BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FUZZ TESTING OF REST API

FUZZ TESTOVÁNÍ REST API

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. PATRIK SEGEDY**
AUTOR PRÁCE

SUPERVISOR **Ing. VIKTOR MALÍK**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Master's Thesis Specification

23094

Student:	**Segedy Patrik, Bc.**
Programme:	Information Technology Field of study: Information Technology Security
Title:	**Fuzz Testing of REST API**
Category:	Software analysis and testing

Assignment:

1. Get acquainted with means of REST API specification and with fuzz testing. Study methods for automatic test case generation and for computing dependencies among API calls.
2. Design an algorithm for automatic computation of REST API call dependencies and for test case generation with respect to the obtained dependencies.
3. Implement the proposed algorithm as a console application
4. Test the created tool on at least two REST API specifications. Choose REST API of public open-source project and of projects withing the Red Hat company.
5. Evaluate bugs found during the testing.
6. Write the final text of the Master's in English.

Recommended literature:

- Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, Renwei Zhang.: Fuzz testing in practice: Obstacles and solutions.
- McNally, R.; Yiu, K.; Grove, D.; et al.: Fuzzing: The State of the Art. Technical Report DSTO-TN-1043. Defence Science and Technology Organisation. Edinburgh, South Australia 5111, Australia. 2012.
- Oficiální stránky projektu AFL: http://lcamtuf.coredump.cx/afl/
- Fielding, Roy Thomas (2000). "Chapter 5: Representational State Transfer (REST)". Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine.

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Malík Viktor, Ing.**
Consultant:	Kouřim Martin, Bc., RedHatCZ
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2019
Submission deadline:	June 3, 2020
Approval date:	October 31, 2019

Abstract

This thesis is dealing with fuzz testing of REST API. After presenting state-of-the-art of fuzzing and assessing the current research regarding REST API fuzz testing, we design and implement our REST API fuzzer. The proposed fuzzer infers dependencies of API calls defined in an OpenAPI specification and makes the fuzzing stateful. One of the features is minimization of the number of successive 404 responses while maintaining exploration of a deeper state space of a tested application. To solve the exploration vs. exploitation problem, we used the ordering of dependencies maximizing the probability of obtaining a needed input values and determining of fuzzability of a required parameters. The implementation is an enhancement of the Schemathesis project that is using the Hypothesis library to randomly generate inputs. Our fuzzer is evaluated against the Red Hat Insights application, finding 32 bugs. Amid them, one bug is reproducible only by a stateful set of steps.

Abstrakt

Táto práca sa zaoberá fuzz testovaním REST API. Po prezentovaní prehľadu techník používaných pri fuzz testovaní a posúdení aktuálnych nástrojov a výskumu zameraného na REST API fuzz testovanie, sme pristúpili k návrhu a implementácii nášho REST API fuzzeru. Základom nášho riešenia je odvodzovanie závislostí z OpenAPI formátu popisu REST API, umožňujúce stavové testovanie aplikácie. Náš fuzzer minimalizuje počet po sebe nasledujúcich 404 odpovedí od aplikácie a testuje aplikáciu viac do hĺbky. Problém prehľadávania dostupných stavov aplikácie je riešený pomocou usporiadania závislostí tak, aby sa maximalizovala pravdepodobnosť získania potrebných vstupných dát pre povinné parametre, v kombinácii s rozhodovaním, ktoré povinné parametre môžu využívať aj náhodne generované hodnoty. Implementácia je rozšírením Schemathesis projektu, ktorý generuje vstupy za pomoci Hypothesis knižnice. Implementovaný fuzzer je použitý na testovanie Red Hat Insights aplikácie, kde našiel 32 chýb, z čoho jednu chybu je možné reprodukovať len za pomoci stavového testovania.

Keywords

fuzz testing, fuzzing, fuzzer, REST API, testing, test generation, inferring dependencies, stateful testing, property based testing, Hypothesis, JSON Schema, OpenAPI, Swagger, Schemathesis

Kľúčové slová

fuzz testovanie, fuzzing, fuzzer, REST API, testovanie, generovanie testov, získavanie závislostí, stavové testovanie, Hypothesis, JSON Schema, OpenAPI, Swagger, Schemathesis

Reference

SEGEDY, Patrik. *Fuzz Testing of REST API.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík

Rozšírený abstrakt

Testovanie pomocou generovania nečakaných, nevalidných alebo náhodných vstupných dát je známe pod pojmom fuzz testovanie. Fuzz testovanie je jednou z najviac úspešných techník pre hľadanie softvérových zraniteľností a vývíja sa každým rokom od jeho vzniku v osemdesiatich rokoch dvadsiateho storočia. V dnešnej dobe existuje mnoho výnimočných nástrojov na fuzz testovanie. Niektoré využívajú mutovanie vstupných dát, iné sledujú pokryté cesty v zdrojovom kóde pomocou genetických algoritmov. Ich spoločným cieľom je vytvoriť testy odhaľujúce čo najviac zraniteľností za pomoci zvyšovania pokrytia programu testami. Ako prvé vznikli nástroje pre fuzz testovanie konzolových aplikácií, neskôr sa k nim pridali nástroje testujúce sieťové protokoly prostredníctvom vytvorenia ich gramatiky. Nakoniec vznikli white-box prístupy využívajúce inštrumentáciu zdrojového kódu počas testovania programu.

Avšak, oblasť fuzz testovania REST API zatiaľ nebola dostatočne preskúmaná. Samozrejme, nástroje pre testovanie webu by mohli byť použité, ale vo svojom návrhu nezohľadňujú špecifiká pre REST API. V súčasnosti sa rozvíja trend vytvrania aplikácii ako mikroslužieb. Každá mikroslužba má vlastné API pre komunikáciu s ostatnými službami alebo s užívateľom aplikácie. S rastúcim počtom závislých mikroslužieb sa testovanie týchto mikroslužieb stáva komplikovaným. Tieto služby zvyčajne využívajú REST API pre komunikáciu a musí byť zaručené, že každý prístupový bod aplikácie sa správa podľa očakávaní užívateľa a je zabezpečený. REST API sú často špecifikované jedným z existujúcich formátov pre popis REST API, ktoré zjednodušujú ich vytváranie, údržbu a z pohľadu fuzz testovania, umožňujú automatické vytvorenie testov na základe špecifikácie. Nedostatočný výskum v oblasti fuzz testovania REST API spojený s veľkým počtom nájdených chýb pomocou bežného fuzz testovania nám ukazuje potenciál pre zlepšenie algoritmov generujúcich testy pre REST API.

Viacero nástrojov testujúcich REST API zdieľa jednu nevýhodu. Testujú zhodnosť špecifikácie a výstupu aplikácie, ale nevytvárajú testy s nevalidným vstupom.

Naša práca vytvára nástroj špecifický pre fuzz testovanie REST API. Základom nášho riešenia je odvodzovanie závislostí z OpenAPI formátu popisu REST API, umožňujúce stavové testovanie aplikácie. Náš fuzzer minimalizuje počet po sebe nasledujúcich 404 odpovedí od aplikácie a testuje aplikáciu viac do hĺbky. Problém prehľadávania dostupných stavov aplikácie je riešený pomocou usporiadania závislostí tak, aby sa maximalizovala pravdepodobnosť získania potrebných vstupných dát pre povinné parametre. Výhodou našej implementácie je taktiež možnosť rozhodnutia, ktoré povinné parametre môžu využívať aj náhodne generované hodnoty. Fuzzer je implementovaný v jazyku Python, ako rozšírenie nástroja Schemathesis, ktorý pre generovanie vstupných hodnôt využíva knižnicu Hypothesis.

Schopnosti implementovaného fuzzeru boli vyhodnotené na službe Red Hat Insights, kde náš fuzzer bol schopný objaviť 32 nových chýb, pričom jedna z chýb bolo reprodukovateľná práve za pomoci stavového testovania.

Fuzz Testing of REST API

Declaration

I hereby declare that this master's theses was prepared as an original work by the author under the supervision of Ing. Viktor Malík. The supplementary information was provided by Bc. Martin Kouřim. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Patrik Segedy
June 3, 2020

Acknowledgements

I would like to thank to Ing. Viktor Malík for his very valuable recommendations on how to write a thesis and his exhaustive text corrections and ideas how to improve the presentation of the thesis. I want to thank to Bc. Martin Kouřim for his technical leadership and the introduction to fuzz testing. Also, I like to thank to Red Hat company for the opportunity to work on this assignment. Last, but not least, I want to thank to my family for supporting me throughout my whole academic studies.

Contents

Chapter 1

Introduction

Testing by generating unexpected, invalid, or random input data, also known as fuzzing, is one of the leading techniques in discovering software vulnerabilities. Fuzz testing is evolving every year since the first experiments in the 1980s. Nowadays, awesome fuzzing frameworks exist, capable of using mutation to create new test cases, watching code paths and employing genetic algorithms during testing to maximize coverage of a tested program, and, ultimately, finding security vulnerabilities caused either by programming flaws or by design issues. Over the years, fuzzing frameworks for various use cases were developed, from tools for testing command-line applications with random inputs, grammar-based network protocols fuzzers, to white-box fuzzers instrumenting target programs. However, the area of REST API fuzz testing is not as deeply researched as others.

Why would one want to have tools that are specific to REST API? In these days, there is a trend to create web services following microservice architecture. Every microservice has its own API for communication with other microservices that makes integration testing of the whole service complicated, especially with the increasing number of dependent microservices. These services are usually utilizing REST API for communication and it must be assured that each endpoint is doing what it is supposed to and that it is secure. REST APIs are commonly specified using some description format that makes it easy to create fully automated test cases based on API specification. The lack of the research in REST API fuzzing and the number of bugs found by fuzzing in different applications gives us an opportunity to come up with a better test generation algorithms and to fulfill its potential by finding interesting bugs.

Many tools for testing REST API are using only the expected inputs to test the conformance of the application to its REST API specification. The goal of this thesis is to create a tool that will test also negative cases using fuzzing. This will be achieved by inferring dependencies between REST resources and by injecting expected inputs acquired from the specification, while avoiding generation of tests that are making requests with invalid combination of dependencies. This way, a small set of meaningful tests is created, that will explore a large portion of code paths.

The content of this thesis consists of a general description of fuzz testing in Chapter 2. This chapter contains terminology, definitions, and overview of fuzz testing methods, including the basic taxonomy of fuzzers. Later, basic workflow and building blocks of a general fuzzer are presented, and the importance of fuzz testing is demonstrated by vulnerabilities and numerous bugs found by commonly used fuzzers. Chapter 3 is dedicated to REST API Fuzzing. This chapter starts with the definition of the Representational State Transfer (REST) architectural style with its constraints. This is followed by specification of formats used to describe REST API and the security problems of applications using the REST approach. At the end of the chapter, the current publications about REST API test generation are discussed and some tools for testing REST API are presented. Chapter 4 explains design of the REST API fuzzer. It also defines a running example that will be used in the subsequent text explaining the design and implementation details. The proposed design of the fuzzer is implemented in Chapter 5 by modifying the Schemathesis tool. Finally, the implemented tool is evaluated against our testing application having a stateful bug, several experiments are conducted to justify implementation details, and the fuzzer is used to find bugs in Red Hat Insights in Chapter 6.

Chapter 2

Fuzz testing

Fuzzing, at a high level, refers to the technique of running a program with generated unexpected, invalid, or random input data that may be syntactically or semantically incorrect. The program is then monitored for failures, such as failing assertions of correct behavior, exceptions, and memory leaks. Fuzzing is widely used by malicious attackers to generate exploits, as well as by defenders for penetration testing in an attempt to discover vulnerabilities faster than attackers do. Numbers of prominent vendors such as Adobe, Cisco, Google, or Microsoft use fuzzing in their software development process to secure the software. Security auditors and open-source developers have recently also started to employ fuzzing to measure software security to assure end-users that the provided software is secure [26].

The term fuzz was born in Madison in the Fall of 1988 by professor Barton Miller during one dark and stormy night. That night, Professor Miller was logged on to the Unix system in his office via a dial-up connection. A Heavy rain created noise interfering with the professor's ability to type sensible commands to the shell and programs. That was not surprising, however, what did surprise him was that the noise seemed to cause programs to crash. To make a systematic scientific investigation to understand the problem and the cause, the professor suggested a new course project in the course on Advanced Operating Systems at the University of Wisconsin, but to describe the project, it was needed to give this kind of testing a name. Professor Miller settled on the term „fuzz“ because he wanted a name that would evoke the feeling of unstructured, random data [32]. The goal of the project itself was to evaluate robustness of various Unix utilities, given an unpredictable input. The first part of the project was to build a fuzz generator, the program that will create a stream of random characters, and the second part was to use the fuzz generator to attack as many utilities as possible. The results of this project were alarming. The best group of students succeeded well beyond professors' expectations. On seven Unix variants, they crashed between 25-33% of the utility programs [28].

