrial examples to demonstrate other types of vulnerabilities identified by our tools.

(2) **Operation Coverage**: Operation coverage directly reflects the successful exploration of RESTful API services. In the experiments, we use the successfully requested operations (SROs) as a criterion, because it reflects whether a technique can generate valid and complex requests to cover the deeper code logic in RESTful services.

Evaluation Settings. We setup all the tools and benchmarks based on their default installation settings. For the benchmark RESTful services, we host them on a local machine and run each technique with 12 hours. After each round, we tear down the benchmark and restore the environment (e.g., docker containers, self-hosted virtual machines) to ensure the consistency of RESTful services between tests. In addition, we repeat all experiments for 5 times to mitigate randomness and adopt Mann-Whitney U test (with the confidence threshold $\alpha = 0.05$) and $\hat{A}_{12}$ [45] calculation for statistic tests. So in total, our experiment records of 1,800, i.e., 6 projects * 5 settings * 12 hours * 5 repetitions, CPU hours of testing. We summarize our findings as follows.

Result Collection. After NAUTILUS and the evaluation baseline solutions report the vulnerabilities, we collect the result and manually conduct the exploitation to confirm the vulnerabilities. The false positives are eliminated, and we discuss their causes in Section 6.

5.2 Vulnerability Detection (RQ1)

The number of unique vulnerabilities identified by different solutions are presented in Figure 7. It can be noticed that all the baseline solutions have similar performances in vulnerability testing. This is because they use the same payload dictionary and follow a similar testing strategy to identify vulnerabilities at each endpoint individually. Particularly, ZAP and w3af have better performance compared to the two RESTful API testing solutions, because they have in-built XSS execution detection modules to identify such vulnerability.

In contrast, NAUTILUS achieves significantly better performance in both vulnerability types and number of vulnerabilities. NAUTILUS is able to uncover different types of vulnerability, including SQL injection, command injection, XSS and privilege management. On average, it identifies more vulnerabilities (69.8%) compared to the other solutions on the three educational benchmarks. Particularly, NAUTILUS can cover all vulnerabilities identified by these solutions. Meanwhile, NAUTILUS identifies 10 0-day vulnerabilities on the three real-world applications while the baseline solutions only identify 3 in total. These 10 vulnerabilities identified from SeoPanel, Navigate CMS and Gila CMS have been confirmed by the vendors. Their details are included in our website [1].

In-depth Analysis. To further explore the accuracy and effectiveness of NAUTILUS, we perform an in-depth analysis about

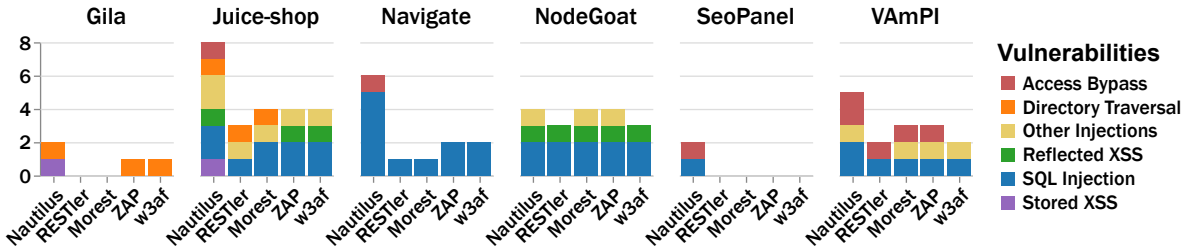


Figure 7: The vulnerabilities and their types uncovered by different tools on the evaluation benchmarks.

Table 3: Selected RESTful API vulnerabilities identified by NAUTILUS. For `Vulnerability Identification Tools`: ✓: this tool can identify the vulnerability, ✗: this tool cannot identify the vulnerability, -: this tool is not designed to uncover the type of vulnerability. For `Manual Annotation`: ✓: the annotations are updated by human experts to provide guidance on operation and parameter generation, ✗: no manual inputs into the specification annotations.

