Bachelor's Thesis Computing Science

Fuzzing JSON API Clients

Harm Roukema
s1029070

June 29, 2023

First supervisor/assessor:
Dr. Ir. Erik Poll

Second assessor:
Dr. David Rupprecht

Radboud University

Abstract

In this thesis, we show how to test a JSON API client for bugs using fuzzing. JSON APIs are commonly confused with REST APIs, which have to satisfy more constraints. JSON APIs are already regularly fuzzed on the server side. However, the client side is often overlooked. By programming a small vulnerable client of a JSON API and testing it using our self-made *BasicFuzzer*, we show that it is possible to test JSON API clients for bugs. We create an expanded *OpenAPIFuzzer* with OpenAPI parsing capabilities and attempt to use it to parse the OpenAPI specification of a 5G network function, but ultimately do not succeed. We successfully fuzz the Radboud Osiris web app with our *OsirisFuzzer*, but do not find any bugs. Finally, we present *ProxyFuzzer*, a mitmproxy add-on that fuzzes responses from a JSON API, while acting as a proxy between the JSON API client and server. We show that *ProxyFuzzer* can also fuzz the Radboud Osiris web app.

Contents

Chapter 1

Introduction

Imagine a webshop with a warehouse. The warehouse contains a server that keeps track of the inventory, which the webshop can query for information. A possible scenario is that a costumer visits the webshop to check the availability of an item. The webshop now queries the warehouse server for this availability, which the warehouse server returns. To facilitate this communication, the warehouse server could implement an HTTP-based API, and use JSON as its data format. This is what we call a JSON API. JSON APIs are widely used in the web and software industry. For example, (RESTful) JSON APIs form the main communication channels between network functions in 5G mobile technology. For more information on JSON APIs, and their relation to RESTful APIs, see Section 2.1.1.

How can we verify the security of this JSON API? One possibility is to test the API for bugs using a fuzzer, an automatic software testing tool. This fuzzer could generate unexpected requests to the warehouse server, and see if it could cause it to crash. Dozens of fuzzers exist that could test the warehouse server in this scenario, we look at some of them in Chapter 4. One popular fuzzer is RESTler [9]. RESTler can extract a grammar from an OpenAPI specification of a JSON API, and fuzz according to that grammar. Nevertheless, we are omitting an equally important participant in the communication: the webshop.

The webshop, the client of the JSON API of the warehouse server, is not tested for bugs by fuzzers like RESTler. In fact, there do not exist fuzzers that are specifically designed to fuzz clients of JSON APIs. This is a potential security flaw: what happens when a malicious actor hijacks or impersonates the warehouse server? Could this actor cause the webshop to crash or malfunction, like in Figure 1.1? Fuzzing could provide us with answers to these questions.

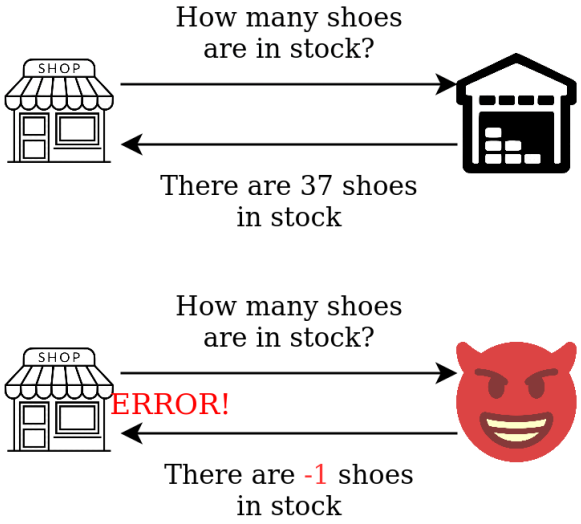


Figure 1.1: Example communication between the webshop and the warehouse or a malicious actor.

The goal of this research is to be able to fuzz an arbitrary JSON API client for bugs. One of the main motivations of this research is to eventually test the security of 5G network functions, so fuzzing one is a stretch goal. This research aims to fill the knowledge gap on JSON API client fuzzing by providing means to do so. Therefore, the research question is: *How can we test for bugs in a JSON API client using fuzzing?*. There will be no distinction made between security and non-security related bugs.

In this thesis, we create our own toy example of a JSON API client. This client contains some bugs in the handling of responses of the JSON API server. The main goal is to find these bugs in the client, using a fuzzer. This main goal is reached by executing the following steps:

1. Create a vulnerable client of a JSON API, programmed in Python (Section 3.3). This allows us full control over this JSON API client, which we think will be practical.

2. Pick an appropriate fuzzer, that can detect if the client is still running correctly (Section 3.2). There are numerous options to choose from, some more practical than other, so it is important that we carefully consider them.

3. Find vulnerabilities in the vulnerable client using our self-made *BasicFuzzer* (Section 3.4).

When this main goal is reached, the research question will be answered, as it will be shown how to fuzz a JSON API client.

After reaching this goal, we have a few stretch goals:

1. Fuzz a 5G network function with *OpenAPIFuzzer* (Section 3.5 & 3.6), a fuzzer that uses an OpenAPI specification.

2. Fuzz the real-world Radboud Osiris web app with *OsirisFuzzer* (Section 3.7) and with *ProxyFuzzer* (Section 3.8), a fuzzer that acts as a proxy to fuzz responses from the JSON API server.

Reaching these stretch goals shows how real-world JSON APIs can be fuzzed, and thus paints a more complete picture of JSON API client fuzzing.

This thesis is structured in the following way. Chapter 2 provides background information for anyone less familiar with certain topics of this thesis, or for anyone that wishes to brush up their knowledge. In Chapter 3, the bulk of the research takes place: we show how we created different fuzzers for JSON API clients and attempt to use these fuzzers to test a self-made vulnerable client, a 5G network function, and the Radboud Osiris web app. Chapter 4 discusses existing JSON API fuzzers, while Chapter 5 mentions potential future work for following research. Finally, we end the thesis with our conclusions in Chapter 6.

Chapter 2

Background

This chapter is meant to educate readers on foundational concepts that are frequently used in this thesis. We start by explaining what REST APIs are and what components they consist of in Section 2.1. We discuss the difference between JSON APIs and REST APIs in Section 2.1.1, and the role of OpenAPI specifications is also covered in Section 2.1.2. Next up, in Section 2.2, we talk about what fuzzers are and which parts are essential to their function. We also discuss the differences between fuzzing a JSON API client and server in Section 2.2.1.

2.1 REST APIs

A REST API (or RESTful API) is an application programming interface (API) that conforms to the architectural style of REST, short for representational state transfer. In this style, there exist a client and a server. This is an example of an architectural constraint: the client and the server should be able to function independently. REST defines several more architectural constraints [11]:

- The communication is stateless; no stored context on the server is needed to process requests.
- Responses to requests are implicitly or explicitly labeled as cachable or non-cachable.
- There exists a uniform interface between the client and the server.
- The system is layered; a client can not tell whether it is talking directly to a server or an intermediary.
- Optionally, servers can extend client functionality by providing executable code to the client.

HTTP Method	Meaning
GET	Retrieve a current representation of the target resource
POST	Process the supplied payload according to the resource
PUT	Replace all representations of a target resource with the provided representation
DELETE	Delete all representations of a resource
OPTIONS	Describe communication options for the resource

Table 2.1: An overview of the most common HTTP methods and their meaning within a REST API [10].

The uniform interface of a REST API can be understood by its four constraints:

- Identification of resources: resources are identified by, for example, URIs.
- Manipulation of resources: a client has enough information to modify a resource if it has permission to do so.
- Self-descriptive messages.
- Hypermedia as the engine of application state (abbreviated to **HATEOAS**), where a server provides links to other resources to the client depending on the state. This allows a REST API client to discover other resources on the server.

Most commonly, the client and server communicate via the HTTP protocol. Distinct HTTP methods provide different functionality for interacting with resources, such as creating, reading, updating and deleting (also known as CRUD) them, see Table 2.1. Resources can have multiple representations (data formats), such as most commonly JSON, JavaScript Object Notation, or XML, Extensible Markup Language.

REST is generally used as a guideline for implementing APIs on the Internet, but also in other parts of the software industry. For example, the communication between different network functions of 5G is done via REST APIs [22].

2.1.1 JSON API vs. REST API

The terms "JSON API" and "REST API" are often confused [15] [12]. This is also the case for some self-proclaimed REST API fuzzers we cover in Chapter 4. When we refer to a JSON API in this thesis, we are talking about an API over HTTP that uses JSON as its data format in requests and

responses[1]. As mentioned above, a REST API has a lot more architectural and interface constraints, like HATEOAS, while not limited to only JSON as its data format.

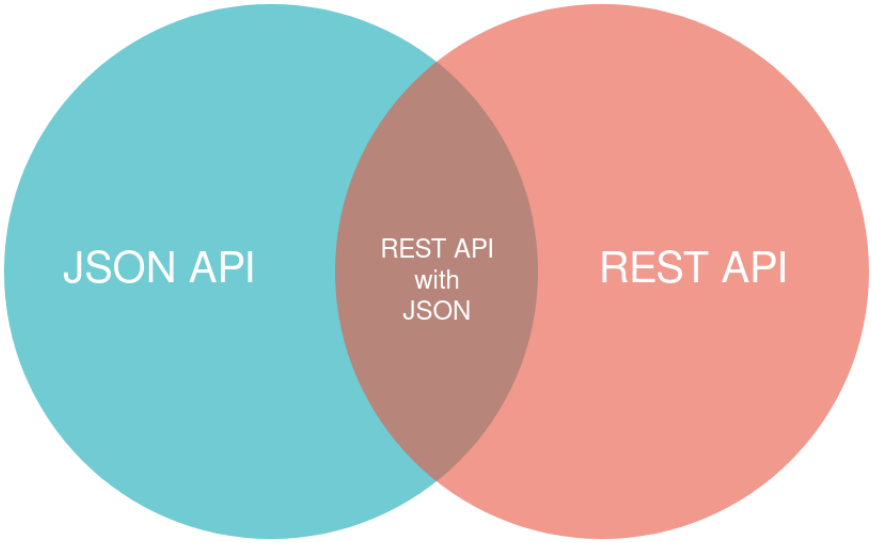


Figure 2.1: A Venn diagram that shows the relation between a JSON API and a REST API.

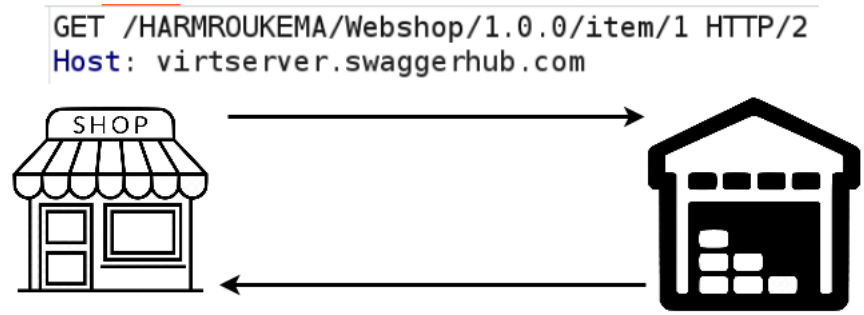
The relation between a JSON API and a REST API is illustrated in Figure 2.1. The example of the JSON API given below belongs in the blue section, as it does not adhere to some REST constraints like HATEOAS. A REST API that uses XML as its data format would belong in the red section. The APIs of 5G network functions are RESTful and use JSON, so they belong in the intersection.

A client or server JSON API fuzzer would also be able to fuzz a client or server REST API that uses JSON, and vice versa, because they use the same data format, while the extra constraints of a REST API are not used in the fuzzing process. A REST API server fuzzer could however potentially use the HATEOAS property of the REST API to discover new endpoints to fuzz, which is not generally possible for JSON APIs.

Example of a JSON API As an example of a JSON API, let us reconsider the example of the webshop communicating with a warehouse via a JSON API. In this case, the webshop is the client, while the warehouse is the server. Say that the webshop wants to see if certain shoes are still available and what their price is. The webshop can now make an HTTP GET request to the `/item/1` endpoint of the warehouse, see Figure 2.2.

[1]Not to be confused with an API that adheres to the JSON:API specification, see `https://jsonapi.org/format/`

This requests information about the item with ID 1, which is the ID of the shoes. The warehouse now responds with status code 200 (meaning the request was successful) and with information about the shoes in JSON format. This includes the price and the availability of the shoes and concludes the communication.




```
HTTP/2 200 OK
Date: Mon, 31 Oct 2022 15:49:23 GMT
Content-Type: application/json;charset=utf-8
Content-Length: 263
Access-Control-Allow-Headers: X-Requested-With,Content-Type,Accept,Origin
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: *
Access-Control-Allow-Credentials: true
Server: Jetty(9.4.29.v20200521)

{
  "id":0,
  "category":{
    "id":0,
    "name":"footwear"
  },
  "name":"Nike AF1",
  "price":119.99,
  "tags":[
    {
      "id":0,
      "name":"new"
    }
  ],
  "available":true,
  "comment":"string",
  "dateAdded":"2015-07-20T15:49:04-07:00"
}
```

Figure 2.2: Example of a GET request with a response in a JSON API.

2.1.2 OpenAPI

The most widely adopted industry standard for describing JSON APIs is OpenAPI [3]. An OpenAPI description document is a YAML or JSON file that describes a JSON API according to the OpenAPI specification, formerly known as Swagger specification. By formally describing a JSON API, one can use automated tools to process its description, which allows

for functionality like mocking, the act of automatically creating a fake API server with example responses.
A typical OpenAPI specification contains the following fields [5]:

- `openapi`: contains the version of the OpenAPI Specification the document is using.
- `info`: provides information about the API, like its author, title, and description.
- `paths`: contains all endpoints of the API. Every endpoint contains all available HTTP methods for that endpoint, which includes possible parameters and responses to the requests.
- `components`: custom structures that can be reused throughout the document, to avoid duplication.

For an example of an OpenAPI specification, see Appendix A.1.