2.1 Terminology

Fuzzing community is very vibrant. The literature contains a number of fuzzers and the number of fuzzing studies appearing at major security conferences is increasing, and also GitHub hosts over a thousand public repositories about fuzzing. With such a popularity, systematization problems arise. Some fuzzers lack documentation and it is easy to lose track of the design decision, while others are using different terms to describe the same technique or a similar term for different techniques. For example, AFL fuzzer uses the term

„test case minimization“ for reducing the size of a crashing input, the same technique is called „test case reduction“ in the *funfuzz* fuzzer, but the BFF fuzzer has a similar-sounding technique called „crash minimization“ that is not related to reducing the input size. Such fragmentation makes it difficult to discover fuzzing knowledge and may decelerate progress in fuzzing research [26].

For this reason, we now introduce the terminology unified by 2019's article by Manes et al. [26] in relation to Program Under Test (PUT) which we will use throughout this thesis

Fuzzing is an execution of the PUT using input(s) sampled from an input space (the „fuzz input space“) that protrudes the expected input space of the PUT [26].

Authors of a survey unifying terminology made three following remarks [26]:

- It is not necessary that the fuzz input space contains the expected input space. It is sufficient when the fuzz input space contains an input not present in the expected input space.
- Fuzzing, in practice, runs for many iterations.
- The sampling process is not necessarily randomized.

Fuzz Testing is the use of fuzzing to test if the PUT violates a correctness policy [26].

Historically, fuzz testing has been used mainly to find security vulnerabilities. Nowadays, it is also used to find non-security bugs [26].

A Fuzzer is a program that performs fuzz testing on the PUT [26].

Fuzz Campaign is a specific execution of a fuzzer on the PUT with a specific correctness policy [26].

A Violation of the specified correctness policy is achieved by finding bugs by running the PUT through a fuzz campaign. An example of a policy violation is crashing the PUT by the test case [26].

A Bug Oracle is a program, perhaps a part of a fuzzer, that determines whether a given execution of the PUT violates a specific correctness policy [26].

Fuzz Configuration of a fuzz algorithm comprises the parameter value(s) that control(s) the fuzz algorithm [26].

Fuzz configuration is a broad term. It depends on the type of the fuzz algorithm since the fuzz algorithm may depend on some parameters beyond the PUT. Each concrete setting of the parameters is a fuzz configuration [26].

2.2 Basic Taxonomy of Fuzzers

Fuzzers can be categorized into three groups based on how much information about the PUT is gathered in each fuzz run. In traditional software testing, we distinguish two types of testing, black-box, and white-box. The classification of fuzzers is slightly different, it has three types of fuzzers. These three groups are called black-box, grey-box, and white box fuzzers. We will look more into techniques used in all three groups of fuzzing in Section 2.5.2.

2.2.1 Black-box Fuzzer

In software testing, the term black-box is commonly used and denotes techniques that do not see the internals of the PUT. In fuzzing, a black-box fuzzer observes only input/output of the PUT. Fuzzers take the structural information about inputs into account to generate meaningful test cases [26]. Black-box fuzzer uses predefined rules to randomly mutate a given valid seed to create slightly malformed but still valid input [24].

2.2.2 White-box Fuzzer

White box fuzzing uses information about the internal logic of a target program to generate test cases. Unlike black-box fuzzer, white-box fuzzer starts execution by gathering symbolic constraints at all conditional statements. The fuzzer combines symbolic constraints to form a path constraint. One of the constraints is then negated and new path constraint is solved. This creates new test cases that explore different execution paths of a program [24].. Due to exploring the state space, dynamic symbolic execution, and satisfiability solving, white-box fuzzers have typically much higher overhead than black-box fuzzers [26].

2.2.3 Grey-box Fuzzer

A middle-ground approach is called grey-box fuzzing. These fuzzers obtain some internal information about the PUT or its executions. Unlike white-box fuzzers, they do not reason about the full semantics of the program, instead, grey-box fuzzers gather approximated, imperfect information about the program by utilizing lightweight static analysis or dynamic code coverage information [26]. Grey-box fuzzers can obtain code coverage of the target program at runtime and use this information in following adjustments of mutation strategies to create new test cases [24].

2.3 Fuzz Testing Overview

The term fuzzing was introduced in 1988, but it wasn't the beginning, the big bang of fuzzing. Software engineers used methods similar to fuzzing since the 1980s, but it wasn't called fuzzing back in the days. According to *Fuzzing for Software Security Testing and Quality Assurance* by Takanen et al. [32], software testing for security and reliability was not widely spread, and it looked like nobody cared about software quality, since the concept of an attacker was unknown. In Figure 2.1, we can see the milestones of fuzzing evolution.

Fuzz testing is still a vital and popular topic amongst researchers even after more than thirty years of research and development. As a demonstration of fuzzing popularity, Figure 2.2 shows the number of publications over years in the fuzz testing domain.

Fuzzing can be divided into two types. An older and a simpler type of fuzzing is fuzzing based on a grammar. A newer approach is to use code coverage information to guide the generation of test cases. In the rest of this section, we present the most popular and used approaches to both types of fuzzing, including the latest published academic papers.

2.3.1 Grammar-based Fuzzing

In the begging, researchers tried to find exploitable security holes by generating random inputs for command-line options with the Fuzz tool. Although it sounds naive, its ability to discover bugs was impressive. The Fuzz tool testing strategy was searching for undefined

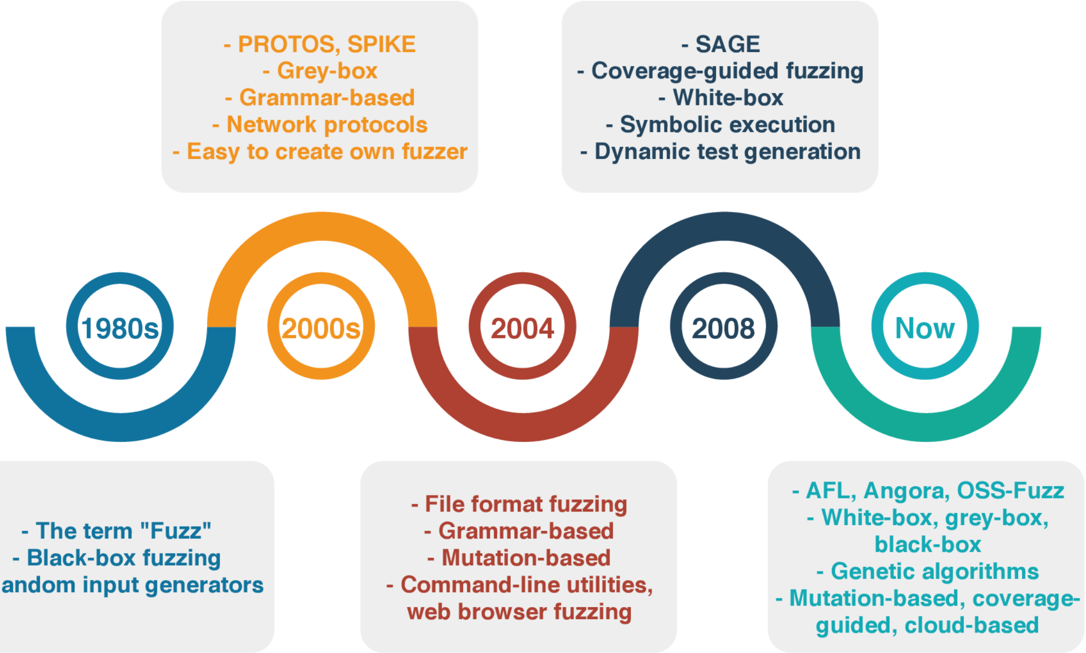


Figure 2.1: Fuzzing evolution

states by a random walk through the state space [32]. It was an overly simplistic, pure black-box approach, but we have to keep in mind that the concept of fuzzing was unheard of at the time [31].

The first modern fuzzer was a set of fuzzing test suites created by analyzing protocol specification, called PROTOS project. In contrast with the Fuzz, PROTOS is a nice example of **mixing white box and black box testing** [31][32].

Another tool marking a significant milestone in fuzzing history was named SPIKE. Mostly, because it **allowed users to easily create their own fuzzers**. It was the more advanced, open-source fuzzer intended for network-enabled applications. It has the ability to describe variable-length data blocks, generate random data, but is also bundled with a library of values that will likely produce faults [31].

After fuzz testing command-line utilities, network protocols, and web browsers, file fuzzing came into vogue in 2004. With file format vulnerabilities another milestone was marked and **mutation-based fuzzing** widened. Files turned out to be suitable for mutation testing since they can be mutated and the target application can be monitored for faults.

While the grammar for the SPIKE was based on the network protocol specification, grammar for file fuzzers followed the tested file format. [31].

Grammar-based methods are still researched. Since the majority of existing fuzzing methods do not take the structure of inputs for the target program into account. Authors of *Grammar-based Fuzzing* [30] came with a new method based on BNF grammar. Every rule of the grammar is designed as an universal pushdown automata, which allows generating BNF compatible data. Authors claim they were able to increase code coverage significantly.

Interestingly, members of the PROTOS team launched a company named Codenomicon [31]. Their researchers, as well as the others, were the ones who discovered the infamous

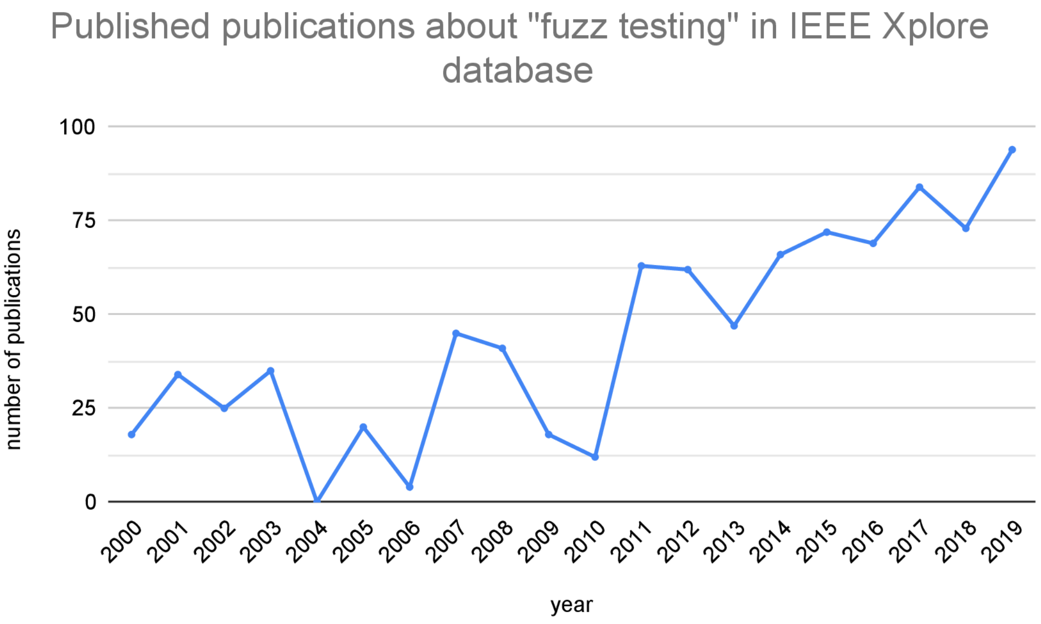


Figure 2.2: Fuzzing research

Heartbleed bug [34]. We cover more details about this and other bugs found by fuzzing in Section 2.4.

2.3.2 Coverage-guided Fuzzing

Traditionally, fuzzers have been very dependant on the input samples or on the provided grammar [32]. Patrice Godefroid et al. [18] came with an alternative approach to traditional fuzzers and developed the SAGE (Scalable Automated Guided Execution) fuzzer. Algorithm of SAGE was inspired by advances in **symbolic execution and dynamic test generation**.The authors called this new approach the **White-box fuzzing** [18]. SAGE had a remarkable impact on Microsoft products since it found many issues thanks to a combination of program analysis, testing verification, model checking, and automated theorem-proving techniques [19]. Since SAGE, many new fuzzers and proofs-of-concept started to use evolutionary and genetic algorithms in combination with code coverage to defeat traditional fuzzers [32].