Target Application	Version	Vendor Confirmation	CVE/Issue-ID	Vulnerability Type	Multi-API	Vulnerability Identification Tools					Manual Annotations
						NAUTILUS	RESTLER	MOREST	ZAP	w3af	
RPCMS	1.8	✓	CVE-2021-37377	SQL Injection	✗	✓	✗	✗	✓	✓	✗
	1.8	✓	CVE-2021-37476	SQL Injection	✗	✓	✓	✓	✓	✓	✗
	1.8	✓	CVE-2021-37475	SQL Injection	✗	✓	✓	✓	✓	✓	✗
	1.8	✓	CVE-2021-37474	SQL Injection	✓	✓	✗	✗	✗	✗	✓
	1.8	✓	CVE-2021-37473	SQL Injection	✓	✓	✗	✗	✗	✗	✓
	1.8	✓	CVE-2021-37394	Privilege Escalation	✓	✓	✗	✗	✗	✗	✓
Azure	-	✓	CVE-2022-33659	Privilege Escalation	✓	✓	✗	✗	✗	✗	✓
	-	✓	CVE-2022-30181	Privilege Escalation	✓	✓	✗	✗	✗	✗	✓
	-	✓	CVE-2022-33657	Privilege Escalation	✓	✓	✗	✗	✗	✗	✓
Confluence	7.13.0	✓	Internally Issued	OGNL Injection	✓	✓	-	-	-	-	✓
Navigate	2.9.4	✓	Issue r1561-#26	SQL Injection	✗	✓	✓	✓	✓	✓	✗
	2.9.5	✓	Issue r1561-#27	Privilege Escalation	✓	✓	✗	✗	✗	✗	✗
Rukovoditel	2.8.3	✓	CVE-2021-30224	CSRF	✗	✓	-	-	✓	✓	✗
SeoPanel	4.0	✓	Issue #219	SQL Injection	✓	✓	✗	✗	✗	✗	✗
GilaCMS	2.1.0	✓	CVE-2021-34113	Directory Traversal	✓	✓	✗	✗	✗	✗	✗
	2.1.1	✓	CVE-2021-34115	Stored XSS	✓	✓	-	-	✗	✗	✗

false positives and false negatives. In particular, a false positive is the case where NAUTILUS reports a non-exploitable vulnerability. As it is hard to define true negatives in the domain of vulnerability detection, we define the false positive rate as $FP/(FP+TP)$. The detailed experimental results are presented in Table 4. In summary, NAUTILUS achieves a false positive rate of 24.74% on benchmark services, which is comparable to other solutions. This is acceptable as a human expert can easily follow the test results to identify valid vulnerabilities. We highlight that false positives are largely dependent on the quality of the payload dictionary and the target service because they are mainly contributed by (1) the non-compliant RESTful endpoint implementation, where the response status code or body content does not fulfill the RESTful standards and triggers the test oracle, and (2) unexpected service behaviors, where the payload triggers a buggy implementation in the service, causing service crashes yet not exploitable. These false positives can be mitigated by modifying the OpenAPI documentation to describe the actual behavior of the service.

We further study the false negatives of NAUTILUS. Due to the inherent difficulties in identifying all vulnerabilities in real-world services, we extend the experiment to explore if NAUTILUS can uncover known vulnerabilities. To do this, we collect 50 reproducible CVEs from 12 different open-source applications containing 25 different components/plugins, and their details are listed in Table 6 of the Appendix. The known vulnerabilities cover all subtypes of injection vulnerabilities illustrated in Section 2. We study the false negatives of these solutions by verifying the number of vulnerabilities that can be reported by each solution. The experimental result shows that NAUTILUS detects 70.0% (35/50) of the known vulnerabilities in the benchmark application, surpassing other solutions that achieve an average detection rate of 39.5%. Such improved performance can be attributed to Nautilus's superior endpoint coverage capability, which allows the detection of vulnerabilities at endpoints not covered by traditional solutions. NAUTILUS fails to uncover some vulnerabilities due to two main reasons: (1)some vulnerability can only be triggered by case-specific payloads, which are not included in the payload dictionary; (2) the service does not fulfill the RESTful standards and the oracle cannot determine if the vulnerability is triggered. We present the detailed analysis of each selected CVE on our website [1].

5.3 Coverage (RQ2)

We present the operation coverage results of different tools in Table 5. NAUTILUS achieves competitive performance in successfully requested operations compared to baseline solutions. Specifically, our solution achieves 163.1% more endpoint coverage on the benchmarks compared to the tradi-

Table 4: False positives / true positives identified by different tools on the evaluation benchmarks.