2.2 Fuzzing

Fuzzing is the act of randomly generating input and feeding it into a program in hope to uncover bugs or vulnerabilities [28], flaws in a system that could be exploited to violate the system's security policy [23]. The term fuzzing was coined by Miller, who crafted a programming exercise for his students which led them to create the first fuzzers [21]. It is currently the most popular vulnerability discovery technique [17]. Compared to other techniques like static code analysis, fuzzing is relatively easy to do, has a high accuracy, and does not require extensive knowledge of the target application, also known as the system under test (**SUT**) [20].
The traditional fuzzing process can be dissected into several stages [17], as shown in Figure 2.3:

1. A test case, random program input, is generated. This test case should fit the requirements of the input format of the program, while still being broken enough to likely result in a crash. There are multiple methods to generate test cases, which we will discuss below. Test cases can be any form of input to the SUT, like files, text, or network packets.
2. The test case is fed to the SUT as input and the targeted program is executed.
3. The state of the SUT is monitored for exceptions or crashes.
4. Exceptions or crashes that have occurred are analyzed or categorized.

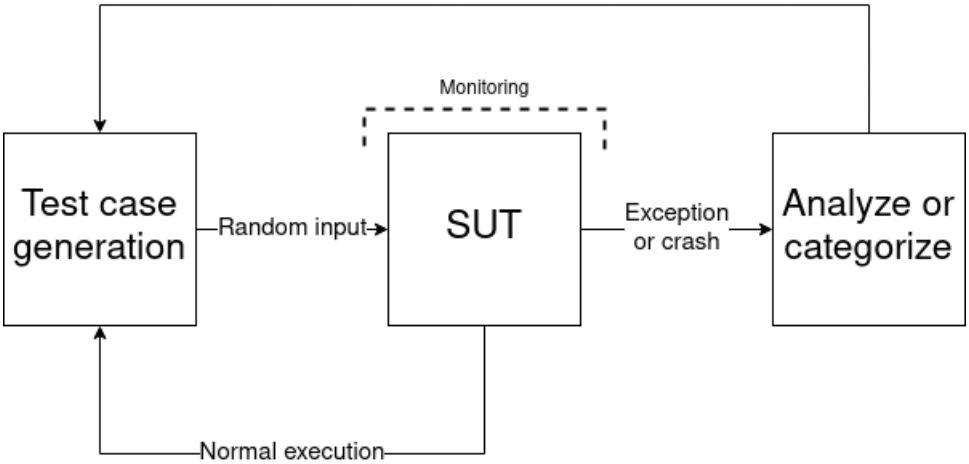


Figure 2.3: A flowchart of the traditional fuzzing process.

These stages are repeated for as many test cases as desired, until one hopes that most of the bugs are found.

The manner in which the test cases are generated is of great importance for the success of the fuzzing. Test cases in fuzzing are generally generated using one of two methods: generation-based or mutation-based [17]. In **generation-based** fuzzing, a generator creates test cases with knowledge of the inputs' data format. This has the advantage of making sure that the test case adheres to the data format, however, it requires knowledge of the data format, which is not always provided. **Grammar-based** fuzzing is a form of generation-based fuzzing, where a specification of legal inputs via a so-called grammar forms the basis of generated input [28]. The other method of test case generation is **mutation-based** fuzzing, where test cases are generated by modifying parts of provided inputs in a mutation process. A mutator randomly modifies bytes of the seed inputs with random or special values. This method is less efficient than generation-based fuzzing if the inputs are modified randomly, as inputs are less likely to adhere to a data format, thus, to determine the location and value to modify is the main challenge of mutation-based fuzzing.

Another less common test case generation method is **search-based** fuzzing [28]. When we have an idea of what inputs we are looking for, we can search for them. Instead of inputs randomly, we can use domain-knowledge to create a heuristic, and use it to search for inputs. This method requires knowledge of the type of inputs we want to find.

Apart from distinguishing between generation-based, mutation-based and search-based fuzzers, there is another, separate way to distinguish between fuzzers. We can also distinguish between three different types of

fuzzers: blackbox, whitebox and greybox fuzzers [13]. **Blackbox** fuzzers randomly generate input for a program without knowledge of its internal function. **Whitebox** fuzzers systematically explore different execution paths in a program. This generally gives better code coverage than blackbox fuzzers [29], and thus, whitebox fuzzers are generally better at finding bugs than blackbox fuzzers. However, whitebox fuzzers are far more difficult to implement. **Greybox** fuzzers are blackbox fuzzers extended with whitebox fuzzing techniques. An example of a greybox fuzzer is American Fuzzy Lop (AFL) [27]. AFL is not a blackbox fuzzer because it leverages some program analysis, but it also is not a whitebox fuzzer because it does not build on program analysis entirely. More specifically, AFL utilizes coverage-guided feedback, which is a method that monitors which parts of the program are executed by each test case, and uses this information to direct new test cases.

But what kind of bugs do fuzzers find? One possible type of bug is a buffer overflow, in which the maximum length for the input gets exceeded [28]. Fuzzers can find these types of bugs by giving longer input than expected to the SUT. Other bugs can for example be caused by missing error checks or rogue numbers. To summarize: a large variety of bugs can be found using fuzzing.

For information on fuzzers for JSON API servers, see Chapter 4. Surprisingly, all JSON API fuzzers are only able to fuzz JSON API servers. They only fuzz requests to the server, not the responses to the client. This is the main motivation of this thesis to focus on JSON API client fuzzing. We can speculate that this is caused by less interest in the security of JSON API clients and some difficulties that come with fuzzing of JSON API clients, which we will discuss next.

2.2.1 Fuzzing JSON API clients vs. servers

Due to the design of JSON API clients and servers, fuzzing them differs in two main ways:

1. **Crash detection**: when fuzzing a JSON API server, the server will generally respond to the fuzzed request. This response can contain information about whether we have found a bug. For example, when the server returns a 500 Internal Server Error, this could be an indication. Also, when the server does not send a response, this could also be informative, as it could indicate that the server has crashed or hangs. JSON API clients do generally not respond to fuzzed responses, so it is challenging to detect bugs, errors, or crashes in the JSON API client.

2. **Automation**: JSON API servers are generally listening for requests at any time, so a fuzzer can easily send multiple requests in succession.

JSON API clients are generally only listening for responses after they have sent a request themselves first. This makes it harder to automate the fuzzing process. Ideally, we want to be able to prompt the JSON API client to make a request during the fuzzing process.

Chapter 3

Fuzzing JSON API clients

This chapter contains the bulk of the research done in this thesis. It aims to answer the research question, *How can we test for bugs in a JSON API client using fuzzing?*, in multiple steps. We start in Section 3.1 with formulating our requirements for the fuzzer. We then choose to use Flask with PyJFuzz according to these requirements in Section 3.2. Next up, we program a vulnerable JSON API client in Section 3.3, which we then use to test our self-made *BasicFuzzer* on in Section 3.4. After we have succeeded in fuzzing a self-made vulnerable client, we first expand our fuzzer to create *OpenAPIFuzzer* in Section 3.5 before trying to fuzz real-world applications, namely a 5G network function in Section 3.6 and the Radboud Osiris web app using our *OsirisFuzzer* in 3.7. We finally present *ProxyFuzzer*, a mitmproxy add-on that acts as a proxy, and show that it can also fuzz the Osiris web app in Section 3.8.

3.1 Fuzzer requirements

Before we start to investigate possible fuzzers to use, it is important to work out the requirements of the fuzzer. This will help with making a calculated choice of the fuzzer.
Our main requirements are as follows:

- The fuzzer needs to be able to act as an HTTP server. This will allow the client to connect to the fuzzer.
- The fuzzer needs to be able to fuzz JSON objects. This will be the format in which the fuzzer will return data.

We also have some optional requirements:

- The fuzzer should be able to check if the client has crashed, for example by checking if the TCP connection is still active.

- The fuzzer should to be able to fuzz the HTTP header, including parameters like the response code and the content length.
- The fuzzer should be able to prompt the client to make a request repeatedly. This allows the fuzzer to fuzz multiple responses instead of just one response. Otherwise, the client has to be prompted to make a request manually by a user, for example.
- The fuzzer should be able to be generated from just an OpenAPI specification.

When creating these requirements, we made the following choices:

- To minimize complexity, the fuzzing will be done over HTTP instead of HTTPS. TLS support would be nice in practice, but is not a key part of a JSON API, which allows us to leave it out.
- Although one of the optional requirements is the ability to fuzz the HTTP header, this will not be the focus of the fuzzing. Most of the application logic of JSON API clients will depend on the data being returned by the server, not HTTP headers.

3.2 Picking a fuzzer or fuzzing framework

There exist a lot of different fuzzers that could be valid candidates for JSON API client fuzzing. There also exist fuzzing frameworks: these are programming libraries that allow us to easily create fuzzers from them, which also makes them possible candidates. In this section, we will shortly talk about the most obvious and promising possibilities, before finally picking a fuzzer or fuzzing framework to use.

- **JSON API server fuzzers**: Some of our JSON API server fuzzers discussed in Chapter 4, like RESTler, can generate a grammar from just an OpenAPI specification, which would be a nice feature for JSON API client fuzzing. However, these fuzzers are unable to act as an HTTP server or to fuzz a JSON API client, which is one of our main requirements. In other words, we are unable to pick one of these fuzzer, without fundamentally changing the inner workings by hand. Therefore, picking a JSON API server fuzzer is probably not the best approach.
- **Kitty**: Kitty [4] is a fuzzing framework written in Python and inspired by the Sulley and Peach fuzzers. Kitty is not a fuzzer in itself: it is a fuzzing framework to program your own fuzzers, which means that using it requires more work than ready-made fuzzers. But because of

this freedom in implementation and extensibility, it fits all our requirements. It is not a fuzzing framework made specifically for JSON APIs, but it allows users to create fuzzers for a variety of protocols, like TCP. It also includes a client fuzzing tutorial in its documentation [2], which is quite handy. It does not support JSON fuzzing, however, we can extend its functionality with a JSON fuzzing library. One disadvantage of Kitty is that it has not been in active development since 2018, but this does not necessarily have to be an issue.

- **BooFuzz**: BooFuzz[1] is a fork and successor of the Sulley fuzzing framework that focuses on extensibility. It allows a user to create grammar-based blackbox fuzzers. One benefit of using BooFuzz would be that it includes an example where an TLS client is fuzzed. This example could be used to create an HTTP client fuzzer. However, it does not include any JSON specific fuzzers, which means that we would have to implement our own JSON fuzzer or use an existing JSON fuzzer within BooFuzz.

- **Flask with a JSON fuzzer**: Flask[2] is a Python web framework that allows programmers to easily setup web servers in Python with advanced functionality like URL routing and templating. We can combine a Flask web server with a JSON fuzzer to easily create a JSON API fuzzer. One of the main benefits of this approach is its simplicity, as one could argue that using a fuzzing framework like Kitty or BooFuzz is unnecessarily complex, while a simple web server combined with a JSON fuzzing library is sufficient for client side JSON API fuzzing. It also allows for great control over the technical setup.

We decided on going for Flask with a JSON fuzzer. We chose this for its simplicity: combining a JSON fuzzer with another fuzzing framework is not needed, and thus overly complex. There exists no ready-made fuzzer for JSON API clients, so using a JSON fuzzer combined with a web server is the closest alternative.

There exist a few JSON fuzzers that we can use with Flask:

- **PyJFuzz**[3]: PyJFuzz is JSON fuzzing library in Python. According to its developers, PyJFuzz provides ready-to-use Python classes and functions that allow you to fuzz JSON inputs. The fuzzer is mutation-based and blackbox. It is able to produce payloads for different vulnerability types, like SQL injection or XSS. Moreover, it can not just fuzz the JSON key and value fields, but also the JSON structure. It

[1] `https://github.com/jtpereyda/boofuzz`
[2] `https://flask.palletsprojects.com/en/2.2.x/`
[3] `https://github.com/mseclab/PyJFuzz`

is the most popular JSON fuzzer we found, with over 350 stars on GitHub.

- **Snodge**[4]: Snodge is a library written in Kotlin that can randomly mutate JSON, but also XML, text and binary data. It does not fuzz the JSON structure, but it can fuzz key and value fields.
- **Templated JSON Fuzzer**[5]: This is an implementation of a templated (or grammar-based) fuzzing engine for JSON in Python. It is not a very popular JSON fuzzer, but its inner workings are very well documented. It does not fuzz the JSON structure, only key and value fields.
- **JSON::Fuzz::Generator**[6]: This JSON fuzzer is written in Ruby. Just like Snodge and Templated JSON Fuzzer, it only fuzzes key and value fields, not the JSON structure.

We decided that we wanted to use a JSON fuzzer written in Python, so that we can easily combine it with a Flask program. We ended up going for PyJFuzz, because it is the most popular JSON fuzzer, it implements XSS payloads, which can be interesting when fuzzing a web app, and it allows us to also fuzz the JSON structure.

After using Flask with PyJFuzz in the rest of our research, we found out that PyJFuzz comes with its own web server[7], which could mean that using Flask for the web server is unnecessary. However, we could not get this web server to work. Using Flask also allows us better control over the web server, so we do not regret using it.

3.3 Vulnerable client

We create a vulnerable JSON API client to test our fuzzer on, because this keeps the technical setup simpler and easier to control than if we start with fuzzing an existing real-world JSON API client.

We start by designing an OpenAPI[8] specification of our imaginary JSON API, see Appendix A.1. We have chosen to use OpenAPI to design the simple JSON API because it allows us to easily mock the server in SwaggerHub[9]. Moreover, it provides a nice overview of the API structure and could allow possible fuzzers to gain a better understanding of the API. The JSON API

[4] `https://github.com/npryce/snodge`

[5] `https://github.com/arbitraryrw/templated-json-fuzzer`

[6] `https://github.com/deme0607/json-fuzz-generator`

[7] `https://github.com/mseclab/PyJFuzz#built-in-tool`

[8] For more information about OpenAPI specifications, see Section 2.1.2

[9] A tool for working creating OpenAPI specifications, `https://swagger.io/tools/swaggerhub/`

is based on our webshop example from Chapter 1 and a very stripped down version of the Pet Store example from Swagger[10].

The `item` schema of the OpenAPI specification is the main schema. It includes most data types (like integer, float, boolean, etc.), as different data types could be interesting for a fuzzer.

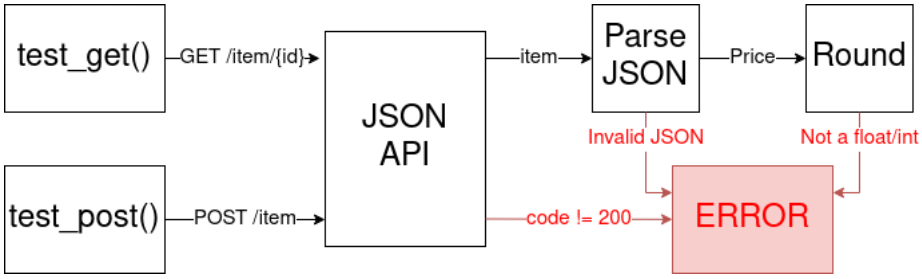


Figure 3.1: An overview of the control flow of the vulnerable JSON API client.