Numerous other frameworks and fuzzers for various use cases emerged. Rather than talking about dead and forgotten applications and fuzzers for surpassed technologies like ActiveX, let us move forward to a more recent history and the presence. The release of the American Fuzzy Lop (AFL)[1] fuzzer by Michal Zalewski meant a major leap in the usability of advanced fuzzing tools. AFL is a security-oriented fuzzer, leveraging compile-time code instrumentation combined with **genetic algorithms** to discover untested, new code paths. It was not the first usage of such techniques, however, AFL was the first tool combining it into an easy-to-use tool to be used without the need of in-depth technical understanding. LLVM libFuzzer is similar to AFL with focus on performance testing and fuzzing of libraries.

[1] American Fuzzy Lop - https://github.com/google/AFL

Since then, fuzz testing is time-consuming and exhausting finding of uncovered paths in code, because of that, new cloud-based fuzzing services emerged. Examples are, Google's ClusterFuzz[2], a **cloud-based fuzzing** infrastructure for fuzzing security-critical components of Chromium web browser[3], later used as a backend for OSS-Fuzz[4] targeting open-source software[5], or Microsoft's first commercial cloud-based fuzzing service, Microsoft Security Risk Detector[6]. With scaling capabilities of cloud infrastructure, these cloud-based fuzzing services are able to use coverage guided fuzzers executed in parallel [32].

Current research is focused on improving fuzzing speed by solving path constraints without symbolic execution like Angora [10]. Transforming the target program to remove sanity checks like T-Fuzz [29]. Increasing efficiency by utilizing static and dynamic analysis of the program to create branch predictions [35]. Some researchers are improving AFL functionality by modifying internal data structures to reduce hash table collisions [17], others use Markov chain model to guide the fuzzing [5] or Markov decision process to formalize fuzzing as a reinforcement learning problem [6][23].

2.3.3 New Fuzz Testing Utilization

Fuzzing is not only used to find vulnerabilities in computer programs and utilities but also in other sectors of cybersecurity. One example could be the Fuze [9] project aiming at fuzzing smart contracts and support fuzz testing of decentralized applications. Another example is a research on fuzz testing in the automotive industry. Computational complexity within a connected car, especially with the advent of autonomous vehicles is increasing. Researchers experimented with fuzz testing against a target vehicle's CAN bus to demonstrate that the fuzz testing has a part to play as one of the many security tests that a vehicle's systems need before series production [16].

2.4 Vulnerabilities Found by Fuzzing

Vulnerabilities can be introduced in various phases of the Software Development Life Cycle (SDLC) shown in Figure 2.3, in design, implementation, and deployment phase. Issues originated in the design phase are fundamental defects that are difficult to fix. Implementation defects are the most common and they are usually caused by bad practices or mistakes in the implementation of the product. Lastly, deployment issues are caused by incorrect and not secure configuration of deployed product, often caused by bad documentation on how to deploy product securely. By analyzing these phases with regards to experience about known mistakes, we observe that implementation flaws prevail. More than 70% of security vulnerabilities are caused by programming flaws, about 20% are caused by bad design, and less than 10% are configuration issues causing not secure deployment [32].

Fuzzing is able to find issues in all the phases, but since the programming faults are the most common, fuzzing will find most issues caused by poor implementation or bad practices. However, deployment security flaws, such as making a management API accessible without authorization, or some design problems can be also found by fuzz testing.

[2] ClusterFuzz - https://github.com/google/clusterfuzz

[3] ClusterFuzz announcement - https://blog.chromium.org/2012/04/fuzzing-for-security.html

[4] OSS-Fuzz - https://github.com/google/oss-fuzz

[5] OSS-Fuzz announcement - https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

[6] Microsoft Security Risk Detector - https://www.microsoft.com/en-us/security-risk-detection/

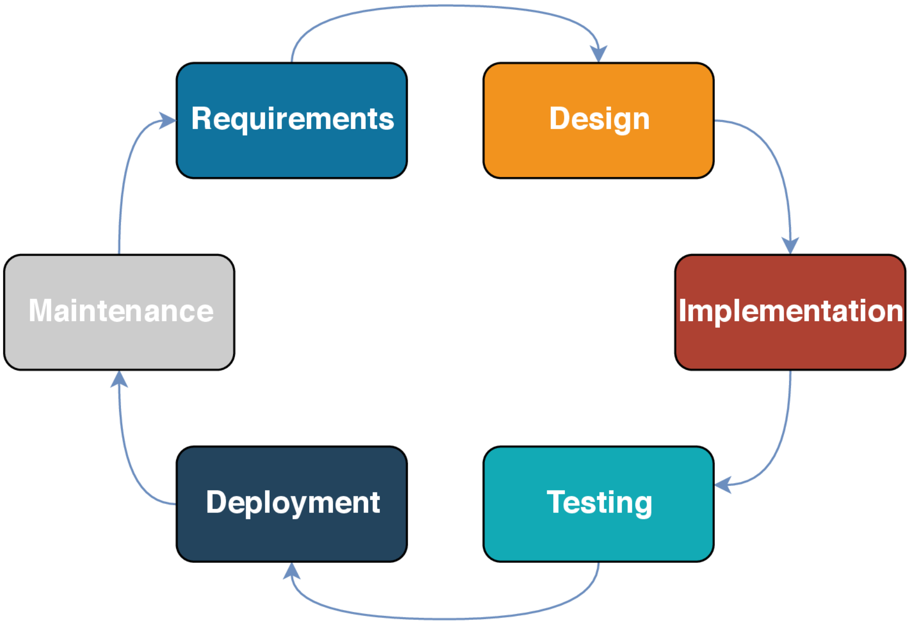


Figure 2.3: Software Development Life Cycle

In this section, we present some of the most infamous CVEs and security issues that were directly found by fuzzing, or whose discovery was made possible thanks to fuzz testing.

These cases demonstrate usefulness and practical applicability of fuzz testing.

2.4.1 Shellshock

Shellshock is a family of vulnerabilities related to a bug called Backdoor (CVE-2014-6271). The Shellshock bug affects GNU Bash, a very popular Unix shell. It is a vulnerability in Bash functionality that evaluates environment variables passed to it from another environment. An attacker could use this feature to execute shell commands before restrictions to the environment have been applied. This leads to privilege escalation vulnerability.[7] Most of the vulnerabilities, in the Shellshock Family were found by the AFL fuzzer.[8]

2.4.2 Heartbleed

Another serious vulnerability found with the help of fuzzing is Heartbleed[9]. It is a bug in the OpenSSL implementation of transport layer security protocols TLS/DTLS heartbeat extension. A heartbeat service is used to check whether the server on the other end is still alive. It works on a simple principle. A client would send the message containing a keyword and its length, and the server should reply the keyword back if it is still alive. The Heartbleed bug was exploited by sending a specially crafted message where the length of the keyword was actually bigger than the keyword itself. The consequence is that the server would reply with the provided keyword concatenated with other information from memory. The explanation can be seen in Figure 2.4. Heartbleed security vulnerability was

[7] Shellshock Red Hat bugzilla - https://bugzilla.redhat.com/show_bug.cgi?id=1141597

[8] Shellshock found by AFL - https://lcamtuf.blogspot.com/2014/09/quick-notes-about-bash-bug-its-impact.html

[9] Heartbleed - http://heartbleed.com/

found by researchers from Codenomicon and Google by compiling the OpenSSL library with a memory sanitizer, to notice an out-of-bound memory access occurrence, followed by fuzzing [34]. Later on, it was described by Hanno Böck how the Heartbleed bug could have been found by AFL.[10]

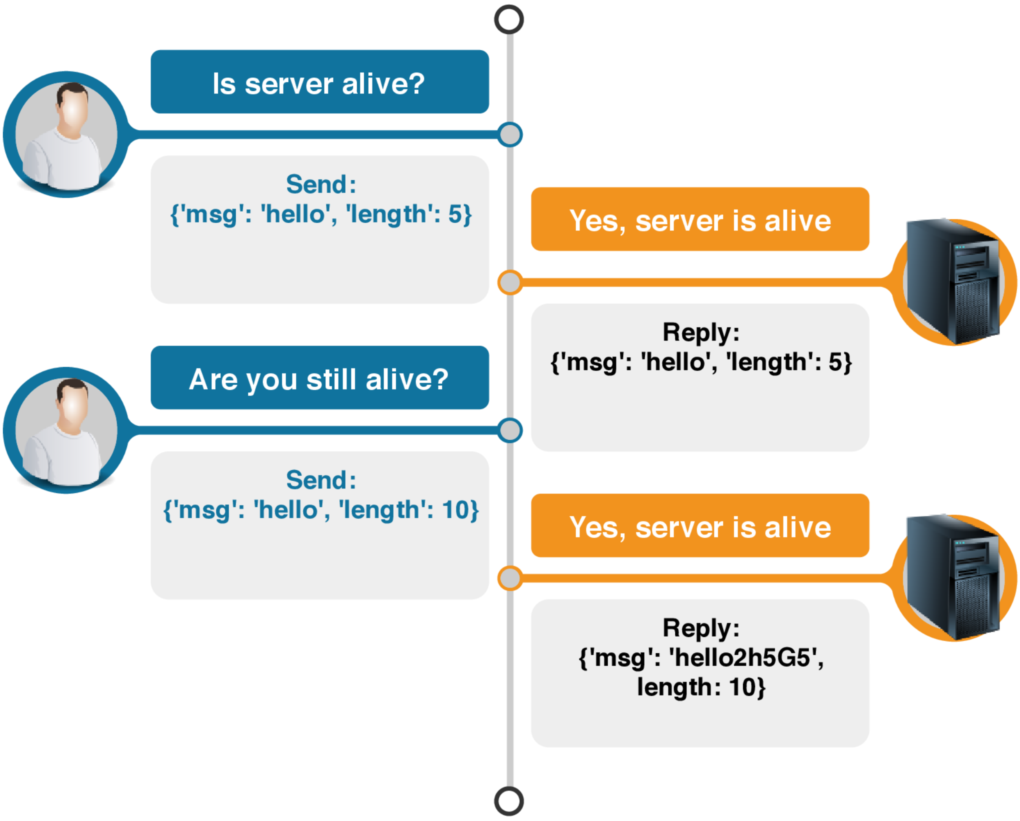


Figure 2.4: Heartbleed illustration

2.4.3 Statistics of Current Fuzzers

Looking at the list of notable vulnerabilities and other uniquely interesting bugs that were found by AFL, fuzzing is very successful in finding bugs. AFL found numerous bugs in 161 products, projects or libraries. To name a few, it found vulnerabilities in web servers as Apache httpd or nginx, Mozilla Firefox or Apple Safari web browsers, mobile operating systems Android and iOS, various open-source libraries as well as in OpenBSD or iOS kernel.[11]

Cloud-based fuzzing services are also very successful, as of December 2019, OSS-Fuzz found more than 15,000 bugs in 200 open-source projects.[12]

The use of fuzzing as a part of SDLC is proactive and makes it easier to find zero-day flaws before product release. Security or vulnerability scanners are reactive tools that fail to do that because they are based on knowledge of previously found vulnerabilities. Reactive

[10] How Heartbleed could have been found - https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html

[11] AFL - http://lcamtuf.coredump.cx/afl/

[12] OSS-Fuzz bug tracking - https://bugs.chromium.org/p/oss-fuzz/issues/list?can=1&q=-status%3AWontFix%2CDuplicate+-Infra

tools are testing only widely used products from major vendors, however, with fuzz testing it is possible to test any process, service, device, or system for security flaws regardless interfaces it uses [32].

2.5 Anatomy of a Fuzzer

In this section, we present a general structure of a fuzzer. Typically a fuzzer's anatomy consists of several phases whose implementation may vary based on the target application and on the format of the data that is fuzzed.