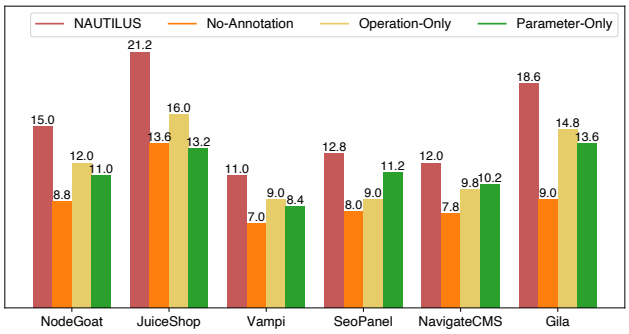
	NAUTILUS	RESTLER	MOREST	w3af	ZAP
Juiceshop	8/7	3/3	4/3	4/2	4/3
Vampi	6/1	2/0	3/0	3/0	2/0
NodeGoat	4/0	3/0	4/0	4/0	3/0
SeoPanel	2/0	0/0	0/0	0/0	0/0
Navigate	5/3	2/1	3/1	3/1	2/1
Gila	2/2	0/2	0/2	1/2	1/2
FP Rate	24.74%	30.56%	27.98%	20.83%	23.81%

tional web vulnerability identification tools. This is because traditional solutions can barely generate valid `POST` or `PUT` requests to interact with the API, thus not efficient in endpoint discovery. Compared to the RESTful API testing tools RESTLER and MOREST, our endpoint coverage increased by 54.8% and 36.9% on average. While the RESTful API testing solutions achieve better performance, they cannot generate specific test sequences to cover the corner cases. For instance, they fail to understand the login-logout logic in all the benchmark applications, thus cannot perform testing with different user accesses. Also, some endpoints have extreme restrictions on input value formats, and it is difficult to generate them by random. Our annotation strategy assists the solution to overcome these drawbacks.

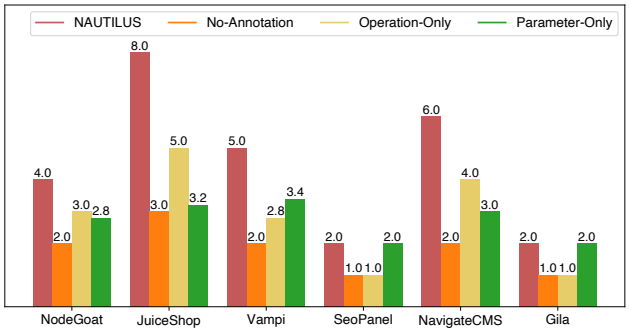
5.4 Ablation Study (RQ3)

To investigate how the annotation strategy contributes to improving the performance of NAUTILUS through guided testing and parameter generation, we perform an ablation study on the two main components of the annotation strategy: operation annotation and parameter annotation. To study their contributions, we implement three variants of NAUTILUS: (1) NAUTILUS-NO-ANNOTATION that disables all annotations, (2) NAUTILUS-OPERATION-ONLY that removes the annotations on parameters, and (3) NAUTILUS-PARAMETER-ONLY that removes the annotations on operations. The results are averaged using five runs, each 8 hours, to avoid statistical bias. Note that as in previous studies, we do not implement additional manual annotations to assist with the algorithms.

The result of the ablation study is presented in Figure 8. In general, we observe that NAUTILUS outperforms the other three ablation baselines in both vulnerability identification and endpoint coverage. Specifically, we have the following findings: (1) Without any annotation, the NAUTILUS-NO-ANNOTATION strategy achieves slightly lower performance in both vulnerability identification and endpoint coverage compared to RESTLER and MOREST. This is because our exploration strategy focuses on discovering the sequences that may contain vulnerabilities, thus may neglect potential relations between endpoints. Also, we can expect a significant performance drop caused by disabling parameter annotations. This is because that parameter annotations brings awareness



(a) Normalized average code coverage



(b) Average unique vulnerability detection

Figure 8: The performance of NAUTILUS, NAUTILUS-NO-ANNOTATION, NAUTILUS-OPERATION-ONLY, and NAUTILUS-PARAMETER-ONLY on both normalized average code coverage (μLOC) and bug detection.

to link the different-name parameters together, which overcomes the syntax errors in API specifications. On the contrary, both RESTLER and MOREST have mechanisms to compensate the potential errors in the specification. (2) Operation annotation contributes more when the service contains complex sequence logics (e.g., Juice-shop example), while parameter annotations contribute more when the service has strict parameter value format restrictions (e.g., SeoPanel) example. This is reasonable, as the two parameters are designed to address the sequence generation and parameter generation challenges, accordingly. (3) All of our baselines for the ablation study achieve competitive vulnerability identification results against benchmark solutions. We believe this is because we consider multi-API vulnerabilities in our tools. NAUTILUS can effectively uncover multi-API vulnerabilities as long as the operation sequences can be correctly executed.