See Figure 3.1 for an overview of the control flow of the vulnerable client. The Python code of the vulnerable client, see Appendix A.2, uses the `requests` library, which allows us to easily make HTTP requests to the fuzzer. Because a possible way to do the alive-check of the SUT is to check if the TCP connection is still active, we make sure that the requests are made by the same `Session` object. This keeps the TCP connection open between requests instead of closing it immediately afterward.

The vulnerable client sends two types of requests to the `/item` endpoint: a GET and a POST request. The main idea is that these requests will be repeatedly sent in a loop.

The `test_get(id)` function sends an HTTP GET request to `/item/{id}` using the created `Session` object. It then tries to parse the JSON with the `json.loads()` function. This is the first bug in the vulnerable client: the client will crash if the JSON returned by the fuzzer is not valid. Next up, the code rounds the price of the requested item. This is the second bug in the client: the data type of price is not verified and could for example be a string, which would cause a crash.

The `test_post()` function sends an HTTP POST request to `/item` containing an example item in the request body. It then checks whether the response code was equal to 200, which means everything went as expected. If the response code is not equal to 200, the client exits. This is the third bug in the vulnerable client: it does not properly handle different response codes.

To summarize, the vulnerable client contains three bugs:

1. It parses JSON without verifying the JSON is of sound structure.

[10] `https://petstore.swagger.io/`

2. It rounds the price without checking whether the price is actually a float type.

3. It errors when the response code of the POST request is unequal to 200.

3.4 *BasicFuzzer*: Fuzzing the vulnerable client

Before we can start with fuzzing the vulnerable client, we have to program the fuzzer in Python using Flask and PyJFuzz. We will call this fuzzer "*BasicFuzzer*", to avoid confusion when we expand on it later. We choose to only fuzz the GET request of the vulnerable client, because we do not prioritize HTTP header fuzzing, as described in Section 3.1. We give an overview of the control flow of *BasicFuzzer* in Figure 3.2. For the code of *BasicFuzzer*, see Appendix B.

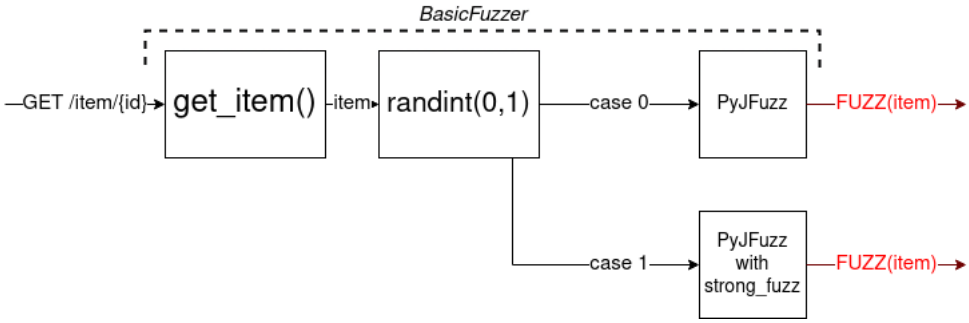


Figure 3.2: An overview of the control flow of *BasicFuzzer*.

Whenever a JSON API client sends a GET request to the `/item/1` endpoint, the `get_item(item_id)` function will be called with an `item_id` of 1. *BasicFuzzer* will then use a dictionary that represents our item, as defined in our OpenAPI specification in Appendix A.1, and pass it to the PyJFuzz library. We flip a coin to determine in which way we fuzz: in the first case, we configure PyJFuzz to also fuzz the JSON structure with the `strong_fuzz` parameter. This is useful for fuzzing the JSON parser of the vulnerable client. In the second case, we fuzz parameters while maintaining the JSON structure. We set the fuzzing techniques of PyJFuzz to only the number 13, which is a random character attack. Other attacks like an SQL injection payload are also possible, but unnecessary for our purposes.

3.4.1 Results

We ran *BasicFuzzer* 50 times on the vulnerable client. The results are shown in Table 3.1. We see that *BasicFuzzer* is consistently able to find the

Iterations	50
Average time (s)	0.111
Average number of responses	2.32
Times JSON bug found	41
Times rounding bug found	9
JSON parsing bug percentage	82%

Table 3.1: Summarized results of fuzzing the vulnerable client with *Basic-Fuzzer*

first two bugs: the `strong_fuzz` parameter of PyJFuzz allows us to break the parsing of `json.loads()` in the vulnerable client in Figure 3.3, while fuzzing the data type of the `price` of an item breaks the rounding function in Figure 3.4. The vulnerable client crashed after 2.32 responses on average. The highest amount of responses it took to crash the vulnerable client was 10, while 1 response was the most common. When the vulnerable client crashed, it was most often caused by the JSON parsing bug: 82% of the time. The vulnerable client crashed on average after 0.111 seconds, which we measured with the `time` Linux shell command.

Figure 3.3: `json.loads()` crashes the vulnerable client, as it fails to parse the fuzzed JSON structure.

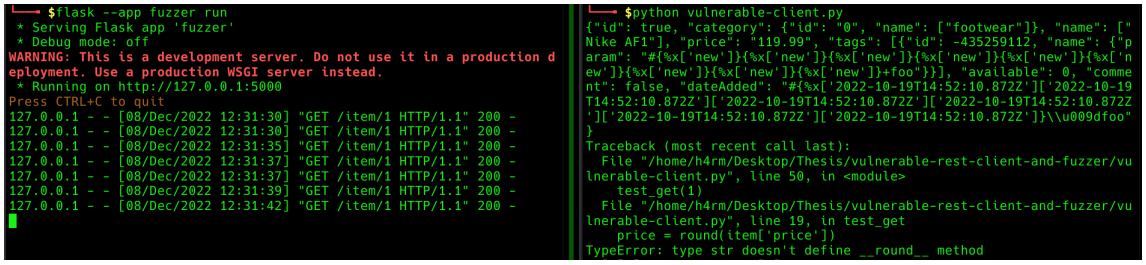


Figure 3.4: The fuzzed price was changed to a string, which makes the `round` function crash the vulnerable client.

BasicFuzzer in its current state still has some limitations:

- It can not detect a crash of the client, it only returns fuzzed responses.
- It can not ask the client to make a request. The fuzzing process is completely dependent on whether the client initiates it.
- It has to be tailored to the client. It would be more flexible if the basic fuzzer could generate responses based on for example an OpenAPI specification.

3.5 *OpenAPIFuzzer*: Adding OpenAPI parsing

Now that we have made our *BasicFuzzer*, we would like to expand it by resolving some of its limitations. Adding the ability to parse an OpenAPI specification seemed most interesting to us, as this would allow us to automatically tailor the fuzzer to a specific JSON API with an OpenAPI specification, which saves a lot of time and effort. Moreover, this functionality could also be used in combination with an OpenAPI specification generator like mitmproxy2swagger[11] to fuzz JSON APIs without an OpenAPI specification. We will call the resulting fuzzer "*OpenAPIFuzzer*".
There appear to be two possible options to add OpenAPI parsing:

- It is possible to generate a JSON example with the Java Swagger Inflector library and pass this to PyJFuzz[12]. This could be easier than the second option, but it does require the use of Java.
- Manually parse and generate examples with the `openapi3-parser` Python library[13]. We thought that this would be better practice than combining our Python fuzzer with Java code, but it could be more difficult to implement.

We picked the second option because we figured that only using Python is the better practice in the long run, and we expected that the programming task was doable. We were able to successfully implement this idea. An overview of the control flow of *OpenAPIFuzzer* is shown in Figure 3.5. For the code of *OpenAPIFuzzer*, see Appendix B.2.

OpenAPIFuzzer parses a given OpenAPI specification, and finds out to which endpoints requests are made by the client. It also generates example JSON payloads that will be given to PyJFuzz, which will create fuzzed JSON based on these payloads. *OpenAPIFuzzer* uses two main functions

[11] `https://github.com/alufers/mitmproxy2swagger`
[12] `https://stackoverflow.com/questions/41408768/how-to-generate-json-examples-from-openapi-swagger-model-definition`
[13] `https://pypi.org/project/openapi3-parser/`

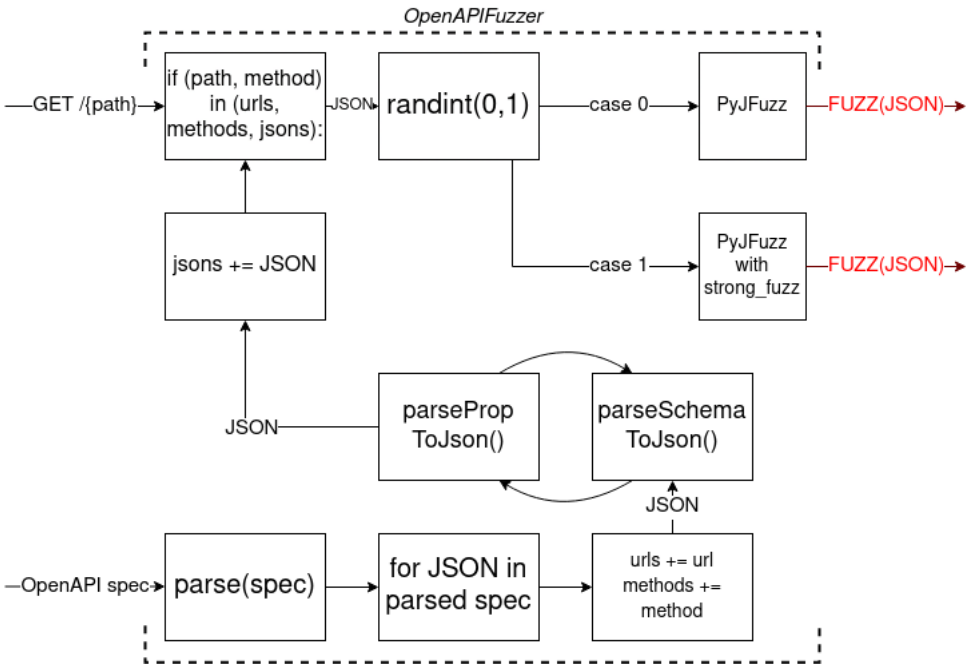


Figure 3.5: An overview of the control flow of *OpenAPIFuzzer*.

to parse the OpenAPI specification, which are sometimes called recursively: `parseSchemaToJson()` and `parsePropToJson()`.

After parsing the OpenAPI specification, the URLs of the endpoints are transformed into regular expression friendly form by the function `processUrls()`. This function removes the first '`/`' of the URL and replaces words in curly brackets with a regular expression match. For example, the URL '`/item/{itemId}`' is changed to '`item/.*`'.

We define a Flask route that matches all possible paths and desired HTTP methods. In the function `fuzz()`, we loop over the URLs and the methods collected from the OpenAPI specification. If we find a match, we respond with fuzzed JSON generated from the corresponding example JSON taken from the OpenAPI specification.

We can now run *OpenAPIFuzzer* on the OpenAPI specification in Appendix A.1 of the imaginary webshop JSON API. When we also run our vulnerable client, *OpenAPIFuzzer* is able to find bugs, as shown in Figure 3.6. We will not discuss results extensively as in Section 3.4.1, because *BasicFuzzer* and *OpenAPIFuzzer* have very similar results.

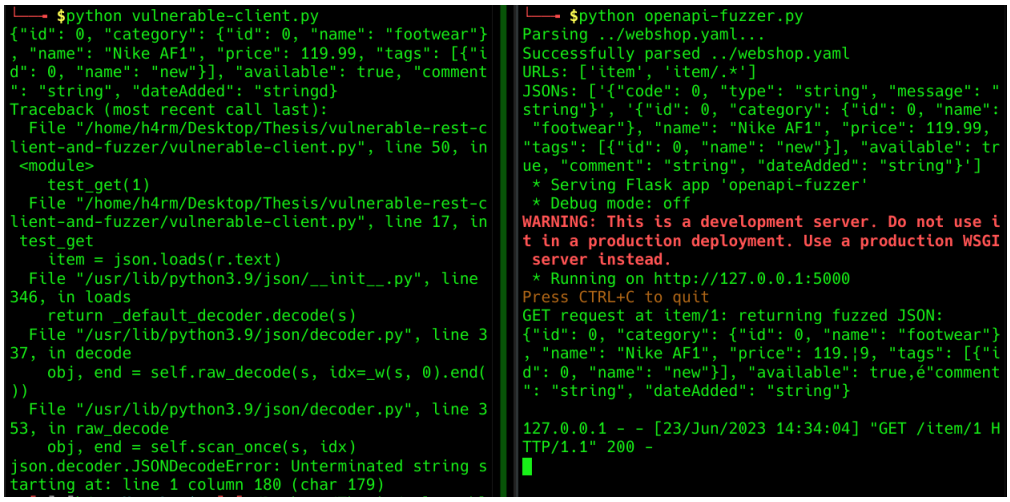


Figure 3.6: *OpenAPIFuzzer* successfully parses the OpenAPI specification of the webshop and uses it to fuzz the vulnerable client.

3.6 Attempt at 5G Fuzzing

One of the main sources of inspiration for this thesis was the communication between 5G network functions. Communication between 5G network functions is done via RESTful JSON APIs [22], which makes these network functions a great target for one of our fuzzers. Moreover, all network functions have a publicly available OpenAPI specification, which fits our *OpenAPIFuzzer* perfectly.

We started out by trying to run *OpenAPIFuzzer* on the network function NFManagement's OpenAPI specification[14]. This gave us multiple errors. After resolving these errors and even patching the `openapi3-parser` library to include more information about the errors, we ran into an error that we could not resolve.

We found out that the 'anyOf' keyword is not properly supported by the `openapi3-parser` library used in *OpenAPIFuzzer* at the time of writing. In fact, there exists a GitHub issue for this exact problem[15]. We thought about replacing all 'anyOf' cases by their first subschema. However, there are a lot of anyOf statements, so this idea is infeasible. Doing this would also oversimplify the OpenAPI specification and make it less representative for the actual API, which means that our *OpenAPIFuzzer* would not completely test the API.

[14] `https://github.com/jdegre/5GC_APIs/blob/Rel-18/TS29510_Nnrf_NFManagement.yaml`

[15] `https://github.com/manchenkoff/openapi3-parser/issues/5`. This issue has recently been resolved, see Chapter 5.

