CS453 Automated Software Testing

Spring 2019

Team Project
SWAGFuzzer, Fuzzer with State diversity in RESTful API

Doyeon Kim

Hyunki Kim

Junghwan Park

1. Introduction

Most of web services are programmingly accessed through REST APIs having a style that defines a set of constraints to be used for web services. Using REST APIs allows people to clearly communicate what each API is trying to do. Despite the large number of web services that use REST APIs, testing web services through its REST APIs is still in their infancy.

Web services have several states through REST APIs. To reach specific states, services must be located at particular states. So we need state diversity to test more states in services. But it's hard to infer the states because these have so many dependency between themselves.

To know the states of a target web service with their dependencies, we use Swagger (recently renamed OpenAPI) which has arguably become the most popular interface-description language for REST APIs. Swagger is a tool that takes REST APIs structure using JSON or YAML and makes interactive API documentation.

To predict target service's states with dependencies, we parse this API documentation and analyze target system. After we predict the states, we make REST API sequences that can explore all possible states using predicted states. Then we fuzz all of sequences to find bugs or vulnerabilities.

2. Swagger

Swagger is a specification method for RESTful API service. It offers a test-bed to test API works correctly when Swagger file is written. Also, it creates a documentation for RESTful API service.

GET /posts/ Returns list of blog posts
POST /posts/ Create a new blog post
DELETE /posts/{id} Delete a blog post with matching "id"
GET /posts/{id} Returns a blog post with matching "id"
PUT /posts/{id} Updates a blog post with matching "id" and "checksum"

Figure1. RESTful API documentation of our blog service through Swagger

We write a simple blog service with Flask and its swagger file. And Figure 1 is the documentation of our swagger file using Swagger UI.

Our simple blog service contains five API.

- `GET /posts/` : return list of blog posts in the current database.
- `POST /posts/` : Create a new blog post and return its *id*.
- `DELETE /posts/{id}` : Delete a blog post with matching *id*.
- `GET /posts/{id}` : Return a blog post's body and checksum with matching *id*.
- `PUT /posts/{id}` : Update a blog post with matching *id* and its *checksum*.

`GET /posts/` and `POST /posts/` are independent requests, because these API don't require dependent parameter. On the other hand, the APIs containing `{id}` are dependent requests, because `{id}` is one of dependent parameter that is generated by `POST /posts/`. In detail, `DELETE /posts/{id}` and `GET /posts/{id}` are dependent with `POST/posts/` because two requests require 'id' as a parameter and `POST /posts/` returns its id. So, `POST /posts/` should be preceded before the two requests. `PUT /posts/{id}` request has dependency with `POST /posts/` and `GET /posts/{id}`. The PUT request requires in 'id' and its 'checksum'. `POST /posts/` returns its id and `GET /posts/{id}` returns its checksum.

Using these request dependencies, we can draw request dependency graph shown in Figure 2.

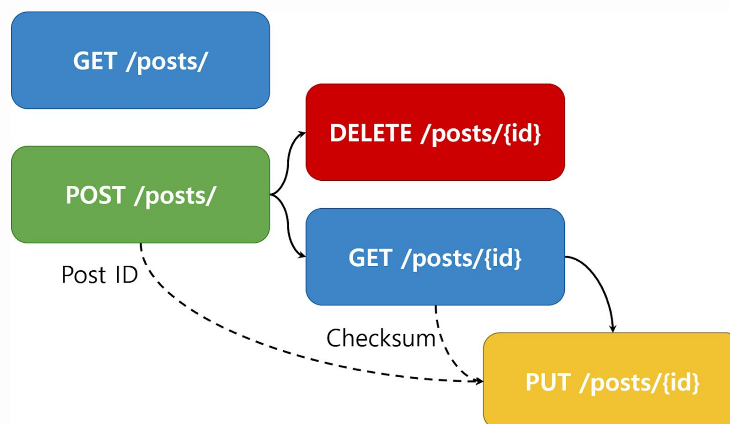

Figure 2. Dependency graph for our blog service

We regard that each request is a state. The deepest state is `PUT /posts/{id}`. So if we travel all the states including `PUT /posts/{id}`, we get the best state diversity.

3. Technique

We implemented a REST API fuzzer, *SWAGFuzzer*, to achieve state diversity thereby finding bugs or vulnerabilities in the web services. The *SWAGFuzzer* written in Python 3 with some modules consists of 3 main steps: parser, sequencer, and fuzzer.

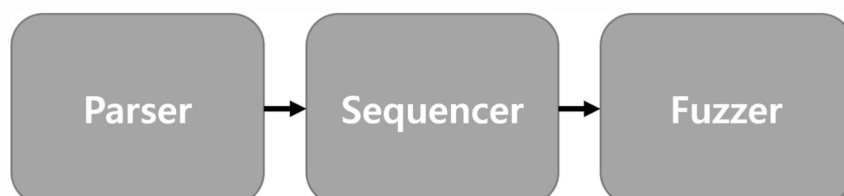

Figure 3. Core modules of SWAGFuzzer

In the first step, *SWAGFuzzer* parses a REST API documentation file written in YAML. It uses pattern matching to extract useful data such as method, path, required objects, return objects, and so on. The data will be used to infer request dependencies. In the parser step, *SWAGFuzzer* return all of request objects that contain several important information.