We further investigate the annotations generated by NAUTILUS, and the details are presented in Figure 9. The maximum sequence length generated by Nautilus for the six services is 5.3 on average, and each service contains 15.3 automatically generated annotations (0.66 annotations per endpoint). By linking the annotations to the identified vulnerabilities, we find that 67% of the vulnerabilities can only be identified with annotations. This shows the effectiveness of

Table 5: Performance of NAUTILUS against RESTLER, ZAP and w3af in terms of both the average endpoint coverage and detected vulnerabilities (DV). We run this experiment 5 times (24 hours each time) and highlight statistically significant results in bold (We calculate the average increased number by $\frac{(\#\ of\ \text{NAUTILUS}) - (\#\ of\ baseline)}{\#\ of\ baseline}$.).

Subjects	Average Endpoint Coverage (EC)									Average # of Detected Vulnerabilities (DV)								
	NAUTILUS	RESTLER		MOREST		ZAP		w3af		NAUTILUS	RESTLER		MOREST		ZAP		w3af	
	μEC	μEC	$\hat{A}_{12}$	μEC	$\hat{A}_{12}$	μEC	$\hat{A}_{12}$	μEC	$\hat{A}_{12}$	μDV	μDV	$\hat{A}_{12}$	μDV	$\hat{A}_{12}$	μDV	$\hat{A}_{12}$	μDV	$\hat{A}_{12}$
NodeGoat	**15.00**	10.40	1.00	11.80	1.00	6.00	1.00	7.20	1.00	4.00	2.60	1.00	4.00	0.50	4.00	0.50	3.00	1.00
Juice-shop	**21.20**	14.00	1.00	16.20	1.00	8.00	1.00	7.00	1.00	**7.00**	3.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00
Vampi	**11.00**	7.60	1.00	7.60	1.00	4.00	1.00	4.00	1.00	**5.00**	2.00	1.00	3.00	1.00	3.00	1.00	2.00	1.00
SeoPanel	**12.80**	9.00	1.00	10.20	1.00	6.00	1.00	3.80	1.00	**2.00**	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
Navigate	**12.00**	8.00	1.00	9.40	1.00	4.00	1.00	4.00	1.00	**6.00**	1.00	1.00	1.00	1.00	2.00	1.00	1.00	1.00
Gila	**18.60**	9.80	1.00	11.00	1.00	8.00	1.00	7.00	1.00	**2.00**	0.00	1.00	0.00	1.00	1.00	1.00	0.00	1.00
Average Increased (%)	0.00	54.08	-	36.90	-	151.67	-	174.55	-	0.00	202.80	-	116.50	-	85.84	-	159.28	-

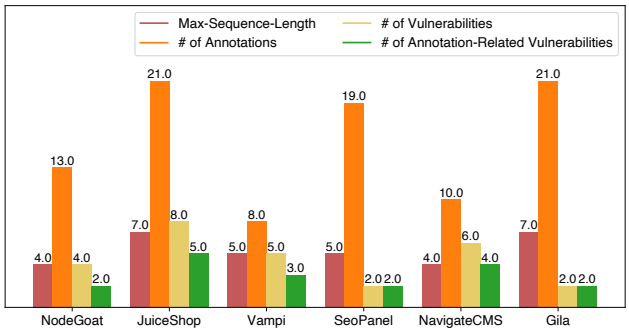


Figure 9: The maximum sequence length, number of auto-generated annotations, number of vulnerabilities, and number of annotation-related vulnerabilities of NAUTILUS.

our annotation-based strategy.

5.5 Real-world Vulnerabilities (RQ4)

5.5.1 Vulnerability Identification

We further apply NAUTILUS to test real-world RESTful applications and try to spot vulnerabilities. In the end, we successfully identify 23 unique vulnerabilities in various applications, and 21 of them have been confirmed by vendors. These vulnerable applications include both open-source ones like SeoPanel and Navigate CMS as mentioned in previous evaluations, and commercialized products/services provided by vendors like Microsoft and Atlassian. We have submitted these vulnerabilities to MITRE and have received 10 CVE numbers to the date of submission. We list selected vulnerabilities in Table 3 for reference, while the complete list with detailed description is available on our website [1].

Annotation Efforts. We manually annotate the OASs of these applications to provide sample parameter values and operation business logics to guide the testing. This process does not incur significant additional effort for testers with prior knowledge of the service, since they only need to provide the operation dependency and parameter value information based on normal queries to the endpoints. Empirically, our testers need an average of 1 minute to annotate one endpoint. We suggest that the annotation process can be integrated into the OAS construction, which is usually completed by service developers before service launch.