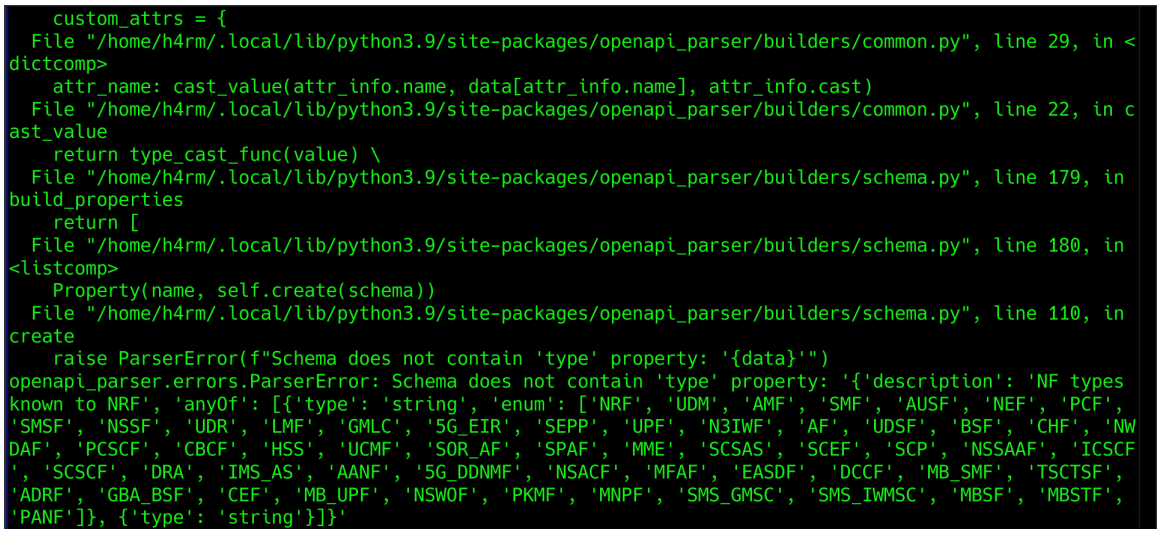


Figure 3.7: One of the errors given by the `openapi3-parser` library.

This is where we were unable to make any further progress. One of the ideas we had was to avoid the (apparently unfinished) `openapi3-parser` library and use a more generic YAML parser, but this will not work because of the referencing to schemas that is frequently done in OpenAPI specifications: we would have to implement referencing to other files, which we consider to be an infeasible task.

3.7 *OsirisFuzzer*: Fuzzing the Radboud Osiris web app

We would now like to test one of our fuzzers in a realistic scenario, on a real-world application. After some thought, we decided that the Osiris web app[16] could be a good target, as it is related to the Radboud University, and as it uses a JSON API. It is also a security-critical application, as it contains, among other things, grades and course registrations of students. We also learned about the Open Education API[17], which is a publicly available OpenAPI specification for applications in the education sector, and we hoped that the Osiris web app used this API, so that we would have an OpenAPI specification ready to use for our *OpenAPIFuzzer*. This, however, did not turn out to be the case. In the web app, we focus on the results page[18], as we identified that this paged used the Osiris JSON API in a straightforward manner. If we succeed in fuzzing this web app, we can in theory fuzz every

[16] `https://ru.osiris-student.nl`
[17] `https://openonderwijsapi.nl/`
[18] `https://ru.osiris-student.nl/#/resultaten`

other web app that uses a JSON API similarly.

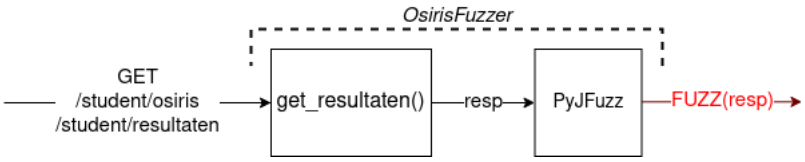


Figure 3.8: An overview of the control flow of *OsirisFuzzer*.

As Osiris does not have an OpenAPI specification, we can not use our *OpenAPIFuzzer* from Section 3.5. Therefore, we had to tailor our basic fuzzer to make it work for the Osiris web app. We call the resulting fuzzer "*OsirisFuzzer*". We give an overview of the control flow in Figure 3.8, and its code can be found in Appendix B.3. This fuzzer is essentially the same as *BasicFuzzer*, except for two small adjustments:

- *OsirisFuzzer* does not include "strong fuzzing": the fuzzing of the JSON structure. We decided to exclude this, as this did not give us interesting results in practice.
- The configured PyJFuzz techniques also include options 0 and 4. These techniques include XSS payloads, which seemed interesting for us to include while fuzzing a web app.
- We run Flask with `ssl_context='adhoc'`, which allows us to fuzz web apps that use TLS.

One issue that remains is that we have to make the Osiris web app connect to our *OsirisFuzzer* instead of the actual JSON API. We were able to identify multiple possible methods:

1. A proxy can be used to only redirect specific requests to `https://ru.osiris-student.nl` to our *OsirisFuzzer* that match a path of a JSON API endpoint, as illustrated in Figure 3.9.
2. A proxy can redirect all requests to `https://ru.osiris-student.nl` to our *OsirisFuzzer*, but this fuzzer requests other data, like CSS files and anything else that we do not want to fuzz, from the server itself, as illustrated in Figure 3.10.
3. We can run a modified copy of the Osiris web app that replaces specific API requests with requests to *OsirisFuzzer*, as illustrated in Figure 3.11.

We decided that the third method is the easiest and most practical, as it does not require setting up a proxy, and we figured that modifying a copy of

the Osiris web app is a relatively easy task. This did not turn out to be the case. In the end, method 1 is the only setup we were able to get working. For the attempts at the other methods, see Section 3.7.2.

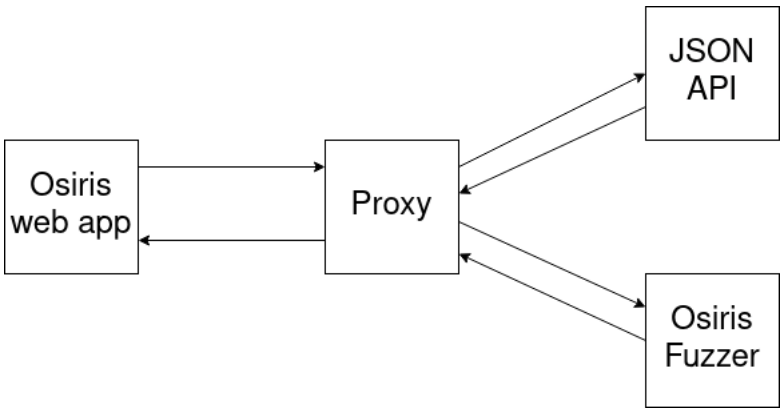


Figure 3.9: Method 1: Proxy redirects specific requests to *OsirisFuzzer*.

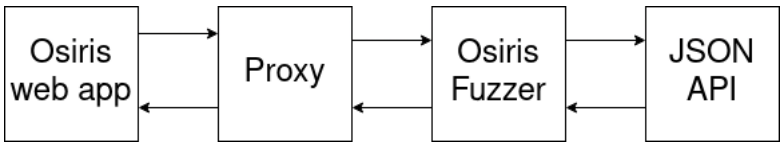


Figure 3.10: Method 2: *OsirisFuzzer* requests non-fuzzed content from the server.

For method 1, we were not able to find an easy way to configure a proxy like Burp Suite[19] to only redirect requests to our *OsirisFuzzer* that matched a certain URL pattern. We decided to look into creating our own Burp Suite extension to be able to do exactly this, because Burp Suite is the proxy we are most familiar with.

After writing the Burp Suite extension[20], see Appendix B.3.1, we were able to successfully fuzz the Osiris web app. The extension matches requests that come from `https://ru.osiris-student.nl`. For these requests, it checks if the URL starts with a target path. In our case, we are fuzzing the results page of Osiris, which has a path of `/student/osiris/student/resultaten`. If the URL starts with this path, we change the host and port of the request to that of our *OsirisFuzzer*, to forward it to this fuzzer.

The extension also replaces the HTTP version to `HTTP/1.1`, as Flask does not seem to support newer HTTP versions. This is a quick and easy solution

[19] `https://portswigger.net/burp`

[20] Our extension was based on an example given by PortSwigger: `https://github.com/PortSwigger/example-traffic-redirector/blob/master/python/TrafficRedirector.py`

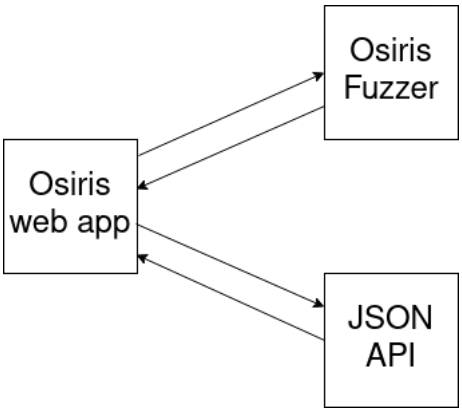


Figure 3.11: Method 3: Modified web app sends specific requests to *Osiris-Fuzzer*.

to this problem, as long as the target web app does not use functionality of the newer HTTP versions. To provide support for newer HTTP versions, we recommend rewriting *OsirisFuzzer* in another Python framework like Quart[21] which does support these newer HTTP versions.

OsirisFuzzer still has some limitations:

- **Automation**: to fuzz the web app, we need to refresh the page for each test case. Ideally, *OsirisFuzzer* would refresh the page automatically. This can be done with software like Puppeteer[22] or Selenium[23].
- **Crash detection**: *OsirisFuzzer* can not detect a crash in the web app, we have to identify a crash ourselves. If we can specify what constitutes a crash, this could also possibly be automated with Puppeteer, Selenium or a similar application.

3.7.1 Results

We ran 50 test cases of *OsirisFuzzer* within 8.5 minutes using the setup shown in Figure 3.12, which is 10.2 seconds per test case on average. Approximately a half of the time was used to save responses in Burp Suite, which we wanted to use for tracking crash-causing payloads. It is worth noting that this time per test case can be significantly reduced by automating the refreshing of the browser and saving the fuzzed responses automatically. Also, each test case fuzzes 25 independent JSON objects represented on the web app as independent list items, so we actually fuzzed $50 \cdot 25 = 1250$ list

[21] `https://quart.palletsprojects.com/en/latest/`
[22] `https://pptr.dev/`
[23] `https://www.selenium.dev/`

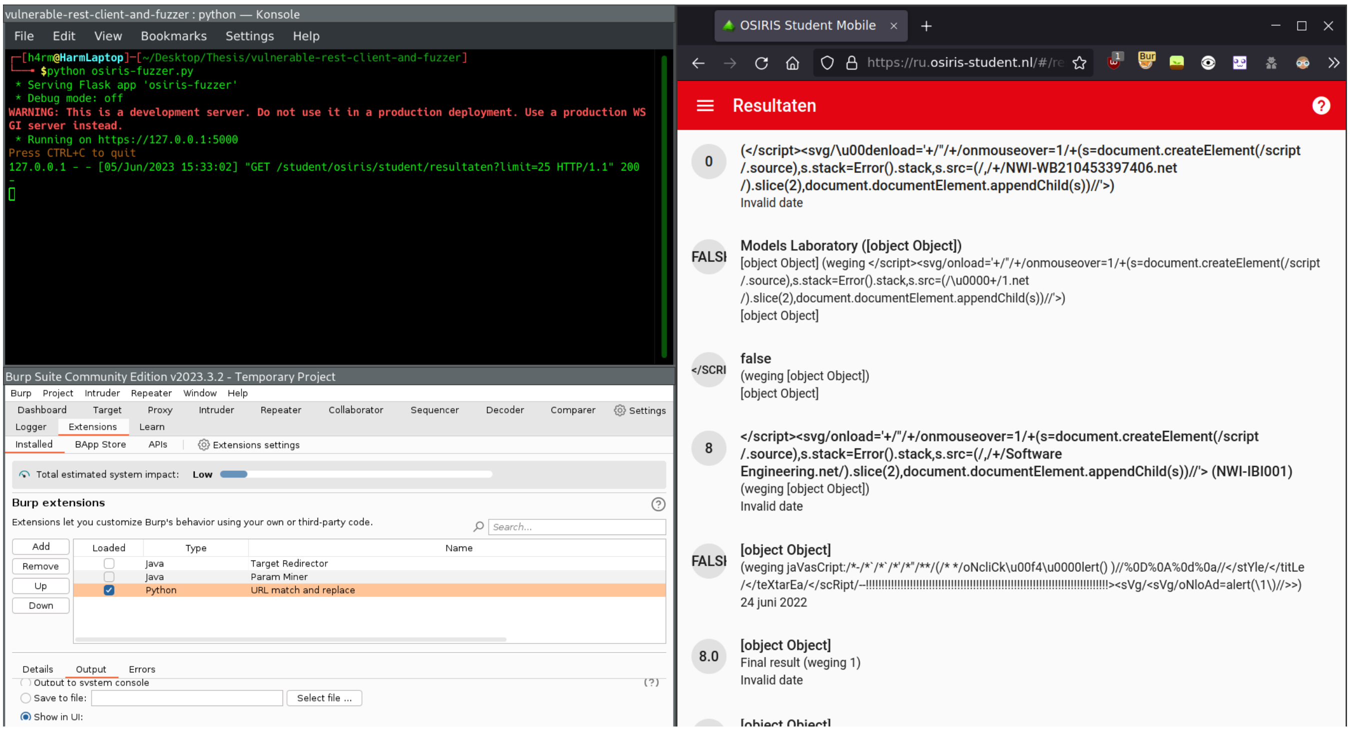


Figure 3.12: Fuzzing the Osiris web app. On the left you can see our *Osiris-Fuzzer* and Burp Suite running, on the right the Osiris web app with fuzzed results.

Test cases	Time (s)	Iterations per second	Bugs/crashes found
50	510	10.2	0

Table 3.2: Summarized results of fuzzing the Osiris web app

items. During the fuzzing, no crashes, errors, or bugs were found in the web app or in the browser console.

While fuzzing the Osiris web app, it gave us an error pop-up every fifth iteration, see Figure 3.13. We first thought that we found a bug, but seeing that we got an error exactly every fifth iteration, we were very skeptical. After some investigation, we found that the error was caused by Burp Suite in a different response than the response we fuzzed. Our guess is that this is caused by a bug in Burp Suite or by some form of DoS protection from Osiris.