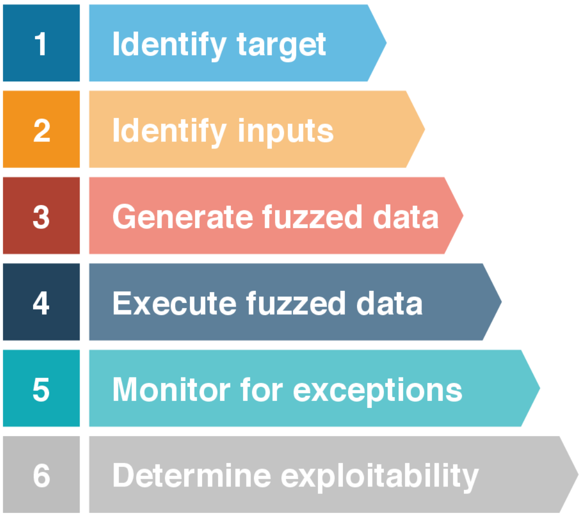


Figure 2.5: Fuzzing phases

Based on the *Fuzzing—Brute Force Vulnerability discovery* book [31], fuzzing can be divided into the following six basic phases, also seen in Figure 2.5:

1. **Identify target.** First, it is neccessary to identify the target application to select an appropriate fuzzing approach. It is different to fuzz an internally developed application during a security audit, and a third-party application in order to discover vulnerabilities. Moreover, the target is not neccessarily an entire application, it can be a specific file or a library within the application.

2. **Identify inputs.** The crucial part for the success of fuzzing is enumerating input vectors. Some input vectors might be obvious, others are subtler. It must be kept in mind that anything sent from the client to the target should be considered an input vector, including headers, file names, environmental variables, etc.. Exploitable vulnerabilities are commonly caused by application accepting malicious user input without sanitizing it.

3. **Generate fuzzed data**, or in other words, create fuzz configuration. Once the input vectors are known, fuzz data must be generated. Multiple strategies can be used such as using predetermined values or random data, mutating existing data, or generating data dynamically. The strategy depends on the target and the data format. The generation should be an automated process.

```python
def fuzz(self, fuzz_configs: List[Config], time_limit: int) ->Set[Bug]:
    """General fuzzer."""
    bugs ={}
    # remove redundant configurations, instrument the PUT
    fuzz_configs =self.preprocess(fuzz_configs)
    # run fuzzing until time_limit
    # or if there are no more paths to discover
    while self.time_elapsed <time_limit and self.cont(fuzz_configs):
        # select a fuzz configuration for the current iteration
        conf =self.schedule(fuzz_configs, time_elapsed, time_limit)
        # generate test cases from fuzz configuration
        tests =self.input_gen(conf)
        # execute test cases
        # collect bugs and gather information about test runs
        new_bugs, exec_info =self.input_eval(conf, tests, self.bug_oracle)
        # update fuzz configurations based on the result of test runs
        fuzz_configs =config_update(fuzz_configs, conf, exec_info)
        bugs.update(new_bugs)

    return bugs
```

Listing 2.1: Algorithm of a general fuzzer

4. **Execute fuzzed data.** This step depends on the previous one. Fuzzed data are executed by sending a data packet to the target, opening a file, or launching the target process. It has to be automated, otherwise, the process is not a fuzzing.
5. **Monitor for exceptions.** An often overlooked step, that is a vital part of fuzzing. Imagine transmitting 10,000 fuzz packets to a target web server, causing the server to crash. All the work would be useless if we were not able to pinpoint the packet responsible for the crash.
6. **Determine exploitability.** It might be necessary to determine if the found bug can be exploited. Determining exploitability requires specialized security knowledge and it is usually performed by someone other than the person conducting the fuzzing and it depends on the goals of the audit.

To describe a generic fuzzer more in detail, we use an the example of a generic fuzzer algorithm based on the algorithm from a survey *The Art, Science, and Engineering of Fuzzing* [26] seen in Listing 2.1. Description of methods used for each part of the algorithm is provided in the following subsections.

2.5.1 Preprocess

The first step of some fuzzers is to prepare the main loop of the algorithm in Listing 2.1 by modifying the initial set of fuzz configurations. The goal of such preprocessing is to instrument the PUT and to remove potentially redundant configurations, to generate driver applications, or to prepare a model for future input generation [26].

Instrumentation

The amount of collected information during PUT instrumentation distinguishes the type of the fuzzer between black, white, or grey-box fuzzer. Program instrumentation can happen in `preprocess` phase for **static** instrumentation, or during `input_eval` phase, if the instrumentation is **dynamic** [26].

A benefit of *static instrumentation* is less runtime overhead since it is performed at compile-time, before runtime. However, it is not suitable for programs that rely on multiple libraries. These libraries have to be instrumented and recompiled separately. *Dynamic instrumentation's* downside is higher overhead than static instrumentation, but it has the advantage of easily instrumenting dynamically linked libraries thanks to doing the instrumentation at runtime. Generally, a fuzzer can support both the static and the dynamic instrumentation [26].

Seed Selection and Trimming

Some parameters of fuzz configurations, for example, seeds for mutation-based fuzzers, can have infinite domains. Imagine fuzzing of an MP3 player that accepts MP3 files as input. The number of valid MP3 files is unbounded, and therefore it is hard to select a seed used for fuzzing. This problem is known as the *seed selection problem*. A common approach to address this problem is a method for finding a minimal set of seeds that maximizes a coverage metric (i.e. node coverage), called *minset*. Minset of some fuzzers, like AFL, is based on branch coverage, while others compute coverage based on the number of executed instructions, branches, and unique basic blocks. Adding longer executions to the minset can help discover performance issues or denial of service vulnerabilities. This step is part of the `config_update` [26] phase.

To consume less memory and to ensure higher throughput, some fuzzers attempt to reduce the size of the seed prior to fuzzing. This method is called *seed trimming* and can happen prior to fuzzing loop in `preprocess` or during `config_update`. Seed trimming can be implemented as iteratively removing a portion of the seed when the modified seed achieves the same coverage [26].

Generating a Driver Application

Sometimes, it is difficult to directly fuzz the target. In these cases, it makes sense to prepare a driver for fuzzing. This is performed once at the begging of the fuzz campaign and it is largely manual process. Fuzzing a library is one example when a driver application is needed. The driver program is calling functions in the library, thus the library can be fuzzed. This approach is used by kernel fuzzers or IoT fuzzers [26].

2.5.2 Scheduling

Selecting a fuzz configuration for the next fuzz iteration is called scheduling. It highly depends on the type of the fuzzer, and its goal is to analyze the available information about the configurations and pick one that more likely leads to the most favorable outcome, as finding bugs or increasing the coverage. Advanced fuzzers use innovative scheduling algorithms which are a major factor for their success. Every scheduling algorithm has to solve *exploration vs. exploitation* conflict. Exploration is spending more time on gathering more accurate information on configuration to inform future decisions, while exploitation is spending time on fuzzing the configurations that are believed to lead to better outcomes. This problem is called **Fuzz Configuration Scheduling (FCS) Problem**. In algorithm 2.1, `schedule` function selects the next configuration based on fuzz configurations, elapsed time and remaining time [26].

Black-box FCS Algorithms

The only information an FCS algorithm gets from a black-box fuzzer is the outcome of a configuration, number of bugs and crashes, and the time spent on configuration. The number of unique crashes in a fixed amount of time can be increased by favoring configurations with higher success ($number\ of\ unique\ crashes/runs$) probability. Common strategy for coping with exploration vs. exploitation problem is applying *multi-armed bandit*[13] algorithm to fuzzing. Another improvement is in normalizing the success probability by the time spent in it to prefer faster configurations, and not running a fixed number of fuzz runs, but limit a fuzz iteration to a fixed amount of time to further deprioritize slower configurations [26].

Grey-box FCS Algorithms

An FCS algorithm for grey-box fuzzer can choose to use from a richer set of configuration information, such as coverage information. Many fuzzers use evolutionary algorithms (EA) to maintain a population of configurations with their fitness value. Offspring of fit configuration is produced using transformations such as mutation and recombination. A fuzzer maintains a circular queue of configurations from which it selects the next fastest configuration with the smallest input that has the highest coverage [26].

2.5.3 Input Generation

One of the most influential design decisions in a fuzzer is the *input generation* technique that controls the content of a test case and triggers bugs. Input generation is defined by the `input_gen` function of algorithm in Listing 2.1. In this section, we describe Generation-based (model-based) fuzzers that produce test cases based on a given model, and Mutation-based (model-less) fuzzers producing test cases by mutating a given seed input.

Generation-based Fuzzers

Generation-based fuzzers are fuzzers that generate test cases based on a given model describing the inputs or executions that the PUT may accept. Three types of models exists [26]:

- **Predefined Model**. Some fuzzers use a model configured by the user, using some kind of template, grammar, or network protocol specification provided by the user. Other model-based fuzzers can target a specific language or grammar. These fuzzers have built-in model based on target language.
- **Inferred Model**. Only a few fuzzers utilize the technique of inferring the model rather than relying on a predefined or user-provided model. Some fuzzers infer the model during the `preprocess` phase, focusing to generate semantically valid inputs. Others are trying to update their model after each fuzz iteration during `config_update`.

[13] Multi-armed bandit - https://en.wikipedia.org/wiki/Multi-armed_bandit

- **Encoder Model**. This model is used to test *decoder* programs that parse a certain file format. New test cases are created by mutating the *encoder* program to produce test cases that are slightly malformed.

Mutation-based Fuzzers

Generating test cases randomly is not efficient for some applications. Imagine fuzzing MP3 files, it is extremely unlikely that random testing will generate a valid MP3 file. Because of that, inputs for model-less mutation-based fuzzers are seed-based. A seed is typically a well-structured input accepted by the PUT. By mutating the seed, it is possible to generate a test case that is accepted by the PUT, but contains abnormal values triggering crashes of the PUT. Variety of methods used to mutate seeds exist and [26] describes the following as the most common ones:

- **Bit-Flipping**. A number of flipped bits can be fixed, random, or user-configurable called *mutation ratio*. Flipping K random bits in N-bit seed is described by mutation ratio of K/N.

- **Arithmetic Mutation**. This mutation operation considers a selected byte sequence as an integer. The byte sequence is then mutated by performing a simple arithmetic operation on that value.

- **Block-based Mutation**. Block is a sequence of bytes of a seed. Block mutation can consist of inserting a randomly generated block into a random position, deleting a randomly selected block from a seed, or randomly permuting the order of a sequence of blocks.

- **Dictionary-based Mutation**. An example of dictionary-based mutation is usage of a predefined set $\{0, -1, 1\}$ when mutating integers, these values have significant semantic meaning for mutation.

White-box Fuzzers

White-box fuzzers can be either model-based or model-less, traditional dynamic execution does not require a model, while some symbolic executors leverage grammar-based input models to guide symbolic execution. Symbolic execution is slower than grey-box or black-box fuzzers as it instruments and analyzes every instruction of the PUT. A common strategy to cope with the high time complexity is to specify uninteresting parts of code. Guided symbolic executors involve a costly program analysis followed by a test case generation with guidance from the analysis. Other white-box fuzzers patch the PUT to bypass validators. Once they find a test case causing a crash, they try to reconstruct the failure on the original PUT with symbolic execution [26].

2.5.4 Evaluation

Input evaluation is a process of deciding what to do with the resulting execution, if we hit a bug, program crash, and which bugs are related. For example getting segmentation fault is a program issue that is easily trapped by a fuzzer, however, other types of memory bugs, such as stack buffer overflow can cause invalid result without a program crash.

In this section we present the most common techniques for test run evaluation. Bug oracle serves as an adviser for automatic bug type detection. Detected bugs then need to be analyzed.

Bug Oracles

Researchers have proposed a variety of efficient program transformations to detect unsafe program behaviors, called *sanitizers*. Sanitizers can be used to detect use-after-free vulnerabilities, capture cross-site scripting (XSS) and SQL injection vulnerabilities, or other information leaks [26].

Triage

Triage, based on [26], is a process of analyzing and reporting test cases that result in vulnerabilities. It is separated into three steps:

1. **Deduplication**, is a process of pruning test a case that triggers the same bug as a different test case did. The ideal state after deduplication is to have a single test case per unique bug.

2. **Prioritization** is a process of ranking or grouping test cases resulting in bugs according to their severity and uniqueness. In the context of memory vulnerabilities, prioritization is often called *exploitability* of a crash. Informally, it describes the likelihood of practical exploit development.

3. **Minimization** of test cases is a part of triage identifying the portion of a violating test case that triggers a vulnerability. The goal is to produce a test case that is smaller and simpler but still hitting the original issue.

2.5.5 Configuration Updating

The `config_update` function's behavior is different for black-box, grey-box, and white-box fuzzers. Black-box fuzzers typically leave the configuration set unmodified, as they are only evaluating a bug oracle. In contrast, grey and white-box fuzzers typically have a complex `config_update` function [26].