Added Values of Manual Annotation. Manual annotation enhances Nautilus in understanding the service logics that can hardly be learned heuristically from the execution feedback. As shown in Table 3 in Appendix, 7 of 23 0-day vulnerabilities require manual annotations to be detected, which provide the parameter generation guidance for fields that rely on external resources. This is common in complex systems, such as cloud services, as users need to acquire parameter values from other interfaces (CLI, etc.). This information is critical to the successful endpoint query and is a prerequisite for vulnerability identification.

In the following of this section, we present two case studies to demonstrate how NAUTILUS uncovers multi-API vulnerabilities in real-world applications.

Case 1: Gila CMS Stored XSS Gila CMS [15] is a content management system that provides both open source solutions and online hosting services. During the testing of Gila CMS v2.1.0, we identify a stored XSS vulnerability with multi-API vulnerability exploitation pattern. In particular, (1) a regular user could login and upload blogs through the `fm/upload` endpoint, providing *filename* and other blog contents. The user could maliciously select a filename that contains common stored XSS payloads, such as '<alert(1)>'. (2) The user receives response from the server that contains the server-generated *id* of the blog. He or she could access the blog content by providing *id*, and the stored XSS is exploited. This is a typical multi-API vulnerability, where the malicious user uses the `POST` method to inject the payload and trigger the payload through the `GET` request by providing parameters obtained from the previous operation.

Case 2: Atlassian Confluence OGNL Injection Atlassian Confluence [22] is one of the most popular team collaboration management tools developed and maintained by Atlassian [21], with millions of active users. In the testing of the Confluence RESTful API, we discover an Object-Graph Navigation Language (OGNL) injection vulnerability, which is specific to Java-based applications. The adversary could exploit the vulnerability and perform arbitrary remote code execution (RCE) in three steps. (1) A user with no access

can register an account, activate the RESTful service and obtain a valid API key. (2) He or she then requests the `doEditDailyBackupSettings` endpoint with the required parameters as declared in the documentation through the `POST` method. Specifically, the *dailyBackupFilePrefix* parameter can be injected by any OGNL injection, such as payload '{222 * 3}'. If the vulnerability exists, the payload is expected to be executed as a mathematical calculation instead of string storage. (3) After than, the user accesses the same endpoint through the `GET` method, and observes that the value of the injected parameter contains '666'. It is worth highlighting that the math operation payload is a proof-of-concept. The adversary could exploit the vulnerability to execute arbitrary code, such as downloading remote resources through the payload with 'wget resource-url'.

The aforementioned two cases demonstrate how NAUTILUS uncovers multi-API vulnerabilities that cannot be identified by traditional RESTful API testing solutions. They also show that NAUTILUS can be integrated with arbitrary payloads to uncover different types of vulnerabilities in real-world applications.

6 Discussion and Future Work

Bug/Vulnerability Analysis. Different from traditional RESTful API testing solutions that leverage HTTP 500 error response code to identify bugs, NAUTILUS does not particularly focus on this. In practice, RESTful API bugs can often be extended as exploitations, especially DoS attacks because these bugs are usually caused by data mis-handling at server backend. However, our goal is to automatically uncover RESTful vulnerabilities, and we do not tackle the manual process to analyze bugs and develop exploitation. How to effectively construct exploitation based RESTful API bugs is an interesting future work that we will work towards.

Improper Resource Handling Vulnerabilities. As another type of typical RESTful API vulnerabilities, improper resource handling vulnerabilities are difficult to be identified automatically. The test oracles for checking improper resource handling vulnerabilities are hard to be uniformly modeled, as it is difficult to classify the *sensitivity* of the data in API responses. The reason is that different systems and services can have different rules/regulations for access control, and the classification criteria for sensitivity are different. It is difficult to define whether the content should be accessible to users, especially when they have different access levels. Considering these two points, we focus on injection vulnerabilities. Identifying other RESTful vulnerabilities is an interesting and challenging topic, and we consider them as the future work.

7 Related Work

Instead of discussing all related works, we focus on the RESTful service testing techniques, penetration testing techniques and human-in-the-loop testing.