Even though we were unable to find a bug or cause a crash, we have still created a setup in which it is possible to fuzz any web app that uses a JSON API. This is thus a useful proof-of-concept. It is also reassuring security-wise that we were not able to find bugs in the Osiris website.

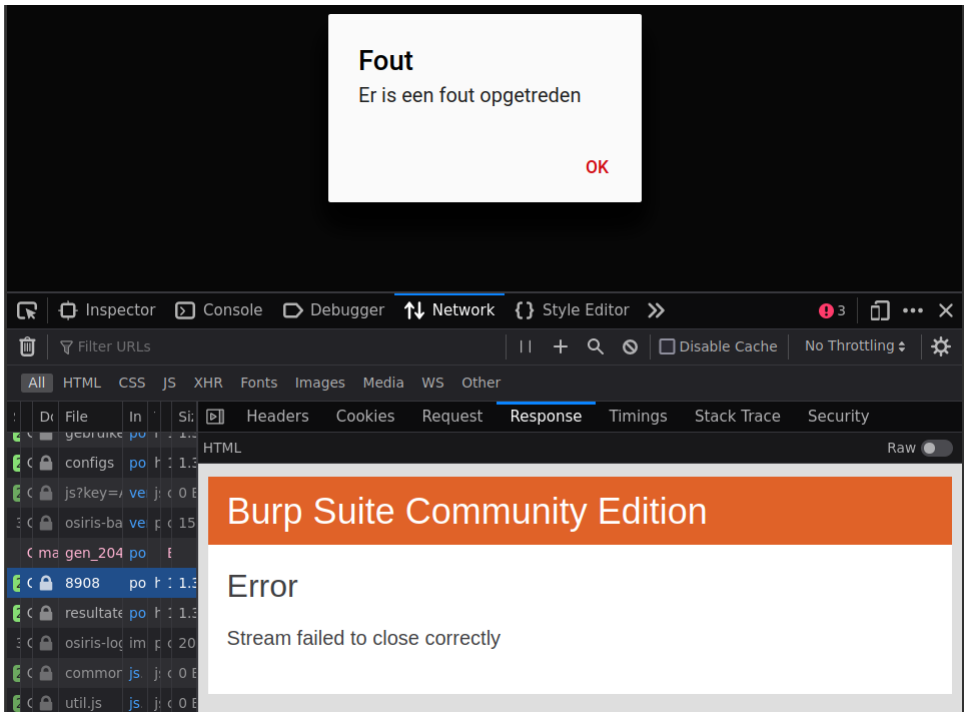


Figure 3.13: The error from Osiris and the Burp Suite error from a non-fuzzed response.

3.7.2 Other attempts

As mentioned before, we figured that method 3, modifying the web app to send specific requests to *OsirisFuzzer*, as illustrated in Figure 3.11, would be the easiest, as it does not require setting up a proxy. With this method, we would replace any mention to the endpoint we want to fuzz, with the URL of our *OsirisFuzzer*. This turned out to be a lot more challenging than we thought. Making a local copy of the web app did not work: it gave just a blank screen when opened. We also tried modifying the JavaScript of the web app in Firefox[24], which contained the requests to the JSON API that we wanted to replace with requests to *OsirisFuzzer*. However, this also broke the web app for unknown reasons.

We decided to take a look at method 2 next, in which all requests get forwarded to *OsirisFuzzer*, which requests other, non-fuzzed data like CSS files from the `https://ru.osiris-student.nl` server, as illustrated in Figure 3.10. We programmed *OsirisFuzzer* to send a request to the Osiris website with the same headers and cookies for content that we didn't want to fuzz. Unfortunately, this also did not work in the end, and the web app would

[24]This is a relatively new feature in Firefox, see `https://www.mozilla.org/en-US/firefox/113.0/releasenotes/`

not finish loading for unknown reasons.

3.7.3 Radboud Osiris Android app attempt

Our initial plan was to fuzz the Radboud Osiris Android app instead of the web app, but later on we decided to focus on the web app, as this allowed for an easier technical setup. We did however attempt to configure the Android app to use the Burp Suite proxy.

We emulated the Android operating system and installed the Radboud Osiris app on this emulated device. Sadly, we quickly ran into an issue: the app makes use of certificate pinning [25]. With this technique, the app is embedded with the public certificate of the Osiris server, which is used to encrypt all the HTTP traffic. This makes it challenging to intercept this traffic or to impersonate the Osiris server, which is required for using our *OsirisFuzzer*. We decided to stop here and to focus on the web app. This does not have a problem with certificate pinning, as it is not implemented in the Osiris web app, as HTTP Public Key Pinning is obsolete [25]. We did come up with some possible methods to bypass certificate pinning in the Android app:

1. Replacing the certificate in the bytecode of the app with our own certificate.
2. Removing the certificate check in the bytecode of the app altogether.
3. Decompiling the app and replacing or removing the certificate in the decompiled code.

Looking into fuzzing Android apps could be interesting future work, see Chapter 5.

3.8 *ProxyFuzzer*

One crucial part for each of our JSON API client fuzzers so far has been how we get the JSON example payload for PyJFuzz to fuzz. *BasicFuzzer* from Section 3.4 used a sample JSON schema from the `item` object that we hard coded in *BasicFuzzer*. *OpenAPIFuzzer* from Section 3.5 used an OpenAPI specification to generate example JSON data for PyJFuzz. *OsirisFuzzer* from Section 3.7 also contained a hard coded JSON example for PyJFuzz, just like *BasicFuzzer*.

What would happen if we fetch this JSON example for PyJFuzz from the JSON API server itself? We could forward a request to the JSON API server, intercept its response, pass it to PyJFuzz and send the fuzzed response to the JSON API client, as shown in Figure 3.14. If we succeed in doing this, we create a JSON API client fuzzer that in theory works on

every JSON API client, without the need for an OpenAPI specification or for hard coding a JSON example for PyJFuzz. This is the main idea behind our *ProxyFuzzer*, which we will discuss in this section. We believe that *ProxyFuzzer* is the best fit for most use cases, as it in theory works on any JSON API client that can use mitmproxy, and does not require an OpenAPI specification.

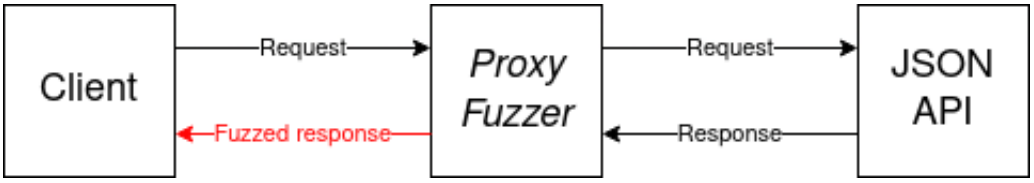


Figure 3.14: *ProxyFuzzer* would act as a proxy between the client and the server.

It is important to note that *ProxyFuzzer* is not suited for fuzzing responses of requests that are considered destructive. One could for example want to fuzz the JSON response from a request to a `/account/delete` endpoint. Even though the response of this request could be interesting to fuzz, *ProxyFuzzer* would forward this request to the actual JSON API server, which would delete your account. If you want to fuzz the response of this `/account/delete` endpoint, *OpenAPIFuzzer* or a tailored fuzzer like *Osiris-Fuzzer* would suit this better.

We can avoid using a Flask web server by writing an extension for a proxy, which reduces complexity. We tried writing an extension for Burp Suite, because this is the proxy we are most familiar with, but we did not get this to work, see Section 3.8.1. We ended up writing an add-on for mitmproxy[25], another HTTP(S) proxy. We chose for mitmproxy because it is written in Python, and we had worked with it before. For an overview of the control flow, see Figure 3.15. The code of *ProxyFuzzer* can be found in Appendix B.4.
The add-on first defines four new options:

1. `fuzzhost`: the host from which the responses will be fuzzed.
2. `fuzzpaths`: a comma-separated list of paths for which the responses will be fuzzed.
3. `fuzztechniques`: a comma-separated list of PyJFuzz fuzzing techniques to use. For example, 13 is a random character attack, while 0 and 4 contains XSS payloads.
4. `strongfuzz`: a boolean for the `strong_fuzz` parameter of PyJFuzz, which allows for fuzzing the JSON structure.

[25]https://mitmproxy.org/

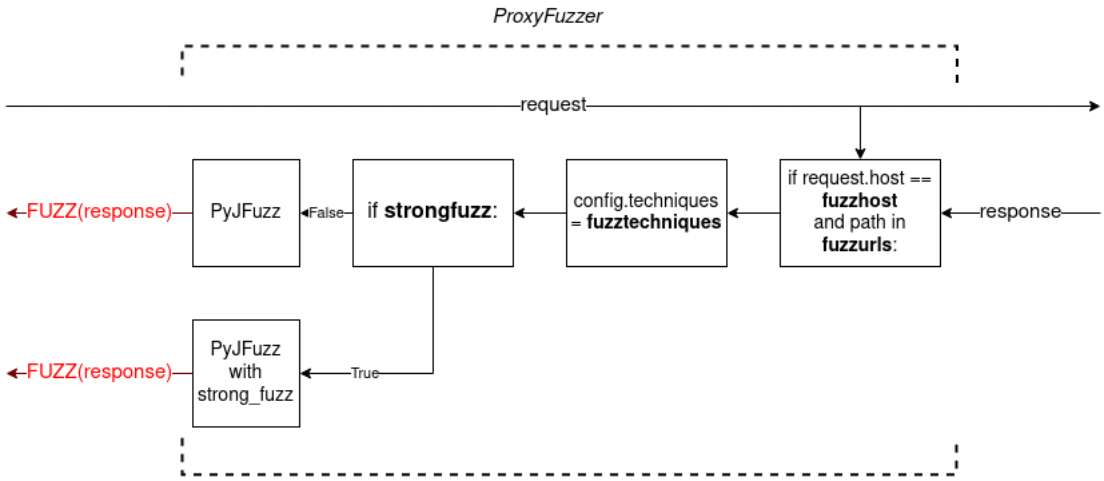


Figure 3.15: An overview of the control flow of *ProxyFuzzer*.

These options can be set in the command line, for example: `mitmproxy -s proxy-fuzzer.py --set fuzzhost=radboudinstituteof.pwning.nl`, or in the mitmproxy configuration file. The default values are set for fuzzing the results page of the Osiris web app.

The `response` function contains the actual fuzzing code, it gets called when mitmproxy is processing an HTTP(S) response. When the host of the corresponding request matches the `fuzzhost` option, and the path of the request starts with one of the `fuzzpaths`, we fuzz the response with PyJFuzz. We use the `strong_fuzz` parameter for PyJFuzz if the `strongfuzz` option is set, and the fuzzing techniques from the `fuzztechniques` option.

When we run our *ProxyFuzzer* on the results page of Osiris web app, we get similar results as in Section 3.7.1, as shown in Figure 3.16. Because of the similarity to *OsirisFuzzer*, we will not discuss the results extensively.

The same limitations in automation and crash detection as for *OsirisFuzzer* still exist, see Section 3.7. *ProxyFuzzer* could also be improved by adding support for regular expressions in the options `fuzzhost` and `fuzzpaths`. This would allow for better control over the matching hosts and URLs, as you could for example use wildcards.

3.8.1 Other attempts

Before we made our mitmproxy add-on *ProxyFuzzer*, we tried to create it in Flask and Burp Suite first.

We first made a Flask fuzzer that used our Burp Suite extension from Section 3.7 and the Python `requests` library to fetch responses from the

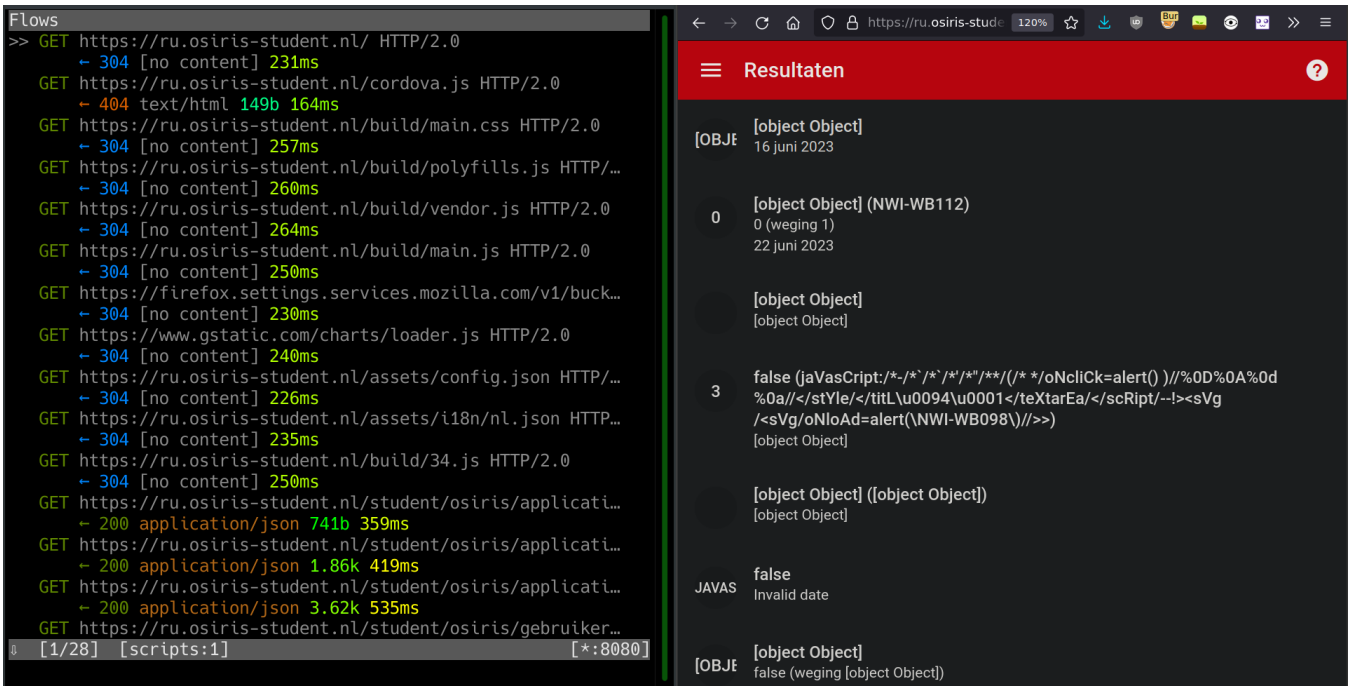


Figure 3.16: Fuzzing the Osiris web app. On the left you can see mitmproxy running with *ProxyFuzzer*, on the right the Osiris web app with fuzzed results.