Evolutionary Seed Pool Update

The concept of Evolutionary Algorithm (EA), a heuristic-based approach involving biology-inspired evolution mechanisms, such as mutation, recombination, and selection, forms the bases of many grey-box fuzzers. Most EA-based fuzzers use node or branch coverage as the fitness function to add a new configuration to the set of configurations. If a new node or branch is discovered, a new test case is added to the seed poll. Common strategies are to refine the fitness function to detect more granular indicators of improvements, or to measure the fraction of conditions that are met when branch conditions are evaluated [26].

Maintaining a Minset

The risk of creating too many configurations rises with the ability to create new fuzz configurations. Solution to this problem is to maintain a *minset*—a minimal set of test cases that maximizes a coverage metric, similarly as it is used during the `preprocess` phase [26].

2.5.6 Fuzzer Quality

Quality of a fuzzer can be determined by numerous different metrics and it depends on the concrete use case why fuzzing was conducted. Metrics of quality can be:

- **Speed**. Possibly one of the most important factors. Intuitively, more test cases per second, more scenarios tested, more bugs found.
- **Vulnerabilities found**. Some fuzzers may be slower but still can find more bugs thanks to advanced test case generation.
- **Code coverage**. How much of a program was tested during fuzzing.
- **Test case minimization**. Fuzzers that report minimal and unique test cases for unique bugs are better.
- **Crash categorizing**. A person conducting fuzzing has immediate information on which bugs have higher severity and what type of bugs are found.

A good fuzzer should satisfy all the factors mentioned above. It should be reasonably quick, while intelligent enough to generate not trivial test cases able to find vulnerabilities. The fuzzer should test as many code lines as possible and report minimal and unique test cases categorized by exploitability of found vulnerabilities.

Chapter 3

REST API Fuzzing

Fuzzers for the HTTP protocol exists and are used, but to fuzz test a REST API, it is better to have a more REST-specific fuzzer to do the job. The most popular fuzzers, like AFL, are not suitable for fuzz testing of a service that may run as multiple communicationg microservices, especially it is not possible to test one microservice without data from another. This is one of the reasons why fuzz testing of Web APIs and especially REST APIs is a complex task [4]. For testing such services we can use black-box grammar-based fuzzing based on REST API specification formats. In fact, it is nothing new and many approaches for specification-based test case generation exists, but mostly for SOAP web APIs relying on Web Services Description Language (WSDL) documents. On the other hand, research targeting REST API fuzzing is rather limited despite the fact that its potential of discovering new vulnerabilities in REST APIs is high, observing the success of popular fuzzers used for other applications.

In the following sections, we describe REST API and look at how the API can be specified and documented. We won't miss a security point of view of REST API and outline which security issues can be found by fuzzing. In the end of this chapter, we assess the current research regarding REST API fuzzing and look into some tools for testing and their suitability for fuzz testing.

3.1 REST API

Representational State Transfer (REST) is the name of a description of the *Web's architectural style*. The key abstraction of information in REST is a *resource*. The resource is any information that can be named: a document, a temporal service, or a collection of other resources. REST is composed of the following constraints [14]:

- **Client-Server**. The Web is a client-server based system. Portability of the user interface across multiple platforms and the scalability is improved. The separation also allows the components to evolve separately.
- **Uniform Interface**. The interactions between clients, servers, and network-based intermediaries depend on uniform interfaces. It has four constraints:
  - *Identification of resources.* Every resource should be addressed by unique identifier, such as URI. For example, *http://example.com* is a unique identifier for a specific website's root resource.

  - *Manipulation of resources through representations.* The same resource can be represented to different clients in different ways. For example, a document can be represented as a JSON to an automated program, and as an HTML to a web browser.
  - *Self-descriptive messages.* The desired state of a resource can be represented within a client's request message. The current state of the resource can be represented within the response message. These messages may include metadata for additional details regarding the resource state.
  - *Hypermedia as the engine of application state (HATEOAS).* Links to related resources are included in resource state representation. For example, it is a link to another item in a collection, or more specifically, to the next page of results. This approach allows to traverse information.

- **Layered System**. This constraint enables intermediaries, such as proxies, to be transparently deployed between the server and the client.

- **Cache**. Caching can help in reducing client-perceived latency by cacheability of each response's data.

- **Stateless**. A web server is not required to memorize the state of its client applications. This trade-off is a key to the scalability of the Web's architectural style.

- **Code-On-Demand**. It enables web servers to temporarily transfer executable programs to clients. This constraint is optional.

REST API is a Web API (application programming interface) conforming to the REST and consisting of an assembly of interlinked resources. A web service utilizing a REST API is called *RESTful* and the set of resources is known as the *REST API's resource model* [27].

Uniform Resource Identifiers (URIs) are used to address resources of REST APIs. The definition for a URI must conform syntax defined by RFC 3986 [3].

REST API is using HTTP methods and HTTP response status codes to inform the client about the result of the called API. Numerous guidelines on how to use the HTTP methods exist and some of them are presented in the following subsection. However, all of these guidelines must comply with HTTP standard [15]. The *safe* `GET` method should not change the state of the resource and *idempotent* `GET`, `PUT`, and `DELETE` methods should result in the same state change of the resource when applied multiple times to a resource.

In this thesis we distinguish terms **endpoint** and **resource**. The term *endpoint* describes HTTP method and URI used to make a request. For example, `GET /systems/{id}`, while `{id}` is a *path parameter*. The term *resource* is a named information returned by the *endpoint*, for example, a JSON document describing a system with the attribute `id=42` returned by the endpoint `GET /systems/42`.

3.1.1 Description Specification

To provide a specification and documentation of a REST API's endpoints and resources to a user, we can utilize many different technologies. Ideal example of a general specification is JSON Schema[1]. An example of more specific, and nowadays widely used descriprion format is OpenAPI[2]. Such specifications contain information about URI's of the resources,

[1] JSON Schema - https://json-schema.org/

[2] OpenAPI - https://swagger.io/specification/

which HTTP methods are accepted by different endpoints, what is the expected input for an API call, and what will be the output, including the status code. Generally, these tools do not define only syntax of the API, but also the semantics of each resource. Authors of the description standards usually come up with time- and experience-proved guidelines for creating REST API.

Naturally, a description of the API is not used only to be readable by users, but mainly by computers. If the specification itself is machine-readable, it gives us the ability not only to show the information nicely to the user, but, for example, to generate clients based on the specification or even to generate part of the server handling the API interface with automatic input validation. In this thesis, we focus on generating test cases based on the specification of REST API.

JSON Schema can be used for REST API specification, however, there exist other languages to describe REST API, too. Sorted by their popularity measured by stars on GitHub, OpenAPI is by far the most popular specification for REST API with more than 16 thousand stargazers. This is the reason why so many frameworks and tools simplifying development of applications described by OpenAPI specification exists. Another popular specification with lots of tooling, called API Blueprint, has almost half of the OpenAPI's popularity, with 7.8 thousands stars. The last complex specification with a number of tools easing the whole API design lifecycle is the RAML specification (3.6 thousand stars).

JSON Schema

JSON has been widely adopted by HTTP servers for automated APIs. To enhance the processing of JSON documents in a RESTful manner, a comprehensive standard for description of any JSON data, called **JSON Schema** was proposed. The drafts include JSON Schema Language [7] and JSON Schema defined media type `application/schema+json` [33] to assert what a JSON document must look like and how to interact with it. It is necessary to mention that JSON Schema is still in a work in progress Internet-Draft state.

OpenAPI

OpenAPI is a broadly adopted industry standard for describing modern APIs. It defines a programming language-agnostic interface description for REST APIs allowing humans and computers to discover capabilities of a service. The OpenAPI specification does not require a specific, design-first development process. Data models (schemas) and data types are based on an extended subset of JSON Schema specification.

The specification allows users to describe an entire API, including:

- Available endpoints and operations on each endpoint
- Operation parameters input and output for each operation
- Authentication methods
- Contact information, license, terms of use, and other information

The format of the specification can be either JSON or YAML, while YAML is recommended. However, it must meet some additional constraints:

- Tags must be allowed by JSON Schema ruleset.
- Keys used in YAML maps must be limited to a scalar string.

API Blueprint

API Blueprint[3] language is based on the Markdown format. Thanks to this format, it might be the easiest for newcomers to understand. It is created for quick prototyping and modeling of APIs or for describing already deployed APIs.

To ease the API design lifecycle and encourage dialogue and collaboration between project stakeholders, API Blueprint comes with many useful tools[4]. Some tools, such as plugins to popular code editors, are developed to simplify writing of a blueprint, others create a mock server that implements the API Blueprint specification, record HTTP communication in the API Blueprint format, generate API Blueprint from request specs, render HTML documentation, or ensure that API documentation is not outdated by testing the server if it reacts according to the specification.

RAML

RESTFul API Modeling Language (RAML) is a YAML-based, human-readable language used for specification of REST API. RAML's great advantage is the ability to document APIs that do not adhere to all REST constraints. While OpenAPI is better suited for creating a specification of an existing API, RAML is focused to make the whole API lifecycle easy, from design to sharing.

Different tooling exists for APIs specified by RAML. It includes tools that can visualize what an API looks like, tools for prototyping, or frameworks for rapid development of applications that expose RAML API. There are also tools to verify API documentation written in RAML format against its back-end implementation and tools to create HTML documentation from RAML specification.

3.1.2 Security

In order to make an API secure, it is necessary to think about many areas that could be vulnerable. The Open Web Application Security Project(OWASP) Foundation[5] provides best practices on how to make a REST API secure. In the following text of this section, we describe security guidelines from OWASPs' REST Security Cheat Sheet [25] with regard to the ability of fuzz testing to find related issues.

Access Control

Access control at each API endpoint is needed for non-public REST services, to avoid problems with unauthorized use of REST endpoints that allow changing database entries.

OWASP's suggestion is that the access control decision should be taken locally by REST endpoints to minimize latency and reduce coupling between services, and access tokens should be issued by a centralized Identity Provider.

Fuzzing can be used to test various endpoints that should be used only by authorized users, in particular, it can test that it is not possible to use these endpoints without authorization.

[3] API Blueprint - https://github.com/apiaryio/api-blueprint

[4] API Blueprint tools - https://apiblueprint.org/tools.html

[5] The Open Web Application Security Project (OWASP) - https://owasp.org/

Restrict HTTP Methods

A suggestion is to create a whitelist of permitted HTTP methods, for example `GET, POST, PUT` and reject all requests not matching the whitelisted HTTP methods with the response code *405 Method not allowed*. It is also needed to verify that the caller is authorized to use the incoming HTTP method on the resource collection, action, and record.

Not using whitelists may result in usage of an HTTP method that should have been disabled, causing an unexpected behavior, such as, deleting a resource. This should be caught by fuzz testing of the endpoints with different HTTP methods.

Input Validation

Input validation issues are probably the most common issues found by fuzzing. Due to insufficient validation, some inputs may cause server crash or information leakage.

OWASP defines the following rules to deal with input validation:

- Do not trust input parameters/objects.
- Validate an input length, range, format, and type of the input.
- Use strong types like numbers, booleans, dates, times, or fixed data ranges in API parameters to achieve an implicit input validation.
- Constrain string inputs with regular expressions.
- Reject unexpected or illegal content.
- Make use of validation/sanitation libraries or frameworks in your specific language.
- Define an appropriate request size limit and reject requests exceeding the limit with HTTP response status 413 Request Entity Too Large.
- Consider logging input validation failures. Assume that someone who is performing hundreds of failed input validations per second is up to no good.
- Have a look at the input validation cheat sheet for a comprehensive explanation.
- Use a secure parser for parsing the incoming messages. If you are using XML, make sure to use a parser that is not vulnerable to XXE and similar attacks.

Validate Content Types

The request and the response body should match the intended content type in the header. Misinterpretation at the consumer/producer will cause confusion for a user and it can lead to code injection/execution.

The solution is to document all supported content types in API, reject requests containing unexpected or missing content types. XML content types should use an appropriate XML parser to avoid XXE.

Fuzzing focused on trying to send different content with different content types should find issues leading to code injection/execution.

Management Endpoints Exposure

Management endpoints are endpoints used for maintenance of an application, and usually, their use is not intended for a regular user. Recommendations for securing management endpoints are following:

- Avoid exposing management endpoints via Internet.
- If management endpoints must be accessible via the Internet, make sure that users must use a strong authentication mechanism, for example, multi-factor authentication.
- Expose management endpoints via different HTTP ports or hosts preferably on a restricted subnet.
- Restrict access to these endpoints by firewall rules or use of access control lists.