In the second step, all of request objects take as inputs. *SWAGFuzzer* inferences request dependency graphs using inputs. First of this step, it makes some information about consuming required objects and producing return objects in each request. Then it finds dependencies between requests to make request dependencies graphs. Finally using DFS algorithm, we can generate set of request sequences that travel all states on the request dependencies graphs. It will send set of request sequences that can explore all states as possible to the last step for data fuzzing and generating packets.

Finally, *SWAGFuzzer* fuzzes data with maintaining a format that should not modify to send to the server. Then it generates a new request packet based on the sequence set. It mutates data field based on the API documentation file used in the first step. It first checks what each data has a type such as integer, string, and so on. For example, when mutated type is string, *SWAGFuzzer* inserts a special character or strings with really long length for buffer overflow or other errors. Moreover, *SWAGFuzzer* uses different types with explicit types in API documentation because different types in the data field can make unexpected situations. *SWAGFuzzer* not only fuzzes a data field but also mutates a request sequence based on request sequence graph. To find an unexpected state, *SWAGFuzzer* just adds a random request to the end of request sequence. Why *SWAGFuzzer* adds a request at the end of a sequence is for maintaining dependencies in the sequence. Finally, with mutated data and sequence, *SWAGFuzzer* generates a packet and sends the packet to a target web server.

4. Evaluation

To evaluate *SWAGFuzzer*, there are two research questions:

Q1. Does *SWAGFuzzer* cover possible states?

Q2. Can *SWAGFuzzer* find practical bugs?

First question can show our implementation achieves state diversity that can perform state traversal to test a REST API and find a bug that existed in the program. Second question tells us our implementation carries out effective mutations that trigger the bugs.

We show a simple blog service in chapter 2. We inject an intentional bug to answer two research questions.

First bug is raising uncatched exception when checksum is correct in `PUT /posts/{id}`. By doing so, we can evaluate our implementation that reaches the deepest state and it can answer research question 1.

```
post = c.fetchone()
if hash(post[0]) == _checksum:
  raise Exception # Intentional bug 1
  c.execute('UPDATE post SET body=? WHERE id=?;', (_body, _id))
```

Figure 4. Intentional bug 1

Second intentional bug is an imitation of a practical bug in some programming language. PHP and javascript occurs type conversion in comparison. For instance, -1 and `-1` are different type of constants, and those languages return True when performing an equivalent operation. String '0e1234' that is float value in the exponential notation and False that represent negative in boolean type are obviously different, but those languages return True.

However, Flask is one of Python web framework. So we simulate this bug for our target service.

```
post = c.fetchone()

try: # Intentional bug 2
  if float(hash(post[0])) == 0:
    raise Exception
except ValueError:
  pass

if hash(post[0]) == _checksum:
  c.execute('UPDATE post SET body=? WHERE id=?;', (_body, _id))
```

Figure 5. Intentional bug 2

The experiment process has 3 steps. First, we execute cleaning our simple blog service with one intentional bug. For easy deployment, we write a Dockerfile that can create docker container. So, if the target environment is poisoned, we reload the container to setup environment. Second, we execute our implementation, *SWAGFuzzer*, about 1 hour with target swagger file and configuration file. Our fundamental approach is fuzzing test, so if we take more time to execute, we would expect get more uncatched exceptions or bugs. However we think 1 hour is enough to evaluate. Third, we inspect the fuzzing log generated in step 2. We record an last status code, the number of executed request in sequence, the sequence list, the mutated parameter, and context variables for one sequence in SQLite Database format.

	# of processed sequence	**# of uncatched exception**
Intentional bug 1	32,428	4,088
Intentional bug 2	31,982	2

Table 1. Experiment Result

The result with intentional bug 1 shows 12.6% of the number of process sequence. We choose a request set in sequence set randomly, and if our implementation doesn't select dependent variable, the uncatched exception should be raised. As you can see Figure 2, there are 5 possible sequences and two dependent variable in `PUT /posts/{id}` that could raise the exception. So, the number of uncatched exception is reasonable.

Therefore, we can answer our implementation covers possible state for research question 1.

The result with intentional bug 2 shows similar number of processed sequence with first one, and two uncatched exception is raised. First we expected those 2 sequences are our intentional bugs, however they aren't. They come from integer overflow that we cannot expected it. Python has defined maximum size or integer as `sys.maxsize`, however it can process more than the

value. However, our simple blog service uses a database system, SQLite3, and the ‘id’ type is integer, that has maximum value as $2^{63}-1$. So, the difference causes uncatched exception.

```
PUT /blog/posts/9233910012361406475
Content-Type: application/json
Accept: application/json
Host: localhost:5000

{"body": "", "checksum": 0.8115879519898935}
```

Figure 6. Unexpected bug in the experiment with intentional bug 2

Even though our implementation hasn’t found intentional bug 2, it finds unexpected practical bug. In other words, our mutation module of fuzzing phase works well. Thus we can answer `SWAGFuzzer` can find practical bugs for research question 2.

5. Conclusion

We present `SWAGFuzzer` that infers request state graphs with Swagger. Based on the graphs, it can successfully generate requests with mutation. We evaluated our implementation with simple blog post service. We showed our implementation has best state diversity and bug detection. For general results, we need to fuzz more services including real world services. But we argue that our tool can efficiently generate request state graphs in any other services. Also we can test all possible states in brief time.

6. Github Repository

https://github.com/kimdora/swagfuzz

7. Reference

- https://swagger.io/
- https://docs.python.org/3/library/sys.html#sys.maxsize
- https://www.sqlite.org/datatype3.html
- https://www.microsoft.com/en-us/research/uploads/prod/2018/04/restler.pdf