JSON API server, as described in a Stack Overflow answer from user Evan[26]. This worked pretty easily, but still required the use of the Burp Suite extension. The resulting code, which is similar to the code of our final *ProxyFuzzer*, can be found in Appendix B.4.1. We prefer *ProxyFuzzer* over this solution, as it does not require the use of the Burp Suite extension.

We tried working around the Burp Suite extension, but ran into issues with lack of HTTP2 support by Flask. We used Quart[27] to add HTTP2 support, but ran into problems with TLS, because Quart does not support the `ssl_context='adhoc'` parameter that we use in Flask. Another issue we ran into was that Osiris kept redirecting us to `https://engine.surfconext.nl` off of our Flask fuzzer for authentication purposes. When we tried to detect redirects in HTTP traffic by looking for the Location header, we did not find any. We guess that Osiris uses JavaScript redirects instead of HTTP redirects. We could search for these redirects in the source code and try to replace them, but we figured that there should be an easier method.

Before using mitmproxy, we tried writing an extension for Burp Suite, because we were more familiar with this proxy. However, we were unable to get the PyJFuzz library working within Burp Suite. Burp Suite uses a different PATH variable for locating Python modules, but even when we

[26] `https://stackoverflow.com/a/36601467`
[27] `https://quart.palletsprojects.com/en/latest/`

changed this to the exact same PATH variable of our other fuzzers, there were errors when loading PyJFuzz, which we were unable to understand and resolve.

Chapter 4

Related Work

This chapter discusses existing research on JSON API fuzzing.

An extensive search online gives no results on REST or JSON API *client* fuzzing specifically. This is a sign that this thesis fills a knowledge gap in this area. There does exist quite a lot of literature on REST API server fuzzing, which is also applicable to JSON API servers, as explained in Section 2.1.1. This will be the main focus of this chapter.

There exist numerous REST API server fuzzers, too many to all mention in this thesis. We will take a look at five of them, in arbitrary order. We chose to look at RESTler and EvoMaster because of their popularity: their corresponding papers have over a hundred citations. Pythia and foREST both claim to outperform at least one of these popular fuzzers, so we discuss them as well. Finally, CATS was mentioned by our supervisor, and is one of the few fuzzers with OpenAPI parsing capabilities. For a comprehensive overview of these five fuzzers and their main features, see Table 4.1.

1. **RESTler** [9] is an open-source grammar-based blackbox REST API server fuzzer that can generate a grammar from an OpenAPI specification and fuzz accordingly, see Figure 4.1. The main feature that distinguishes RESTler from other REST API server fuzzers is that it is the first stateful REST API server fuzzer. In RESTler, each test case is defined as a sequence of requests and responses. It generates tests in two main ways: by inferring producer-consumer dependencies, that some request should be executed after another certain request, and by analyzing dynamic feedback from responses to for example avoid unnecessary request sequences in the future. Another feature of RESTler is its bug classification: it uses so-called buckets to avoid duplicates by merging bugs that end in the same sequence of requests, while also saving replay logs that can be used to reproduce the bug. RESTler has been successfully used to find numerous bugs in GitLab, Microsoft Azure and Office 365.

Fuzzer	Year	Open-source	Basis	Type	OpenAPI	Comment
RESTler	2018	✓	Grammar-based	Blackbox	✓	First stateful REST API fuzzer
CATS	2020	✓	Grammar-based	Blackbox	✓	Multiple independent fuzzers
EvoMaster	2018	✓	Search-based	Whitebox	✓	Generates JUnit tests
Pythia	2020	X	Grammar-based	Greybox	X	Coverage-guided feedback
foREST	2022	X	Tree-based	Blackbox	✓	Novel tree-based approach

Table 4.1: An overview of the discussed REST API server fuzzers and their main features.

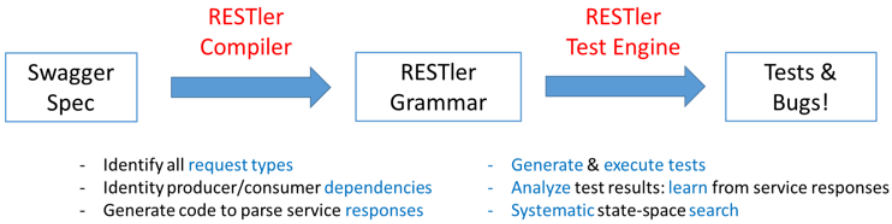


Figure 4.1: Diagram of RESTler's architecture[1]. With "Swagger Spec", the OpenAPI specification is meant. The "RESTler Test Engine" is the actual fuzzer.

2. **CATS**[2] is an open-source grammar-based blackbox REST API server fuzzer. Short for Contract Auto-generated Tests for Swagger, it is similar to RESTler in the sense that it can fuzz a REST API based on its OpenAPI specification. CATS consists of multiple independent fuzzers that test for specific scenarios. CATS will run this scenario and match the result with its expectations.

3. **EvoMaster** [7] is an open-source search-based whitebox REST and GraphQL API server fuzzer. Just like RESTler and CATS, EvoMaster can fuzz a REST API based on its OpenAPI specification. One of its main features is that EvoMaster is able to do whitebox fuzzing (as well as blackbox fuzzing), which distinguishes it from other, commonly blackbox REST API server fuzzers like the ones we mention in this chapter. EvoMaster can automatically generate JUnit tests while fuzzing for bugs at the same time. To generate these tests, it uses evolutionary algorithms, an artificial intelligence technique, in particular the MIO algorithm [6]. This makes it a search-based REST API server fuzzer. Moreover, EvoMaster has been successfully applied to find multiple bugs in open-source projects. RESTler and EvoMas-

[1]Taken from `https://raw.githubusercontent.com/microsoft/restler-fuzzer/main/docs/user-guide/RESTler-arch.png`

[2]`https://endava.github.io/cats/`

ter, among others[3], have been compared against each other [29]. The results concluded that EvoMaster has a better average line coverage overall.

4. **Pythia** [8] is a grammar-based greybox REST API server fuzzer that uses coverage-guided feedback and a learning-based mutation strategy for stateful fuzzing. It does not use an OpenAPI specification, but requires seed inputs to generate its test cases. The coverage-guided feedback helps with prioritizing test cases that are more likely to find bugs, while the learning-based mutation strategy allows Pythia to generate grammatically valid test cases. Atlidakis et al. claim that Pythia is able to outperform RESTler in terms of code coverage and new bugs found. Pythia has been successfully used to find numerous bugs, but is not publicly available or open-source.

5. **foREST** [18] is a tree-based blackbox REST API server fuzzer. According to their own research, existing REST API fuzzers are generally based on classic API-dependency graphs. However, these dependencies are inefficient for REST services because of the explosion of dependencies among APIs. To improve on this, foREST uses a novel tree-based approach, which largely improves the efficiency of the fuzzing. Lin and al. claim that foREST is able to reach better code coverage than EvoMaster and RESTler by using this approach, however, no comparison with Pythia or CATS has been made. foREST generates this API tree according to an OpenAPI specification. By using foREST, multiple previously unknown bugs have been discovered. Unfortunately, foREST is not publicly available or open-source.

It is worth mentioning that, although these fuzzers describe themselves as REST API (server) fuzzers, some of them could be better described as JSON API fuzzers, as the APIs they fuzz do not adhere to the definition of a REST API [12]:

1. **RESTler** does not support other resource representations than JSON[4], so it can only fuzz JSON APIs.

2. **CATS** also does not support other resource representations than JSON. In fact, it contains a specific fuzzer that verifies that an API endpoint does not accept XML as its Content-Type[5]. It argues that endpoints should avoid the XML Content-Type[6].

[3] bBOXRT [16], RestCT [26], RESTest [19], RestTestGen [24] & Schemathesis [14]

[4] `https://github.com/microsoft/restler-fuzzer/issues/335`

[5] `https://endava.github.io/cats/docs/fuzzers/linters/xml-content-type`

[6] `https://github.com/Endava/cats/blob/85947d0e316b46e926c371638ca644d6dbdb703c/src/main/java/com/endava/cats/fuzzer/contract/XmlContentTypeContractInfoFuzzer.java#L24`

3. **EvoMaster** is not just limited to JSON APIs, but can also fuzz GraphQL APIs and supports different resource representations like XML and YAML[7]. Because of this support for other resource representations, we view it as an actual REST API fuzzer, but it is certainly not limited to only REST APIs.

4. **Pythia** generates test cases based on seed input [8], which allows it to generate test cases in any resource representation. This allows it to fuzz any REST API, and thus makes it a real REST API fuzzer.

5. **foREST** generates test cases based on an OpenAPI specification, but does not put limitations on its Content-Type [18]. This allows foREST to fuzz any resource representation and thus any REST API, as long as an OpenAPI specification is provided.

[7] `https://github.com/EMResearch/EvoMaster/blob/master/docs/blackbox.md`

Chapter 5

Future Work

Our JSON API client fuzzers, *BasicFuzzer* from Section 3.4, *OpenAPIFuzzer* from Section 3.5, *OsirisFuzzer* from Section 3.7, and *ProxyFuzzer* from Section 3.8, could still be improved in a number of ways:

- One could add HTTP response code and header fuzzing to these fuzzers. This would allow the fuzzers to test for bugs in JSON API clients that use these HTTP fields, like our vulnerable client from Section 3.3.
- Testing for non-crashing bugs in JSON API clients might also be possible, although we are not sure how this could be done.
- Looking into a different JSON fuzzer from PyJFuzz could also improve the fuzzers. PyJFuzz worked fine for our purposes, but we do not know if there exist useful payloads it can not create. It is possible that a different JSON fuzzer, like Snodge as mentioned in Section 3.2, could create other payloads that result in finding more bugs. It is also possible to create an entirely new JSON fuzzer from scratch.
- In the case that the JSON API client is a web app, like the Osiris web app, one could use software like Selenium or Puppeteer to make the JSON API client automatically send requests to the fuzzer, or to add crash detection, as described in Section 3.7.
- *ProxyFuzzer* from Section 3.8 could be improved by adding support for regular expressions in the options `fuzzhost` and `fuzzpaths`, which would increase control by for example adding support for wildcards.

We made an attempt at fuzzing a 5G network function in Section 3.6. At the time, our *OpenAPIFuzzer* did not work because the `openapi3-parser` library it used did not support the 'anyOf' keyword. This issue has recently been fixed[1]. It would be interesting to see if our *OpenAPIFuzzer* would now

[1] `https://github.com/manchenkoff/openapi3-parser/issues/5`

be able to successfully parse an OpenAPI specification of a network function. We can also try to fuzz a 5G network function with our *ProxyFuzzer*, which we created later in our research.

It would also be interesting to see if different JSON API clients could be fuzzed, like other parts of the Osiris web app, other web apps, Android apps and desktop applications. Our *ProxyFuzzer* should theoretically work on web app JSON API clients, so it would be interesting to see if we could find bugs in these other web apps. Even though we were unable to get the technical setup for Android apps working in our research, it may very well still be possible using one of the potential solutions we provided in Section 3.7.3: by replacing or removing the certificate in the bytecode, or replacing it after decompiling the app. We did not look at desktop applications during this research, but these can also be JSON API clients. Especially fuzzing a JSON client that is written in C or C++ would be interesting, as programs in these languages are more prone to vulnerabilities like buffer overflows, because they allow accessing and overwriting data in any part of memory and do not check the bounds of an array [1].

Chapter 6

Conclusions

In this chapter, we present all our conclusions in Section 6.1. We also discuss the choices we made and reflect on them in Section 6.2.

6.1 Conclusions

We concluded that there exist no fuzzers for JSON API clients yet, as opposed to JSON API servers, and there exists no literature about them. Existing REST API fuzzers[1] like RESTler [9] are only able to fuzz JSON API servers, not clients. We speculate that this is caused by less interest in the security of JSON API clients and some difficulties that come with fuzzing of JSON API clients in Section 2.2. Fuzzing JSON API clients is namely fundamentally different from fuzzing JSON API server, as there is no straight-forward way to detect crashes or to automate the fuzzing process when fuzzing JSON API clients, see Section 2.2.1. JSON APIs are also fundamentally different from REST APIs [12], because REST APIs can use different resource representations from JSON and have to conform to more constraints, like HATEOAS, see Section 2.1.1.

We have successfully answered the research question, *How can we test for bugs in a JSON API client using fuzzing?*, in the following steps:

1. We created *BasicFuzzer*, a JSON API client fuzzer that consist of a Flask[2] web server that returns fuzzed JSON output created by the PyJFuzz library[3] in Section 3.4.

2. We made a vulnerable client in Section 3.3 and put three types of bugs in this vulnerable client: a JSON parse without validating the structure, a float round without verifying the datatype, and a planted

[1] For more information about existing REST API server fuzzers, see Chapter 4

[2] `https://flask.palletsprojects.com/en/2.2.x/`

[3] `https://github.com/mseclab/PyJFuzz`

error for a response code other than 200. We found the first two of these bugs using our *BasicFuzzer* in Section 3.4.

3. We created *OpenAPIFuzzer* that can fuzz a JSON API client when provided with its OpenAPI specification in Section 3.5. We were unable to use *OpenAPIFuzzer* to fuzz 5G network functions in Section 3.6, because of limitations of the `openapi3-parser` Python library.

4. We were able to fuzz the results page of the Osiris web app with *OsirisFuzzer* by using the Burp Suite[4] proxy with a custom self-made extension, see Section 3.7. We did not find any bugs, which is reassuring security-wise.

5. We created *ProxyFuzzer*, an add-on script for mitmproxy, that can acts as a proxy and fuzz responses sent to the JSON API client. This in theory allows us to fuzz any JSON API client that can use mitmproxy.

	BasicFuzzer	***OpenAPIFuzzer***	***OsirisFuzzer***	***ProxyFuzzer***
Specific to single API	✓	X	✓	X
Requires OpenAPI specification	X	✓	X	X
TLS support	X	X	✓	✓
Fuzzes JSON structure	✓	✓	X	✓
Random character mutations	✓	✓	✓	✓
XSS payloads	X	X	✓	✓

Table 6.1: A comparison between our four self-made fuzzers.