To test that management endpoints are secure, we can utilize fuzzing. If management endpoints are exposed via Internet, fuzzing should find these endpoints and use them for management of the service.

3.2 Assessment of the Current State of REST API Fuzzing

In this section we closely look at the current research of REST API fuzz testing and how the approaches differ. Some open-source and commercial tools offering automated REST API test generation exists, but the most of them have a problem that they only do testing using correct data to verify that documentation is not outdated, but they lack the negative testing that might uncover security problems [12].

3.2.1 Related Work

Even though fuzz testing of REST API is not deeply researched, we can find different approaches to test generation. Some authors follow black-box approach relying on manual definition of a model, others are trying to generate tests using a white-box approach utilizing search algorithms.

Chakrabarti and Kumar [8] proposed an approach to test the REST API in their tool called Test-the-REST. It is a black-box, specification-based testing that relies on test case definition in XML format that has to be written manually. The manual model definition is the main downside of their proposal, despite that, they were able to find a lot of bugs on a daily basis. Another model-driven approach was proposed by Fertig et al. [13]. They were influenced by Chakrabarti and they also created an approach with the need of manual model definition using Domain Specific Language (DSL), but with less knowledge about testing. It required to define REST resources using DSL. A similar approach was used by Earle et al. [11]. They created a library to generate test cases from a JSON Schema characterized data enabling QuickCheck[6] state machine to generate different JSON data. A web service is tested by following the links in the JSON Schema. Again, this is an approach that needs manual model definition.

In contrast to the previously mentioned proposals, Arcuri [1] came with a fully automated approach that uses search algorithms to white-box test the API relying on OpenAPI

[6] QuickCheck - https://hackage.haskell.org/package/QuickCheck

specification. While this is a very powerful tool, it has a downside of the need for full access to the code to generate tests.

On the other hand, other authors proposed fully automated black-box tools. The approach of Ed-douibi et al. [12] consist of four steps. First, they extract an OpenAPI model from the definition document, by processing the JSON (or YAML) file. The second step is very valuable for future test generation. From an OpenAPI model, they extract parameter examples and use them later as the input data for test cases. Thanks to this step, they do not have a lot of generated date that results in 404 response code due to accessing not existing resources. Then, they create a TestSuite model and, finally, transform the TestSuite model into an executable JUnit code. Unlike others, they generated nominal tests validating the definition document, as well as fault-based tests. Sadly, in their proposal, they are not mentioning bug triage, deduplication, prioritization and minimization of test cases.

Subjectively, one of the best papers researching REST API fuzzing was proposed only recently by Atlidakis and Polishchuk in collaboration with fuzzing veteran Patrice Godefroid [2]. They introduced the first stateful REST API fuzzer. The process of fuzzing is similar to Ed-douibi's approach. The main difference is an automatic dependency inference among request types and a dynamic generation of tests guided by feedback from service responses. The authors showed necessary techniques for effective stateful fuzzing by experimenting with different strategies for searching the large search space combined with dependency inferring and dynamic feedback avoiding dependency combinations refused by the service. After testing they did not forget to use bucketization scheme to cluster similar bugs and avoid redundancy.

Inspired by Atlidakis and QuickCheck, Karlsson et al. created a tool called QuickREST [22]. It is generating stateful tests based on an OpenAPI specification by using Clojure Extensible Data Notatation[7] format. However, their tool is just a proof-of-concept, set of scripts [21] written in Clojure, tested against Gitlab[8] API.

3.2.2 Existing Tools

For different specification format, different testing tools exist. If we want to automate testing of API Blueprint specified API, we can use Apiary[9]. For RAML format, we can use Abao[10]. Ready API[11] can be used for OpenAPI format. Tools like Dredd[12] or API Fortress[13] support more specification formats and they can be used with all mentioned API formats. However, all of these tools share a common drawback, they are primarily created to test that the application complies with the schema and they are not generating negative tests.

One tool overcoming this drawback and testing the REST API defined by OpenAPI format is Schemathesis[14]. It automatically generates test cases based on OpenAPI specification. After parsing the schema and inferring the data types, the Hypothesis[15] project

[7] Extensible Data Notation (EDN)—https://github.com/edn-format/edn

[8] Gitlab - https://about.gitlab.com/

[9] Apiary - https://apiary.io/

[10] Abao - https://github.com/cybertk/abao

[11] Ready API - https://smartbear.com/product/ready-api/overview/

[12] Dredd - https://dredd.org/en/latest/

[13] API Fortress - https://apifortress.com/

[14] Schemathesis - https://github.com/kiwicom/schemathesis

[15] Hypothesis - https://hypothesis.readthedocs.io/en/latest/

is used to generate test data based on the data type of fuzzable parameters. Test data generated by Hypothesis consist of valid inputs and invalid inputs and, thanks to this, Schemathesis is generating also negative test cases. Hypothesis is also used to evaluate results and find minimal falsifying input. Schemathesis is rapidly developing open-source tool that is gaining popularity, however, at the time of writing this thesis, it lacks opportunities such as specifying the valid input (e.g. id of a resource) which leads to creating a lot of test cases resulting in 404 response code, or stateful dependency-based testing.

Chapter 4

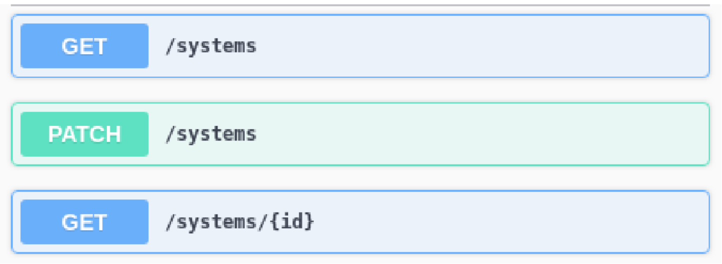
Design of REST API Fuzzer

Knowing the usage, specifics, limitations, and possible security issues of REST API, we can tailor fuzzer design to the REST API testing needs. Design of our proposed REST API fuzzer follows algorithm of a general fuzzer shown in Listing 2.1 with modification specific to REST API testing. Our main goal is to solve the problem of not exercising deeper states of the tested application that occurs because the generated invalid data have to be valid enough to pass application's input sanitizing. The solution is in creating a **stateful fuzzer** that will gather the inputs needed for required parameters of a tested endpoint. We demostrate the proposed design of our stateful REST API fuzzer on endpoints of a testing application thath is introduced in detail in Section 4.1.

In Section 4.2 we discuss a high level design of the REST API fuzzer. The rest of the sections describe every phase of the fuzzer in more detail. An exception is the *Execute* phase, which is not described, since it only features execution of test cases and provides feedback for scheduling and configuration update algorithms.

4.1 Running example

During the design, implementation, and evaluation phases of the thesis, we demonstrate the proposed concepts on a testing application. The testing application contains one, intentionally created stateful bug. The application has three endpoints as seen in Figure 4.1.



GET /systems

PATCH /systems

GET /systems/{id}

Figure 4.1: Test application

The purpose of the testing application is to list systems, their attributes, and to change system's display `name` and Fully Qualified Domain Name (FQDN), `fqdn` attribute.

The endpoint `GET /systems` lists all systems and their attributes. In a stateful testing, it will be used to obtain the required `id` parameter needed by other endpoints. The

`GET /systems/{id}` endpoint returns the FQDN of a system specified by its `id`. Finally, the endpoint causing the stateful bug is the `PATCH /systems` endpoint modifying a system resource. It is used to change system's `name` and `fqdn` attributes of the resource specified by the `id` parameter.

The bug is caused by a *typo* while modifying in-memory database, changing `fqdn` parameter to `fqnd`. Even though the `PATCH /systems` is causing an issue, it will return 200 status code. The problem appears, when `GET /systems/{id}` wants to access the incorrectly modified resource. Thus, the issue can be found only by a stateful testing.

4.2 High Level Design

Many tools and researches for testing REST APIs discussed in Section 3.2 have one common downside. They are testing that the application conforms to its schema by creating valid inputs for the application but they are not testing invalid inputs for the input schema. This is the first issue that we want to address with our proposal—**testing invalid inputs**. The next issue of the existing tools is that none of them is trying to get the valid input for some of the required input parameters. This leads to a lot of inputs rejected by the service. Some researchers have focused on this problem and their solution is in using example values specified by the endpoint's input schema [12] or by finding endpoints providing necessary inputs [2]. Example values are good for using real strings or integers accepted by the service, but if the required input is the *id* of the resource, an example value is not necessarily an *id* of an existing resource and the request will be rejected by the service. **Dependency inference** is a better solution in this case, and therefore we combine these two approaches. Another problem tightly connected to the previous one in REST API testing is the exploration of deeper states of the application which can be also solved by inferring dependencies and **chaining tests of endpoints**. Th last observed problem is connected with using input values from dependencies. We do not always want to use all the gathered values. Some parameters may be optional and some required parameters can be fuzzed while the request is still accepted by the service. **Fuzzing parameters** not resulting in rejection of the request can be another improvement in REST API testing.

To clearly illustrate the problems consider extending the testing application by the endpoint shown in 4.1. The endpoint is changing the Fully Qualified Domain Name (FQDN) of a resource. It has two required parameters, the system's *id* and the wanted *fqdn*. The system's name can be derived from its FQDN, but the user can specify some other name for the system, thus a *name* parameter is optional. If we are going to generate values for all required parameters, we will always get 404 HTTP status code for accessing non existing endpoint, since the *id* format is more complex than a single number. Using the example from the schema won't almost certainly mean any difference, because the example *id* won't exist. Getting the real system *id* from other endpoint will finally result in 200 status response, but we might have used valid inputs for both *fqdn* and *name* parameters and therefore not testing anything at all. Generating values for the required *fqdn* parameter and creating requests with an omitted or a generated *name* parameter will result in a better testing of the endpoint. Finally, if we want to examine the application deeper, we can execute tests on endpoints using the same resource to test possible scenarios and detect how the tested endpoint reacts to possible changes made to the resource. Doing these steps is our key how to properly test a REST API.

The first change that narrows down the general algorithm is that our fuzzer is a **black-box fuzzer**. The expected use of the fuzzer is in Software as a Service environment, where

```
'/systems/{id}':
    parameters:
  ---name: id
        required: true
        example: 03708698−7921−11ea−b755−48a4720be785
    body_parameters:
  ---name: fqdn
        required: true
  ---name: name
        required: false
```

Listing 4.1: Endpoint for problems illustration

the application consists of multiple micro-services, and we are going to test their public API. This is the reason, why we cannot collect any data about the tested application neither by static or dynamic code analysis. The *preprocess* phase won't contain any PUT instrumentation or generation of a driver application. Its main goal is to determine neccessary information by parsing the REST API specification. After this phase the fuzzer knows all endpoints of the application, their input parameters, data types of these parameters as well as responses of the endpoints. Basically, the preprocess phase prepares a model of the application for future input generation. We cover this phase of the general fuzzer algorithm in sections 4.3 and 4.4.

Scheduling, *input generation*, *evaluation* and *configuration updating* are next steps that need a specific design for stateful testing of REST API. In the scheduling phase, we need to solve exploration vs. exploitation problem. We have to decide which endpoints we want to test, in which order they should be tested, how many tests should be performed on a single endpoint, and whether to continue with the testing of the current endpoint, move to the next one, or return to an already tested endpoint. Since our fuzzer is a black-box fuzzer, the only information guiding the scheduling algorithm is the outcome of the current configuration. The outcome consists of the HTTP status code of the API call, the endpoint response, the time spent in the test, the number of crashes, and the stateful information gathered from outcomes of the previous tests.

Once we know which endpoint is to be tested, we need to generate inputs for the fuzz configuration. Our approach can be considered as a **generation-based fuzzer** with a model inferred from schema of the application during the preprocess phase. Model based generation will create an input schema for the tested endpoint and the values for data in the schema are generated based on their data type.

After the execution of a fuzz configuration, we need to evaluate and test the outcome and update the configuration set. While most of the black-box fuzzers leave configuration set unmodified and they are only evaluating the bug oracle, we modify the generation model based on the outcome of the previous test. The model is modified after each fuzz configuration to create better and more relevant inputs for the following tests. The evaluation phase deals with detecting test failure based on HTTP status code and endpoint response as well as on minimization of test cases and their deduplication.