RESTful Service Testing Techniques. Several blackbox techniques were proposed to generate operation sequences for RESTful service testing. RestTestGen [46] builds Operation Dependency Graphs (ODGs) to model RESTful services and crafts operation sequences via top-down graph traversal. RESTLER [23] builds operation sequences with a bottom-up approach, which starts with single operation call sequences and extends the call sequences by appending more operations after trial and error. MOREST builds operation sequences based on dynamically updated RESTful-service Property Graph (RPG), which supports both top-down construction and bottom-up updates. Different from these techniques, NAUTILUS focuses on interesting operations that may contain vulnerabilities and construct sequences accordingly to obtain necessary parameter values. This strategy benefits NAUTILUS to efficiently test potentially vulnerable endpoints.

Whitebox testing techniques are also proposed for bug detection in RESTful services and general web services [47]. EvoMaster [19] is such a solution that leverages instrumentation to collect execution feedback during testing, and guides the evolutionary-algorithm -based test case generation. While EvoMaster is more effective in testing deeper logic inside the RESTful services, it is limited by the testing environments because it can only instrument Java/Scala/Kotlin-based services and requires access to the database.

Penetration Testing Techniques. With the development of fuzzing techniques, tools with automatic attack generation capabilities are developed for different types of vulnerabilities (sqlmap [29] for (No)SQL injection, XSStrike [43] for XSS, etc.). However, the vulnerability detection through penetration testing has still been largely a manual work, because these tools can only be adopted in restricted testing environments, and they do not have automatic exploitation generation capability. NAUTILUS addresses the vulnerability detection problem in the context of RESTful services, and its output reveals the API operation sequences of the exploitation.

Human-in-the-loop Testing. Human-in-the-loop testing techniques have been developed recently to explore complex applications with human-generated seeds. For instance, HaCRS [44] provides an emulated terminal for humans to interact with the target application and collect possible description related to the applications current behavior. IJON [20] annotates the source code of the target application with customized primitives to guide the testing. Compared with these solutions, NAUTILUS's annotation strategy is less intrusive because the annotated OAS can be normally parsed by other applications. It is also human-readable, which makes it possible to be updated by humans during the testing, or automatically updated based on predefined rules.

8 Conclusion

We propose NAUTILUS, an automated vulnerability detection tool for RESTful services. tool is designed to uncover multi-API vulnerabilities, which are exploited by performing multiple API operations in certain sequences. NAUTILUS parses the OpenAPI specifications to understand the relations between API endpoints, and uses novel annotation primitives to label the operations and parameters for the generation of logical operation sequences. During the testing phase, NAUTILUS explores potentially vulnerable API endpoints and automatically updates annotations from the dynamic feedback. The evaluation on 6 benchmark services shows that NAUTILUS outperforms state-of-the-art techniques in both vulnerability identification and coverage. We use NAUTILUS to uncover 23 zero-day vulnerabilities in real-world RESTful applications.

References

[1] Anonymous Submission Website for Nautilus. https://sites.google.com/view/nautilus-testing.

[2] OpenAPI Specification Version 3. https://swagger.io/specification/.

[3] OWASP API Security project. https://owasp.org/www-project-api-security/.

[4] How to test a REST API. https://docs.rapid7.com/appspider/how-to-test-a-rest-api/, 2018.

[5] w3af: Open Source Web Application Security Scanner. https://w3af.org/, 2018.

[6] Azure rest api reference, 2021. https://learn.microsoft.com/en-us/rest/api/azure/.

[7] Cloud apis, 2021. https://cloud.google.com/apis.

[8] Fast, Secure Managed WordPress Hosting. https://wordpress.com/, 2021.

[9] Fun & Flexible Software for Online Communities, Teams, and Groups. https://buddypress.org/, 2021.

[10] Google cloud penetration testing: A complete guide, 2021. https://www.getastra.com/blog/security-audit/google-cloud-penetration-testing/.

[11] Penetration testing, 2021. https://learn.microsoft.com/en-us/azure/security/fundamentals/pen-testing.

[12] Rest api handbook, 2021. https://developer.wordpress.org/rest-api/.

[13] The state of api security – q1 2021, 2021. https://www.securitymagazine.com/articles/94509-of-organizations-had-api-security-incident-last-year.

[14] The OWASP Zed Attack Proxy (ZAP). https://www.zaproxy.org/, journal=OWASP ZAP, 2021.

[15] Gila cms. https://gilacms.com/, 2022.

[16] Navigate CMS. https://www.navigatecms.com/en/home, 2022.

[17] OWASP Juice-Shop Project. https://owasp.org/www-project-juice-shop/, 2022.

[18] SEO Panel: World's first SEO Control Panel for Multiple Websites. https://www.seopanel.org/, 2022.

[19] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1), January 2019.