For a comparison between our four fuzzers, *BasicFuzzer*, *OpenAPIFuzzer*, *OsirisFuzzer*, and *ProxyFuzzer*, see Table 6.1. We believe that *ProxyFuzzer* is the best fit for most use cases, as it in theory works on any JSON API client that can use mitmproxy, and does not require an OpenAPI specification.

We tried fuzzing the Radboud Osiris Android app, but we ran into problems due to certificate pinning in Section 3.7.3.

6.2 Evaluation of choices

6.2.1 Fuzzer requirements choices

When choosing requirements for our fuzzer, see Section 3.1, we made the following choices:

- To minimize complexity, the fuzzing will be done over HTTP instead of HTTPS. TLS support would be nice in practice, but is not a key part of a JSON API, which allows us to leave it out.

[4] `https://portswigger.net/burp`

- Although one of the requirements is the ability to fuzz the HTTP header, we choose to omit this. Most of the application logic of JSON API clients will depend on the data being returned by the server, not HTTP headers.

Overall, these choices turned out to be fine. Fuzzing over only HTTP simplified things in the beginning, but eventually we were able to extend functionality to HTTPS quite easily in Section 3.7, as the Osiris web app does not use certificate pinning.

6.2.2 Implementation choices

We made the following choices regarding the implementation of our fuzzers:

- We chose to use OpenAPI to specify the JSON API, as 5G uses OpenAPI to specify its API, and fuzzing 5G was one of our stretch goals.
- We chose Python to program our vulnerable client and fuzzer, as Python is the programming language most familiar to us.
- We chose to use Flask with PyJFuzz for our fuzzer implementation, because JSON API client fuzzers do not exist and using a JSON fuzzing library like PyJFuzz is the closest alternative. Moreover, using a fuzzing framework would be overly complex, as discussed in more detail in Section 3.2. We chose PyJFuzz over other JSON fuzzers, because of its popularity, its XSS payloads, and its capability to fuzz the JSON structure.
- We chose to use the `openapi3-parser` Python library[5] for parsing OpenAPI specifications to generate JSON examples for PyJFuzz, because this seemed like a doable programming task. Also, we thought that it was better practice to only use Python, rather than combining it with Java code, which was the alternative we found in Section 3.5.
- We chose to use Burp Suite as our proxy in Section 3.7, because this was the proxy most familiar to us.
- We used mitmproxy for our *ProxyFuzzer* in Section 3.8, because it is written in Python, we had used it in the past before, and Burp Suite did not work.

Working with Python and OpenAPI turned out to be pleasant, OpenAPI allowed us to easily mock the JSON API server. Using Flask with PyJFuzz turned out to be a very flexible approach, as we could easily expand or tailor our fuzzer. PyJFuzz does come with its own web server, but we

[5] `https://pypi.org/project/openapi3-parser/`

did not get this to work, and with Flask we had better control over the setup. It is possible that using a different JSON fuzzing library would have been able to find bugs in the Osiris web app, see Section 3.7, but this is merely speculation. We are also content with our choices for Burp Suite and mitmproxy, as these allowed us to create *OsirisFuzzer* and *ProxyFuzzer* respectively.

Using the `openapi3-parser` Python library did not turn out to be a great choice. We were able to create our *OpenAPIFuzzer* that could parse OpenAPI specifications in Section 3.5, but it did not support the use of the 'anyOf' keyword at the time[6]. It is possible that using a different OpenAPI parsing library would have allowed us to parse specifications with the 'anyOf' keyword, or that using the Java Swagger Inflector library as described in Section 3.5 would have allowed this.

6.2.3 Case study choices

We chose to first fuzz a self-made vulnerable client in Section 3.3, because this would keep the technical setup simpler and easier to control. This turned out to be a great choice: it was nice to have full control over the setup, and extending the fuzzer to fuzz other JSON API clients was just a small task. We were able to find the first two bugs of our vulnerable client with our *BasicFuzzer* using different payloads, so we consider these bugs to be well-chosen. We did not attempt to find the third bug, an HTTP response code parsing bug, as we decided not to fuzz the HTTP header and response codes. In hindsight, we could have also omitted this bug in our vulnerable client.

We chose to fuzz the Osiris web app in Section 3.7, because it is an application related to the Radboud University that uses a JSON API, and is security-critical, as it contains, among other things, grades and course registrations of students. Also, fuzzing a web app was easier than fuzzing for example an Android app, as we did not have to worry about certificate pinning as described in Section 3.7.3. Fuzzing a web app was definitely easier than fuzzing an Android app, but fuzzing the Osiris web app turned out to be quite challenging.

When trying to make the Osiris web app connect to our *OsirisFuzzer* in Section 3.7, we identified three possible methods:

1. A proxy could be used to only redirect requests to *OsirisFuzzer* that match a path of a JSON API endpoint.

2. A proxy could redirect requests to *OsirisFuzzer*, but our fuzzer requests other data that we do not want to fuzz from the server itself.

[6]This issue has been fixed: `https://github.com/manchenkoff/openapi3-parser/issues/5`. Seeing if this resolves our issues is interesting future work, see Chapter 5.

3. We could run a modified copy of the web app that replaces API requests with requests to *OsirisFuzzer*.

We first tried to make methods 2 and 3 work. These did not turn out to work, for unknown reasons. In hindsight, it would have been more efficient if we started with method 1 first, but we do not know how we could have been able to tell this, as we were unable to tell that methods 2 and 3 would not work.

Bibliography

[1] Buffer overflows. https://www.owasp.org/index.php/Buffer_Overflows, archived 2016-08-29 at the Wayback Machine: https://web.archive.org/web/20160829122543/https://www.owasp.org/index.php/Buffer_Overflows.

[2] Client fuzzing tutorial - Kitty 0.7.1 documentation. https://kitty.readthedocs.io/en/latest/tutorials/client_fuzzing.html.

[3] Getting started - OpenAPI documentation. https://oai.github.io/Documentation/start-here.html.

[4] Introduction - Kitty 0.7.1 documentation. https://kitty.readthedocs.io/en/latest/base_introduction.html.

[5] Structure of an OpenAPI document - OpenAPI documentation. https://oai.github.io/Documentation/specification-structure.html.

[6] Andrea Arcuri. Many independent objective (mio) algorithm for test suite generation. In *Search Based Software Engineering: 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings 9*, pages 3–17. Springer, 2017. DOI: 10.1007/978-3-319-66299-2_1.

[7] Andrea Arcuri. Evomaster: Evolutionary multi-context automated system test generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 394–397. IEEE, 2018. DOI: 10.1109/ICST.2018.00046.

[8] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. Pythia: Grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. *CoRR*, abs/2005.11498, 2020. DOI: 10.48550/arXiv.2005.11498.

[9] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. RESTler: Stateful REST API fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019. DOI: 10.1109/ICSE.2019.00083.

[10] R. Fielding and J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. RFC 7231, RFC Editor, June 2014. https://www.rfc-editor.org/rfc/rfc7231#section-4.

[11] Roy Thomas Fielding. *Chapter 5: Representational State Transfer (REST)*. PhD thesis, University of California, 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[12] Roy Thomas Fielding. REST APIs must be hypertext-driven, Oct 2008. https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[13] Patrice Godefroid. *Fuzzing: Hack, art, and science*, volume 63. ACM New York, NY, USA, 2020. DOI: 10.1145/3363824.

[14] Zac Hatfield-Dodds and Dmitry Dygalo. Deriving semantics-aware fuzzers from web API schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 345–346, 2022. DOI: 10.1145/3510454.3528637.

[15] htmx. How did rest come to mean the opposite of rest?, Jul 2022. https://htmx.org/essays/how-did-rest-come-to-mean-the-opposite-of-rest/.

[16] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. *A black box tool for robustness testing of REST services*, volume 9. IEEE, 2021. DOI: 10.1109/ACCESS.2021.3056505.

[17] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. In *Cybersecurity*, volume 1, pages 1–13. SpringerOpen, 2018. DOI: 10.1186/s42400-018-0002-y.

[18] Jiaxian Lin, Tianyu Li, Yang Chen, Guangsheng Wei, Jiadong Lin, Sen Zhang, and Hui Xu. foREST: A tree-based approach for fuzzing RESTful APIs. *arXiv preprint arXiv:2203.02906*, 2022. DOI: 10.48550/arXiv.2203.02906.

[19] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. RESTest: Black-box constraint-based testing of RESTful web APIs. In *Service-Oriented Computing: 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings 18*, pages 459–475. Springer, 2020. DOI: 10.1007/978-3-030-65310-1_33.

[20] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. *Fuzzing: the state of the art*. DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012. Technical report. https://apps.dtic.mil/sti/citations/ADA558209.

[21] Barton P Miller, Lars Fredriksen, and Bryan So. *An empirical study of the reliability of UNIX utilities*, volume 33. ACM New York, NY, USA, 1990.

[22] Lionel Morand. OpenAPIs for the service-based architecture. `https://www.3gpp.org/technologies/openapis-for-the-service-based-architecture`, 2022.

[23] R. Shirey. Internet security glossary. RFC 2828, RFC Editor, May 2000. `https://www.rfc-editor.org/rfc/rfc2828#section-3`.

[24] Emanuele Viglianisi, Michael Dallago, and Mariano 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. IEEE, 2020. DOI: 10.1109/ICST46399.2020.00024.

[25] Jeffery Walton, John Steven, Jim Manico, Kevin Wall, and Ricardo Iramar. Certificate and public key pinning. `https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning`.

[26] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. Combinatorial testing of RESTful APIs. In *Proceedings of the 44th International Conference on Software Engineering*, pages 426–437, 2022. DOI: 10.1145/3510003.3510151.

[27] Michal Zalewski. American fuzzy lop. URL: `http://lcamtuf.coredump.cx/afl`.

[28] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2022. Retrieved 2022-08-06.

[29] Man Zhang and Andrea Arcuri. Open problems in fuzzing RESTful APIs: A comparison of tools. *arXiv preprint arXiv:2205.05325*, 2022. DOI: 10.48550/arXiv.2205.05325.

Appendix A

Code of the vulnerable client

This appendix includes an OpenAPI specification of our REST API in A.1 and the Python code of the vulnerable client in A.2. The OpenAPI specification is used as a description of our API and allows us to mock a server for testing purposes. For more information about OpenAPI, see Section 2.1. The vulnerable client is an example application that contains three planted bugs. For more information about the vulnerable client, see Section 3.3.

A.1 OpenAPI specification

```yaml
openapi: 3.0.0
info:
  description: |
    This is a sample warehouse server.  You can find
    out more about Swagger at
    [http://swagger.io](http://swagger.io) or on
    [irc.freenode.net, #swagger](http://swagger.io/irc/).
  version: "1.0.0"
  title: Webshop
servers:
  # Added by API Auto Mocking Plugin
  - description: SwaggerHub API Auto Mocking
    url: https://virtserver.swaggerhub.com/HARMROUKEMA/Webshop/1.0.0
tags:
  - name: item
    description: Everything about your items
paths:
  /item:
    post:
      tags:
        - item
```

```
    summary: Add a new item to the webshop
    operationId: addItem
    responses:
      '200':
        description: successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ApiResponse'
      '405':
        description: Invalid input
    requestBody:
      $ref: '#/components/requestBodies/Item'
'/item/{itemId}':
  get:
    tags:
      - item
    summary: Find item by ID
    description: Returns a single item
    operationId: getitemById
    parameters:
      - name: itemId
        in: path
        description: ID of item to return
        required: true
        schema:
          type: integer
          format: int64
    responses:
      '200':
        description: successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/item'
      '400':
        description: Invalid ID supplied
      '404':
        description: item not found
  delete:
    tags:
      - item
    summary: Deletes a item
    operationId: deleteitem
```

```yaml
      parameters:
        - name: itemId
          in: path
          description: item id to delete
          required: true
          schema:
            type: integer
            format: int64
      responses:
        '400':
          description: Invalid ID supplied
        '404':
          description: item not found
components:
  schemas:
    Category:
      type: object
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
          example: footwear
    Tag:
      type: object
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
          example: new
    item:
      type: object
      required:
        - name
      properties:
        id:
          type: integer
          format: int64
        category:
          $ref: '#/components/schemas/Category'
        name:
```

```yaml
        type: string
        example: Nike AF1
      price:
        type: number
        format: float
        example: 119.99
      tags:
        type: array
        items:
          $ref: '#/components/schemas/Tag'
      available:
        type: boolean
      comment:
        type: string
        nullable: true
      dateAdded:
        type: string
        format: date-time
  ApiResponse:
    type: object
    properties:
      code:
        type: integer
        format: int32
      type:
        type: string
      message:
        type: string
        nullable: true
requestBodies:
  Item:
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/item'
    description: item object that needs to be added to the warehouse
    required: true
```

A.2 Python code

```python
import requests
import json

SERVER_IP = '127.0.0.1'
SERVER_PORT = 5000
URL = f'http://{SERVER_IP}:{SERVER_PORT}'
# Overwrite URL to mocking server
# URL = 'http://virtserver.swaggerhub.com/HARMROUKEMA/Webshop/1.0.0'

s = requests.Session()

def test_get(id):
    # GET /item/1
    r = s.get(f"{URL}/item/{id}")
    print(r.text)
    # BUG 1: json.loads() crashes in case of invalid json
    item = json.loads(r.text)
    # BUG 2: no validation of data type before rounding
    price = round(item['price'])
    print(f"Price of item in euro: {price}")

def test_post():
    # POST /item
    item = {
      "id": 0,
      "category": {
        "id": 0,
        "name": "footwear"
      },
      "name": "Nike AF1",
      "price": 119.99,
      "tags": [
        {
          "id": 0,
          "name": "new"
        }
      ],
      "available": True,
      "comment": "string",
      "dateAdded": "2022-10-19T14:52:10.872Z"
    }
```

```
    r = s.post(URL + '/item', json=item)
    # BUG 3: Crash when the status code is not 200:
    if r.status_code != 200:
        print("ERROR: unexpected status code returned")
        exit()

for i in range(50):
    test_get(1)
```

Appendix B

Code of the fuzzers

This appendix contains the code of the JSON API client fuzzers, written in the Python Flask framework or as a mitmproxy add-on, using the JSON fuzzer PyJFuzz. For code explanation, see Section 3.4 for explanation of *BasicFuzzer*, Section 3.5 for *OpenAPIFuzzer*, Section 3.7 for *OsirisFuzzer* and Section 3.8 for *ProxyFuzzer*.