A simplified overview of the proposed stateful REST API fuzzer is displayed in Figure 4.2.

We discuss the design of the *scheduling*, *input generation*, and *configuration updating* phases of our fuzzer further in Section 4.5. The *Evaluation* phase of the REST API fuzzer,

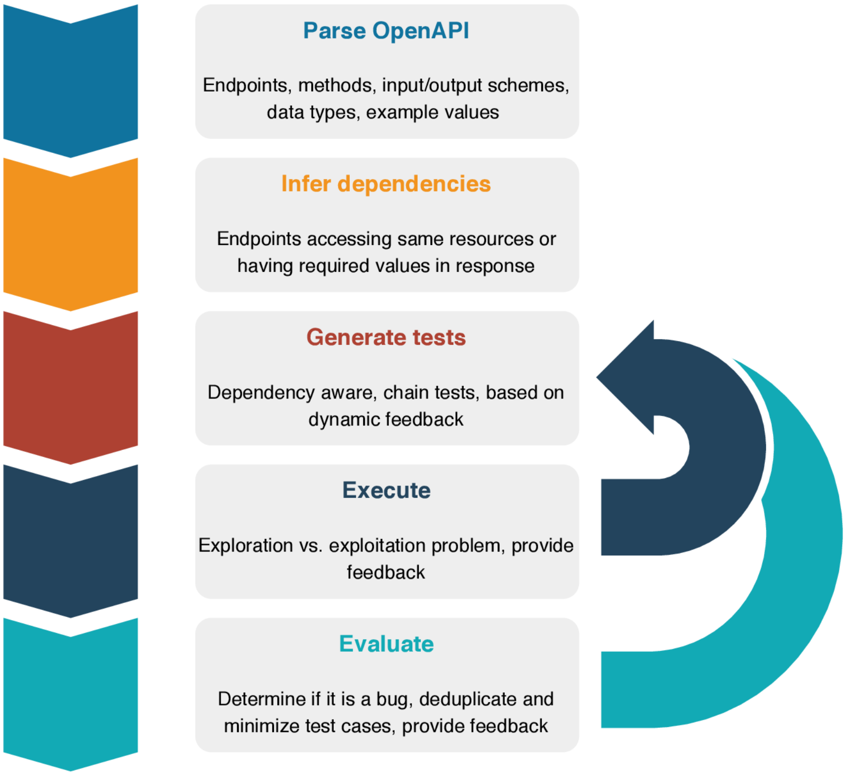

Parse OpenAPI
Endpoints, methods, input/output schemes, data types, example values
Infer dependencies
Endpoints accessing same resources or having required values in response
Generate tests
Dependency aware, chain tests, based on dynamic feedback
Execute
Exploration vs. exploitation problem, provide feedback
Evaluate
Determine if it is a bug, deduplicate and minimize test cases, provide feedback

Figure 4.2: Stateful REST API fuzzer design

dealing with test minimization, deduplication, and failure detection is addressed in Section 4.6.

4.3 Parsing OpenAPI

The first task of a black-box fuzzer is to obtain a model of the application for future input generation. In case of the REST API, the model will contain list of available endpoints, their input parameters and output schema. Due to the large popularity of OpenAPI specification description format and its vast usage among other formats, we have decided to support APIs described by OpenAPI and its predecessor, the Swagger format. An example of endpoint definition by OpenAPI schema is shown in Listing 4.2 From the schema above, we can process all the necessary data needed for creating a testing model. The model will consist of schemas for each endpoint. The process of obtaining the data from the schema is further described in following the subsections.

Processing endpoint's schema is quite simple and straightforward task thanks to the way the endpoint is defined in OpenAPI. Firstly, we need to save the same endpoint entry with different HTTP methods as separate endpoints. That said, `GET /systems/{id}` and `POST /systems/{id}` endpoints will be treated separately. For every endpoint we can then store multiple values:

- Base URL for application's endpoint
- Input schema for path parameters

```
'/systems/{id}':
  get:
    deprecated: true
    description: Get a FQDN of a system
    parameters:
  ---name: id
        description: System id
        required: true
        schema:
          type: string
        in: path
        x-example: 03708698-7921-11ea-b755-48a4720be785
    responses:
      '200':
        description: OK
        content:
          application/json:
            schema:
              type: object
              properties:
                id:
                  type: string
                fqdn:
                  type: string
```

Listing 4.2: Endpoint definition in OpenAPI schema

- Input schema for request body
- Input schema for form data
- Input schema for query parameters
- Schema for input with example values
- Response schema
- List of parameters in response
- List of parameters that are required

Storing these values for every single endpoint is necessary for obtaining a list of dependent endpoints and for the subsequent test cases creation.

4.4 Inferring Dependencies

Before we start with inferring dependencies from an API schema, we have to determine *what should be considered as a dependency of a tested endpoint.* In the first place, the dependency is a value for a parameter required by the endpoint's schema. Thus, it is a knowledge needed to make an API call accpeted by the tested service. Therefore, a dependency can be found in the response of other endpoints described by the API schema. However, endpoints providing information that is needed for by some following endpoint are not the only endpoints that could change the behavior of the tested endpoint. We have to also focus on endpoints accessing the same resource as the tested endpoint.

With respect to the above, in order to get dependencies of an endpoint, we need to create a list of dependent endpoints consisting of:

- Endpoints returning one of the required parameters for the tested endpoint in their response schema.
- Endpoints having the same required parameters as the tested endpoint.

To get an insight what are the dependencies for the endpoint `GET '/systems/{id}'` specified by the example schema in Listing 4.2 of our testing application, let us show the example of dependency inference. First, we obtain the required parameters from the schema. There is only one required parameter, called `id`, denoting the system's UUID. This parameter is found only in a response of the `GET /systems` endpoint, hence the `GET /systems` endpoint is our first dependent endpoint for the requirement `id`. There is one other endpoint in the list of application's endpoints and we see that the `PATCH /systems` endpoint has the parameter `id` as a requirement for its request body, thus it can affect the tested endpoint since it modifies the same resource. The list of dependent endpoints therefore consists of `GET /systems` and `PATCH /systems` endpoints.

4.5 Creating Test Cases

Test cases creation is an abstraction of multiple phases of the general fuzzer algorithm shown in Listing 2.1 (in particular of scheduling, of input generation, and of configuration update). During the scheduling phase, we select a fuzz configuration for the current iteration and solve the exploration vs. exploitation problem while relying on the output of the preprocess phase and on the current state. The input generation phase generates scheduled tests, schema of JSON input, as well as values for each parameter in the generated schema. Then, after execution of the test case, we need to update all fuzz configurations and store the current state for future test generation.

4.5.1 Scheduling

Black-box fuzzer does not have a very rich set of configuration information that can be used for scheduling. The absence of metrics such as code-coverage leaves us only with information that the fuzz configuration scheduling algorithm gets from the fuzzer configuration outcome, the number of crashes, or the time spent on configuration. These are the information that we have to rely on when solving the exploration vs. exploitation problem.

The exploration vs. exploitation problem is commonly known as multi-armed bandit problem. In case of REST API testing, the problem can be considered as restless multi-armed bandit, because executing tests against one endpoint can affect different endpoints. For example, one endpoint call can modify or delete a resource and the following endpoint call may result in a failure or can be rejected by the services as the resource does not exist anymore. Restless multi-armed bandit problem is known to be PSPACE-Hard [20]. However, we are not going to find an optimal solution of the problem, but we are going to apply the known constraints of REST API testing with regard to the total test duration. The goal is to test deeper states of the PUT by testing various combinations of endpoints, while maximizing the number of meaningful tests for a single endpoint. Our exploration vs. exploitation problem can be broken down into the following problems:

1. The order of endpoints to be tested.
2. The decision whether to continue with the current endpoint testing or to move to another endpoint tests.

The order of test cases. First of all, we need to clarify which endpoints should be tested. We are creating a stateful fuzzer, therefore, we want to primarily test all endpoints having some dependencies. However, we also want to test endpoints that do not need any input data from other endpoints. Before determining the order of the tested endpoints, we need to explain types of endpoint that we have:

- A *Target endpoint* is an endpoint defined in the API specification that we want to primarily test.
- A *Dependent endpoint* is an endpoint providing example values for required parameters of the target endpoint or an endpoint modifying a common resource.

The target and the dependent endpoints belong to two categories of endpoints:

- *Endpoints without dependencies.* These endpoints do not depend on any other endpoint and are not modifying resources accessed by other endpoints. Outputs from these endpoints can be useful as input for endpoints tested later on and they can be tested right away. An example of such endpoint is an endpoint listing `id`s of resources.
- *Endpoints with dependencies.* These are the endpoints having required parameters. The endpoint cannot be tested alone, thus we create a set of *dependent* endpoints which should be tested prior to the endpoint itself.

With respect to the above, a single **test case** must specify the following information:

- a single *target endpoint* `e` that is primarily tested,
- a set of *dependent endpoints* of `e`,
- a schema of the input, and
- values of input parameters (which are the input parameters of `e` and the input parameters of the dependent endpoints).

Every target endpoint has its own set of dependent endpoints and they are tested prior to the target endpoint. Testing dependent endpoints has two objectives. The first one is to get valid inputs for the target endpoint, so that the API call to the endpoint won't be rejected. The second purpose of dependent endpoints is to create a test scenario to test deeper state of the application. The order of of testing of dependent endpoints could be random, however, we want to maximize the number of obtained required values and thus we prioritize dependent endpoints providing more values of required parameters in their response. In addition, we prioritize endpoints having less required parameters to avoid getting too many 404s for a dependency in the early phase of the test run. The exact algorithm that we use for ordering of tests of dependent endpoints is presented in Section 5.2.

Based on the information described above, we are executing tests for a single target endpoint defined in the API schema in the following order:

1. **Dependent endpoints**. Execute tests against an ordered set of dependent endpoints of the target endpoint, gather example values for required parameters or modify the tested resource.

2. **Target endpoint**. Test the target endpoint with values obtained from the dependencies. If the target endpoint does not have any dependencies or if there are no obtained values, endpoint will be tested either with the example values from schema (if they are defined) or with randomly generated (fuzzed) data.

Testing dependencies of dependent endpoints can result in an exponential number of tests, thus we test only direct dependencies. Moreover, we save all parameters that are required by some endpoint. Due to this, it may happen that a target endpoint tested later is tested more thoroughly than one tested earlier. Later tested endpoints have higher probability of having the values of required parameters saved in test run state. The same situation occurs for dependent endpoints for a single target endpoint since our ordering algorithm does not consider transitive dependencies and therefore it does not produce a total ordering. To achieve a uniform coverage of all endpoints, we can solve this problem by randomizing the order of the target endpoints among test runs and of dependent endpoints for a single target endpoint.

The decision whether to continue with the current endpoint is made based on multiple variables. For example, it depends on the reason why the current endpoint is tested, since testing of dependencies is different from testing a target endpoint.

The first factor is a limit of the number of tests created for every endpoint. We use fixed number of fuzz runs in combination with deprioritization of slower configurations. Every response to API call has to be fast enough so that it does not result in *504 „Gateway timeout“* HTTP status code. The gateway timeout will naturally work as a fixed amount of time for a test run that will deprioritize slower configurations. After exhausting the fixed number of test runs, we move to testing of the next scheduled endpoint. If the generation of input is too slow, we as well continue with the next endpoint.

The difference between the decision for dependent endpoints and the target endpoints is the number of successive 404s. We want to skip tests of dependency if we hit a threshold of successive responses with 404 HTTP status code. The threshold is used for dependent endpoints since they serve, in the first place, as endpoints providing values of the required parameters for the target endpoints. Hitting one 404 may be caused by an incorrect, randomly generated input, on the other hand, multiple 404s mean that the endpoint is rejected by the service and it is not going to provide any needed data. Therefore, we can skip the testing of the dependent endpoint before reaching the limit of fuzz runs and continue with the next dependency or with the target endpoint. On the other hand, hitting the 404 threshold for the target endpoint usually means that we do not have correct values of required parameters. Still, we want to continue with testing of this endpoint using randomly generated inputs to possibly find inputs resulting in a failure (since this is the main goal of fuzz testing).

4.5.2 Input Generation

Input will be generated just before execution of the test. This means that we can generate input based on feedback from the previous test runs. A generated input consists of two parts: the schema of input and the input for parameters inside the schema.