[20] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612, 2020.

[21] Atlassian. Atlassian: Software development and collaboration tools. https://www.atlassian.com/, 2022.

[22] Atlassian. Confluence: Your remote-friendly team workspace. https://www.atlassian.com/software/confluence, 2022.

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

[24] erev0s. Erev0s/VAmPI: Vulnerable REST API with OWASP top 10 Vulnerabilities for Security Testing. https://github.com/erev0s/VAmPI, 2020.

[25] Roy Thomas Fielding and Richard N. Taylor. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, 2000. AAI9980887.

[26] Fuzzdb-Project. FuzzDB: Dictionary of attack patterns and primitives for black-box application fault injection and resource discovery. =https://github.com/fuzzdb-project/fuzzdb, 2022.

[27] Bernhard Garn, Jovan Zivanovic, Manuel Leithner, and Dimitris E Simos. Combinatorial methods for dynamic gray-box sql injection testing. *Software Testing, Verification and Reliability*, 32(6):e1826, 2022.

[28] François Gauthier, Behnaz Hassanshahi, Ben Selwyn-Smith, Trong Nhan Mai, Maximilian Schlüter, and Micah Williams. Backrest: A model-based feedback-driven greybox fuzzer for web applications. *ArXiv*, abs/2108.08455, 2021.

[29] Bernardo Guimaraes and Miroslav Stampar. sqlmap: Automatic SQL injection and database takeover tool. https://sqlmap.org/, 2022.

[30] Manuel Leithner, Bernhard Garn, and Dimitris E. Simos. Hydra: Feedback-driven black-box exploitation of injection vulnerabilities. *Information and Software Technology*, 140:106703, 2021.

[31] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*, 10:707–710, 1965.

[32] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao. Morest: Model-based restful api testing with execution feedback. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1406–1417, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

[33] Yi Liu. Restcluster: Automated crash clustering for restful API. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 198:1–198:3. ACM, 2022.

[34] Yi Liu. Restinfer: automated inferring parameter constraints from natural language restful API descriptions. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1816–1818. ACM, 2022.

[35] D.P. Lopresti. Robust retrieval of noisy text. In *Proceedings of the Third Forum on Research and Technology Advances in Digital Libraries,*, pages 76–85, 1996.

[36] Microsoft. Cloud computing services: Microsoft azure, 2022.

[37] MITRE. CVE Project by MITRE. https://cve.mitre.org/cve/search_cve_list.html, 2021.

[38] MITRE. Common Weakness Enumeration (CWE). https://cwe.mitre.org/index.html, 2021.

[39] MITRE. CVE-2021-21389, 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21389.

[40] Offensive Security. Exploit Database. https://www.exploit-db.com/, 2021.

[41] OpenAPI. OpenAPI Specification. https://swagger.io/specification/, 2020.

[42] OWASP. Nodegoat: The owasp nodegoat project. https://github.com/OWASP/NodeGoat, 2021.

[43] s0md3v. S0md3v/xsstrike: Most advanced xss scanner. https://github.com/s0md3v/XSStrike, 2022.

[44] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 347–362, New York, NY, USA, 2017. Association for Computing Machinery.

[45] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[46] E. Viglianisi, M. Dallago, and M. Ceccato. Resttestgen: Automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152, Los Alamitos, CA, USA, oct 2020. IEEE Computer Society.

[47] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. Automatic web testing using curiosity-driven reinforcement learning. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 423–435. IEEE, 2021.

Table 6: Selected Reproducible CVEs that contain RESTful API vulnerabilities. For `Component`: -: the vulnerability is from the target application native component; otherwise the plugin name is listed. For `Vulnerability Identification Tools`: ✓: this tool can identify the vulnerability, ✗: this tool cannot identify the vulnerability