B.1 *BasicFuzzer*

This section includes the Python code of the first fuzzer we created, the *BasicFuzzer* discussed in Section 3.4.

```
from flask import Flask
from argparse import Namespace
from pyjfuzz.lib import *
import random

app = Flask(__name__)

@app.get('/item/<int:item_id>')
def get_item(item_id):
    # Example item for PyJFuzz
    item = {
      "id": item_id,
      "category": {
        "id": 0,
        "name": "footwear"
      },
      "name": "Nike AF1",
      "price": 119.99,
      "tags": [
```

```
            {
              "id": 0,
              "name": "new"
            }
          ],
          "available": True,
          "comment": "string",
          "dateAdded": "2022-10-19T14:52:10.872Z"
        }

        # We fuzz the JSON structure half of the time
        if random.randint(0,1):
            config = PJFConfiguration(Namespace(json=item,
            ↪  nologo=True, level=6, strong_fuzz=True))
        else:
            config = PJFConfiguration(Namespace(json=item,
            ↪  nologo=True, level=6))
        config.techniques = [13]
        fuzzer = PJFFactory(config)

        print(f"Returning fuzzed JSON:")
        print(fuzzer.fuzzed)
        print()
        return fuzzer.fuzzed

    if __name__ == '__main__':
        app.run()
```

B.2 *OpenAPIFuzzer*

Here we include the Python code of the *OpenAPIFuzzer*, discussed in Section 3.5.

```
from openapi_parser import parse
from flask import Flask
from flask import abort, request
from argparse import Namespace
from pyjfuzz.lib import *
import random
import json
import re

app = Flask(__name__)

filename = '../webshop.yaml'

print(f"Parsing {filename}...")
content = parse(filename)

urls = []
methods = []
jsons = []

def quote(s):
    return '"' + s + '"'

def parseSchemaToJson(schema):
    if schema.type.name == 'OBJECT':
        return '{' + ', '.join([parsePropToJson(prop) for prop
        ↪  in schema.properties]) + '}'
    elif schema.type.name == 'ARRAY':
        return '[' + parseSchemaToJson(schema.items) + ']'
    elif schema.type.name == 'INTEGER':
        return (str(schema.example) if schema.example else
        ↪  '0')
    elif schema.type.name == 'STRING':
        return (quote(schema.example) if schema.example else
        ↪  '"string"')
    elif schema.type.name == 'NUMBER':
        if schema.example:
            return str(schema.example)
```

```
        if schema.format.name == 'FLOAT' or schema.format.name
        ↪  == 'DOUBLE':
            return '3.7'
        else:
            return '0'
    elif schema.type.name == 'BOOLEAN':
        return (str(schema.example) if schema.example else
        ↪  'true')

def parsePropToJson(prop):
    if prop.schema.type.name == 'OBJECT':
        return quote(prop.name) + ': ' + '{' + ',
        ↪  '.join([parsePropToJson(prop) for prop in
        ↪  prop.schema.properties]) + '}'
    elif prop.schema.type.name == 'ARRAY':
        return quote(prop.name) + ': ' + '[' +
        ↪  parseSchemaToJson(prop.schema.items) + ']'
    elif prop.schema.type.name == 'INTEGER':
        return quote(prop.name) + ': ' +
        ↪  (str(prop.schema.example) if prop.schema.example
        ↪  else '0')
    elif prop.schema.type.name == 'STRING':
        return quote(prop.name) + ': ' +
        ↪  (quote(prop.schema.example) if prop.schema.example
        ↪  else '"string"')
    elif prop.schema.type.name == 'NUMBER':
        if prop.schema.example:
            return quote(prop.name) + ': ' +
            ↪  str(prop.schema.example)
        if prop.schema.format.name == 'FLOAT' or
        ↪  prop.schema.format.name == 'DOUBLE':
            return quote(prop.name) + ': ' + '3.7'
        else:
            return quote(prop.name) + ': ' + '0'
    elif prop.schema.type.name == 'BOOLEAN':
        return quote(prop.name) + ': ' +
        ↪  (str(pop.schema.example) if prop.schema.example
        ↪  else 'true')

def processUrls(urls):
    res = []
    for url in urls:
        url = url[1:]
        url = re.sub('{.*}', '.*', url)
```

```
        res.append(url)
    return res

# Loop over the parsed OpenAPI specification and parse the
↪  schemas to JSON
for path in content.paths:
    for operation in path.operations:
        for response in operation.responses:
            if response.content:
                for content in response.content:
                     if content.type.name == 'JSON':
                        urls.append(path.url)
                        methods.append(operation.method.name)

                         ↪  jsons.append(parseSchemaToJson(content.schema))

print(f"Successfully parsed {filename}")

urls = processUrls(urls)
print(f"URLs: {urls}")
print(f"JSONs: {jsons}")

@app.route('/<path:path>', methods = ['GET', 'POST', 'PUT',
↪  'DELETE', 'OPTIONS'])
def fuzz(path):
    for i in range(len(urls)):
        # We fuzz if we match a URL and the corresponding HTTP
        ↪  method
        if re.search(urls[i], path) and request.method ==
        ↪  methods[i]:
            payload = json.loads(jsons[i])
            # We fuzz the JSON structure half of the time
            if random.randint(0,1):
                config =
                ↪  PJFConfiguration(Namespace(json=payload,
                ↪  nologo=True, level=6, strong_fuzz=True))
            else:
                config =
                ↪  PJFConfiguration(Namespace(json=payload,
                ↪  nologo=True, level=6))
            config.techniques = [13]
            fuzzer = PJFFactory(config)
```

```
            print(f"{request.method} request at {path} -
            ↪  returning fuzzed JSON:")
            print(fuzzer.fuzzed)
            print()
            return fuzzer.fuzzed
    abort(404)

if __name__ == '__main__':
    app.run()
```

B.3 *OsirisFuzzer*

This is the code of *OsirisFuzzer*, discussed in Section 3.7. Some parts of the example response have been anonymized for privacy reasons, or removed for brevity.

```
from flask import Flask
from argparse import Namespace
from pyjfuzz.lib import *

app = Flask(__name__)

@app.get('/student/osiris/student/resultaten')
def get_resultaten():
    # Example response for PyJFuzz
    resp = {"items": [{"id_resultaat":"scto:1",
    ↪  "studentnummer":"1", "cursus":"NWI-WB098",
    ↪  "collegejaar":2022, "blok":"KW2",
    ↪  "periode_omschrijving":"Period 2",
    ↪  "startdatum_blok":"2022-11-06T23:00:00Z",
    ↪  "cursus_korte_naam":"Random Graphs",
    ↪  "toets_omschrijving":"Exam", "weging":"1",
    ↪  "resultaat":"10", "resultaat_omschrijving":"10",
    ↪  "score":"", "score_omschrijving":"", "voldoende":"J",
    ↪  "status":"", "toetsdatum":"2023-01-12T23:00:00Z",
    ↪  "docent":"", "onderwerp":"",
    ↪  "mutatiedatum":"2023-01-30T20:22:59Z",
    ↪  "id_cursus":"curs:152879",
    ↪  "id_geldend_resultaat":"sgre:1",
    ↪  "id_beoordelingsformulier":"null",
    ↪  "id_cursus_toetsins":"null",
    ↪  "id_toets_toetsins":"null", "soort_resultaat":"toets",
    ↪  "nieuw":"N", "bonustoets":"N"}], "hasMore":True,
    ↪  "limit":25, "offset":0, "count":25}

    config = PJFConfiguration(Namespace(json=resp,
    ↪  nologo=True, level=6))
    # We also include XSS techniques 0 and 4
    config.techniques = [0, 4, 13]
    fuzzer = PJFFactory(config)

    print(f"Returning fuzzed JSON:")
    print(fuzzer.fuzzed)
    print()
```

```
    return fuzzer.fuzzed

if __name__ == '__main__':
    app.run(ssl_context='adhoc')
```

B.3.1 Burp Suite extension

This subsection contains the code of the Burp Suite extension we wrote to forward certain requests to our *OsirisFuzzer*, see Section 3.7.

```
from burp import IBurpExtender
from burp import IHttpListener
from java.io import PrintWriter
from java.lang import RuntimeException

HOST_FROM = "ru.osiris-student.nl"
HOST_TO = "127.0.0.1"
PORT_TO = 5000
matching_URLs = [
    '/student/osiris/student/resultaten'
]

class BurpExtender(IBurpExtender, IHttpListener):

    #
    # implement IBurpExtender
    #

    def registerExtenderCallbacks(self, callbacks):
        # obtain an extension helpers object and set callbacks
        self._helpers = callbacks.getHelpers()
        self.callbacks = callbacks

        # set our extension name
        callbacks.setExtensionName("Replace host for matching
        ↪  URL")

        # register ourselves as an HTTP listener
        callbacks.registerHttpListener(self)

    #
    # implement IHttpListener
    #

    def processHttpMessage(self, toolFlag, messageIsRequest,
    ↪  messageInfo):
        # only process requests
        if not messageIsRequest:
            return
```

```
# obtain our output stream
stdout = PrintWriter(self.callbacks.getStdout(), True)

# get the HTTP service for the request
httpService = messageInfo.getHttpService()

# if the host is HOST_FROM, change it to HOST_TO
if (HOST_FROM == httpService.getHost()):
    # get the request of the HTTP server
    request =
    ↪  self._helpers.bytesToString(messageInfo.getRequest())
    first_line = request.split('\n')[0].split(' ')
    URL = first_line[1]

    # check if the path of the URL matches one of the
    ↪  matching_URLs
    for match in matching_URLs:
        if URL[:len(match)] == match:
            # change HTTP version to 1.1 to not
            ↪  confuse Flask
            first_line[2] = 'HTTP/1.1'
            first_line = ' '.join(first_line)
            new_request =
            ↪  self._helpers.stringToBytes('\n'.join([first_line]
            ↪  + request.split('\n')[1:]))
            messageInfo.setRequest(new_request)

            # change the host and port of the request
            messageInfo.setHttpService(
            ↪  self._helpers.buildHttpService(
            ↪  HOST_TO, PORT_TO,
            ↪  httpService.getProtocol()))
```

B.4 *ProxyFuzzer*

This section contains the Python code of *ProxyFuzzer*, an add-on script for mitmproxy. For an explanation of this code, see Section 3.8.

```
from argparse import Namespace
from pyjfuzz.lib import *
import random
import json
from mitmproxy import ctx, exceptions

class FuzzClient:
    def load(self, loader):
        loader.add_option(
            name = "fuzzhost",
            typespec = str,
            default = "ru.osiris-student.nl",
            help = "Set the host from which the responses will
            ↪  be fuzzed",
        )
        loader.add_option(
            name = "fuzzurls",
            typespec = str,
            default = "/student/osiris/student/resultaten",
            help = "Comma-separated list of URLs from which
            ↪  the responses will be fuzzed"
        )
        loader.add_option(
            name = "fuzztechniques",
            typespec = str,
            default = "0,4,13",
            help = "Comma-separated list of PyJFuzz fuzzing
            ↪  techniques to use",
        )
        loader.add_option(
            name = "strongfuzz",
            typespec = bool,
            default = False,
            help = "Use the strong_fuzz parameter of PyJFuzz
            ↪  half of the time, which includes fuzzing the
            ↪  JSON structure"
        )

    def response(self, flow):
```

```
request = flow.request
response = flow.response

if request.host == ctx.options.fuzzhost:
    for match in ctx.options.fuzzurls.split(','):
        if request.path[:len(match)] == match:
            ctx.log.info(f"Found matching URL:
            ↪ {request.host}{request.path}")
            try:
                payload =
                ↪ json.loads(response.content.decode())
            except:
                ctx.log.error(f"Error while parsing
                ↪ JSON")
                return

            if ctx.options.strongfuzz and
            ↪ random.randint(0,1):
                ctx.log.info(f"Fuzzing the JSON
                ↪ structure")
                config =
                ↪ PJFConfiguration(Namespace(json=payload,
                ↪ nologo=True, level=6,
                ↪ strong_fuzz=True))
            else:
                ctx.log.info(f"Not fuzzing the JSON
                ↪ structure")
                config =
                ↪ PJFConfiguration(Namespace(json=payload,
                ↪ nologo=True, level=6))

            fuzztechniqueserror =
            ↪ exceptions.OptionsError("Option
            ↪ 'fuzztechniques' needs to be a
            ↪ comma-separated list of integers
            ↪ between 0 and 14.")
            try:
                config.techniques = list(map(int,
                ↪ ctx.options.fuzztechniques.split(',')))
                for technique in config.techniques:
                    if technique < 0 or technique >
                    ↪ 13:
                        raise fuzztechniqueserror
            except:
```

```
                raise fuzztechniqueserror

            fuzzer = PJFFactory(config)

            ctx.log.info(f"Returning fuzzed
            ↪  JSON:\n{fuzzer.fuzzed}")
            response.content = fuzzer.fuzzed.encode()

addons = [FuzzClient()]
```

B.4.1 Flask fuzzer

This is the Python code of an alternative Flask proxy fuzzer from Section 3.8.1, that requires the use of the Burp Suite extension from Appendix B.3.1.

```
from flask import Flask
from flask import abort, request
from argparse import Namespace
from pyjfuzz.lib import *
import random
import json
import re
import requests

app = Flask(__name__)

@app.route('/', defaults={'path': ''})
@app.route('/<path:path>', methods = ['GET', 'POST', 'PUT',
↪  'DELETE', 'OPTIONS'])
def fuzz(path):
    print(f"{request.method} request at {path} - forwarding
    ↪  request to server...")
    r = requests.request(
        method          = request.method,
        url             = request.url,
        headers         = {k:v for k,v in request.headers if
        ↪  k.lower() != 'host'}, # exclude 'host' header
        data            = request.get_data(),
        cookies         = request.cookies,
        allow_redirects = False,
    )
    if r.text:
        try:
            payload = json.loads(r.text)
```

```
            except ValueError:
                print("ERROR: Response contains invalid JSON")
                abort(400)
        else:
            print(f"ERROR: No response: {r.status_code}")
            abort(404)

        if random.randint(0,1):
            config = PJFConfiguration(Namespace(json=payload,
            ↪  nologo=True, level=6, strong_fuzz=True))
        else:
            config = PJFConfiguration(Namespace(json=payload,
            ↪  nologo=True, level=6))
        config.techniques = [0, 4, 13]
        fuzzer = PJFFactory(config)

        print(f"Returning fuzzed JSON:")
        print(fuzzer.fuzzed)
        print()
        return fuzzer.fuzzed

    if __name__ == '__main__':
        app.run(ssl_context='adhoc')
```