Target Application	Version	CVE/Issue-ID	Vulnerability Type	Multi-API	Vulnerability Identification Tools				
					NAUTILUS	RESTLER	MOREST	ZAP	w3af
WordPress	-	CVE-2019-8942	Command Injection	✓	✗	✗	✗	✗	✗
	-	CVE-2022-21661	SQL Injection	✓	✓	✗	✗	✗	✗
	-	CVE-2019-8943	Path Traversal	✓	✗	✗	✗	✗	✗
	Modern Events Calendar	CVE-2021-24145	Command Injection	✓	✗	✗	✗	✗	✗
	Modern Events Calendar	CVE-2021-24146	Command Injection	✗	✓	✓	✓	✓	✓
	Backup Guard	CVE-2021-24155	Privilege Escalation	✓	✓	✗	✗	✗	✗
	Responsive Menu	CVE-2021-24160	Command Injection	✗	✓	✗	✗	✗	✗
	SP Manager	CVE-2021-24347	Command Injection	✓	✓	✗	✗	✗	✗
	Deplcate	CVE-2021-43408	SQL Injection	✗	✓	✓	✓	✓	✗
	Secure Copy	CVE-2021-24931	SQL Injection	✗	✓	✓	✓	✓	✓
	WP Visitor Statistics	CVE-2021-24750	SQL Injection	✗	✓	✓	✓	✗	✗
	Registration Magic	CVE-2021-24862	SQL Injection	✓	✓	✗	✓	✗	✗
	Modern Events Calendar	CVE-2021-24946	SQL Injection	✗	✓	✓	✓	✓	✓
	Download Monitor	CVE-2021-24786	SQL Injection	✓	✗	✗	✗	✗	✗
	WPGateway	CVE-2022-3180	Privilege Escalation	✗	✓	✓	✓	✓	✓
	Elementors	CVE-2021-24786	Code Injection	✓	✗	✗	✗	✗	✗
OpenEMR	-	CVE-2018-15142	Command Injection	✓	✓	✓	✓	✗	✗
	-	CVE-2017-9380	SQL Injection	✓	✗	✗	✗	✗	✗
	-	CVE-2018-15139	Command Injection	✓	✓	✗	✗	✗	✗
	-	CVE-2018-15152	Privilege Escalation	✓	✓	✓	✓	✓	✓
	-	CVE-2019-14530	Path Traversal	✓	✓	✓	✓	✗	✗
Umbraco	-	CVE-2020-9472	Command Injection	✓	✗	✗	✗	✗	✗
	-	CVE-2020-5811	Path Traversal	✓	✓	✗	✗	✗	✗
Drupal	-	CVE-2018-7600	Command Injection	✗	✓	✗	✓	✗	✗
	-	CVE-2020-13671	Command Injection	✓	✗	✗	✗	✗	✗
DotCMS	-	CVE-2022-26352	Command Injection	✓	✗	✗	✗	✗	✗
	-	CVE-2017-3187	CSRF	✓	✓	✗	✗	✓	✓
NavigateCMS	-	CVE-2022-28117	CSRF	✗	✓	✗	✗	✓	✓
	-	CVE-2020-14014	Reflected XSS	✗	✓	✓	✓	✓	✓
Knowage Suite	-	CVE-2021-30211	Stored XSS	✗	✓	✗	✗	✓	✓
	-	CVE-2018-12355	Reflected XSS	✗	✓	✓	✓	✓	✓
	-	CVE-2018-12353	CSRF	✗	✓	✓	✓	✓	✓
	-	CVE-2018-12354	CSRF	✗	✓	✗	✗	✓	✓
Backdrop	-	CVE-2021-45268	CSRF	✗	✓	✗	✗	✓	✓
	-	CVE-2022-42092	Code Injection	✓	✗	✗	✗	✗	✗
	-	CVE-2022-42095	Stored XSS	✓	✓	✗	✓	✗	✗
	-	CVE-2022-42094	Stored XSS	✓	✓	✗	✗	✗	✗
Piwigo	-	CVE-2022-48007	XSS	✓	✓	✓	✓	✓	✓
	-	CVE-2012-2209	Stored XSS	✗	✓	✓	✓	✓	✓
	-	CVE-2012-2208	Directory Traversal	✓	✓	✓	✓	✓	✓
	-	CVE-2018-5692	Stored XSS	✗	✓	✓	✓	✓	✓
Gitlab	-	CVE-2022-2185	Command Injection	✓	✓	✗	✗	✗	✗
	-	CVE-2022-2884	Command Injection	✓	✗	✗	✗	✗	✗
	-	CVE-2021-22205	Privilege Escalation	✓	✗	✗	✗	✗	✗
	-	CVE-2022-1175	Stored XSS	✓	✗	✗	✗	✗	✗
Casdoor	-	CVE-2022-24124	SQL Injection	✓	✓	✓	✓	✗	✗
	-	CVE-2022-44942	Directory Traversal	✓	✗	✗	✗	✗	✗
Strapi	-	CVE-2019-19609	Command Injection	✓	✓	✓	✓	✗	✗
	-	CVE-2022-27263	Command Injection	✓	✗	✗	✗	✗	✗
	-	CVE-2022-0764	Command Injection	✓	✗	✗	✗	✗	✗