First, we generate the schema of the desired input. Generating schema is necessary because we do not want to include all possible path parameters, body parameters, and query parameters in every request. We need to create an input JSON that will consist of a subset of all possible parameters. For example, `GET /systems` can have a query parameter `name` that will show only systems with matching name, and a query parameter `uuid` filtering systems by UUID. Next, we want to test all possible combinations of these parameters. We need to consider three types of input parameters:

- **Optional fuzzable** parameter is a parameter defined as optional in the API specification. The value for this parameter will be always randomly generated and the parameter does not have to be used in an API call.
- **Required fuzzable** parameter is a parameter required by the API specification which won't cause a rejection by the service with a randomly generated input value. An example is `PATCH /systems` and its required `fqdn` parameter. A randomly generated value will successfully change the `fqdn` attribute of the tested system.
- **Required non-fuzzable** parameter is a required parameter by the API specification and generated value for such parameter will most probably result in the 404 status code. An example is the `id` parameter of the `GET /systems/{id}` of the testing application. Without the `id` value obtained from dependencies, the request will be rejected by the service.

The next step is generation of input values based on input data types from the input schema. Inputs will be generated randomly trying multiple values of the desired data type for the parameter, as well as invalid data type input. This is the phase where, finally, data gathered from dependencies will be used. If the parameter is a *required non-fuzzable* parameter, a value from a dependency will be used.

While generating the input schema, we need to think about which parameters can be fuzzed and which, if they are fuzzed, could result in rejection by the service. Both `name` and `uuid` query parameters of `GET /systems` can be omitted, and later on they can use a generated value and the request will be still accepted by the service. On the other hand, situation for `PATCH /systems` may be different. It can have multiple required body parameters, e.g. `name` and `uuid`. Even though the `name` parameter is a required parameter and it has to be present in the generated schema, the input value for the `name` parameter can be generated and the input will be accepted. On the other hand, a generated or an invalid input for the `uuid` parameter will be always rejected by the service. We automate the process of classifying parameters by maintaining a so-called *confidence score* of a parameter. Higher confidence means that the required parameter is *required non-fuzzable* parameter. The confidence score is further described in the following Section 4.5.3 as it is computed after a test execution, during the update of the model.

4.5.3 Configuration Updating

After execution of a single test, we want to assess its result and potentially update the model for a better input generation for following tests. We collect the state of the executed test, its status code alongside with the values from the JSON output that can be used as inputs for required parameters for different endpoints.

Stateful information will be reflected in the model, hence the following input generation phase will use values for required parameters obtained from the dependencies instead of randomly generating these values. It as well solves our problem with fuzzability of the required parameters. If the endpoint test with generated required parameter results in too many 404s, we can for sure say that the parameter is *required non-fuzzable* and can not be fuzzed. However, if the randomly generated value for a parameter is accepted by the service, then the parameter is very probably *required fuzzable*. A request to an endpoint with *required fuzzable* parameter resulting in 404 status code is caused by the endpoint having another parameter that is *required non-fuzzable* when the value for the *required non-fuzzable* parameter is invalid. This information is passed to input generation algorithm to properly generate schema and the data for the next API call.

Fuzzability of a required parameter is determined by the *confidence score* of the parameter. Confidence is higher with the increasing number of 404 responses and decreasing with 2xx status of the responses. Once the confidence reaches a configurable threshold, the parameter is considered as *required non-fuzzable* and its value is never randomly generated but obtained from the dependencies.

4.6 Test Evaluation

The detection if the result is a bug is fairly straightforward for a black-box fuzzer. There are two types of bugs detectable by a black-box fuzzer. Either the request takes too long or the HTTP status code of the response is 5xx. However, some status codes such as *503 Service unavailable* might not be caused by the bug, but the service can be under scheduled maintenance. The only 5xx issue that we can be sure about is the *500 Internal server error* status code that is caused by user input, because client errors should be handled by 4xx codes.

We provide the failing example as a minimized test case. If multiple tests for the same endpoint fail with the 500 status code, we minimize test case as follows. First, we create the minimal subset of the parameters used in failing tests. Then, we get a minimal, more readable, values of these parameters from the failing tests. The last step is to provide information about the previously tested endpoints. We provide a list of dependency endpoints executed before the failed test. Even if some previous target endpoint test could have changed the tested resource resulting in failure of the current test, this endpoint is then a part of dependencies of the current test too, and therefore it occurs in the endpoint' dependency list. This is the reason why we store only dependent endpoints and not target endpoints in the list of previously tested endpoints.

The same failures can be found during testing of multiple target endpoints, or even during the testing of their dependencies. To deduplicate such failures, we report only those having the same parameters and the shortest list of previously tested endpoints.

Chapter 5

Implementation

This chapter further describes how the designed features are implemented. We explain decisions made during the preprocessing phase, how we gather requirements for endpoints, and how we infer their dependencies. Next, we follow up with implementation details of selecting the current fuzz configuration. Then, we move over to input generation specific to the REST API fuzzer followed by the execution and evaluation of the test. Finally, we present details of how the configuration is updated, i.e. how we update the application model for better input generation in the following iterations.

Instead of implementing a REST API fuzzer from scratch, we decided to extend the **Schemathesis** project, an open-source tool for REST API testing. Schemathesis is written in Python and generates test cases from an OpenAPI/Swagger specification. For test generation, it utilizes property-based testing library named *Hypothesis*[1]. *Hypothesis-jsonschema*[2] is used for creation of a Hypothesis strategy that generates data matching a provided JSON Schema. The tests are executed by the *pytest* framework. Schemathesis allows us to run tests from its command line interface or using the parametrization of a pytest test. Schemathesis is gaining its popularity and is already used for fuzz testing capabilities in Red Hat's Insights QA internal framework and for testing Red Hat Insights services at `http://cloud.redhat.com`. Another reason for extending Schemathesis is that it already contains processing of OpenAPI specification and generating of test data, and therefore we may concentrate on adding a functionality for stateful testing.

Our implementation is extending Schemathesis by adding a stateful testing functionality. To create stateful tests, we need to modify every part of the Schemathesis, i.e. schema processing, models representing endpoints and test results, generation of tests, pytest and CLI runner, and failure reporting. These changes are discussed in the following sections organized as phases of a general fuzzer algorithm shown in Listing 2.1. The result of our implementation is a pull request[3] to Schemathesis GitHub repository.

5.1 Preprocess

Preprocessing in Schemathesis is based on conversion of an OpenAPI/Swagger specification into a JSON Schema needed by Hypothesis-jsonschema for input generation. Then, models for schema (`class BaseSchema`) and endpoint (`class Endpoint`) are created. `BaseSchema`

[1] Hypothesis - `https://hypothesis.readthedocs.io/en/latest/`

[2] Hypothesis-jsonschema - `https://github.com/Zac-HD/hypothesis-jsonschema`

[3] Schemathesis stateful testing pull request - `https://github.com/kiwicom/schemathesis/pull/520`

contains basic information from the JSON Schema such as the base URL or the raw JSON Schema with methods for obtaining all endpoints and later all tests for the schema. The `Endpoint` class holds information about a single endpoint, namely its path, method, definition and parameters.

In order to allow statefulness, we need to extend the `Endpoint` attributes by adding parameters tat can be modified in the future. Also, we added properties for obtaining a set of required parameters (`requirements`), a list of endpoints having a subset of required parameters in their set of requirements, and a hash table of dependencies consisting of a name of the required parameter and of a list of endpoints providing the required value.

The `BaseSchema` class was extended with `state` attribute, which is an instance of the `State` class consisting of attributes such as the previous result, the requirements and the number of successive 404 status codes. The `Requirements` attribute will be updated with every successful response, therefore it is implemented as a hash table where the key is the name of the required parameter and the value is an instance of the *Requirement* class. The `Requirement` class has a list of values for the required parameter and a confidence if the requirement is *required non-fuzzable* and the request will be rejected if it does not provide correct value. We will discuss fuzzability based on the confidence attribute further in Section 5.3.

5.2 Scheduling

Scheduling algorithm follows the design described in Section 4.5. However, due to usage of Schemathesis, we are limited to generate test cases endpoint-wise. We can only execute tests for a single endpoint and then continue to another one. The approach of Schemathesis does not allow to alternate tests of multiple endpoints.

The `BaseSchema` class is responsible for generating all tests in `get_all_tests` generator. Test generation works as follows.

1. Iterate over all endpoints
2. Get all dependencies for the currently tested endpoint, extend the set of dependencies by endpoints having a subset of the tested endpoint's requirements.
3. Sort these dependencies by two keys. The first key sorts dependencies ascending by the number of required values, second key sorts them descending by the dependency count. Thanks to the sorting, dependency having the least number of required parameters and providing values for a longest number of other endpoints is used first.
4. Yield test case for the current dependency
5. After all dependencies are tested, yield test case for the target endpoint with example values for required parameters obtained from the dependencies.

The decision whether to continue with the testing of current endpoint or to continue with another is made after the execution. We monitor the HTTP status codes reported by each test and when the number of successive 404 status codes for a single endpoint hits a threshold, we continue to the next endpoint. This is handled by the CLI runner. If the user wants to achieve the same functionality by executing tests from pytest, he can easily create his own counter for successive cases or use `update_case` and `_should_skip_case` from `schemathesis.runner.impl.core`. The usefulness of this approach for test scheduling is demonstrated on code coverage of a tested application in Chapter 6.

5.3 Input generation

Input generation is implemented using *hypothesis-jsonschema* library. As it is evident from its name, it uses JSON Schema to generate a *Hypothesis* search strategy. The hypothesis search strategy is an object describing how to generate certain kind of input values. In our case, it generates the `Case` object containing basic parameters to make a HTTP request including the input needed by the service. It can generate inputs with different parameters and corresponding random values. For illustration, take a look at the request body schema in Listing 5.1 and the generated Schemathesis Case in Listing 5.2. Schemathesis generated random unicode string inputs for all required parameters in the request body, left out a optional parameter, and added one extra, randomly generated parameter that is not defined in the schema.

```
{
    "properties": {
        "fqdn": {
            "type": "string"
        },
        "id": {
            "type": "string"
        },
        "name": {
            "type": "string"
        }
    },
    "required": [
        "id",
        "fqdn"
    ],
    "type": "object"
}
```

Listing 5.1: Request body schema

```
Case(
    endpoint=Endpoint(
        path=’/systems’,
        method=’PATCH’,
        base_url=’http://localhost:8080’,
        ...,
    ),
    path_parameters=None,
    headers=None,
    cookies=None,
    query=None,
    body={
        ’fqdn’: ’\x06\U0004bdd9\x12’,
        ’id’: ’’,
        ’\x13\U0009985d’: [True],
    },
    form_data=None
)
```

Listing 5.2: Schemathesis Case

To achieve stateful generation of inputs based on previous tests, we need to modify the input schema and generate values based on the modified definition. Modification can be done in two ways, it depends whether we want to fuzz parameter values or not. Typically, we want to fuzz all *optional fuzzable* parameters and we need to determine fuzzability for required parameters. Fuzzability of parameters depends on the `confidence` score of the requirement. This is further discussed in Section 5.5. An example of a modified schema is shown in Listing 5.3 and the corresponding generated stateful Case is shown in Listing 5.4.

The `id` parameter is *required non-fuzzable* and its value was selected from the output of the previous endpoint. The value of a *required fuzzable* parameter `fqdn` was randomly generated, and in the next test, the value can be selected from the `enum`. The *optional* parameter `name` was generated randomly. The generated `Case` is then executed and evaluated.

5.4 Execution and Evaluation

A generated and scheduled test is represented by a `Case` object. The object can be executed by two different runners—either by pytest or by the CLI runner. Both options will store the test result in a `TestResult` object, consisting of performed checks, error codes, and of a list of dependent endpoints executed prior the current test.

Apart from executing the test itself and storing the test result, the test executor is responsible for skipping tests and for running the configuration update phase. In order to minimize tests which are rejected by the service, we introduced skipping of tests resulting in too many 404s. Naturally, we skip only dependency tests, as we want to test the target endpoints thoroughly. Tests for an endpoint are skipped once they hit a threshold for successive 404 count. The threshold limit is set to five successive 404 responses for the same endpoint and is determined experimentally. Comparison of different threshold limits is shown in Section 6.1.1.

As mentioned earlier in the Design chapter, the test is evaluated as a failure when the API call returns 5xx HTTP status code or the request times out.

Test minimization and deduplication is primarily handled by the Hypothesis library, it deduplicates and mimimizes a test case by finding common parameters used in a test and minimizes the generated data types. To deduplicate potential failures found by stateful and non stateful tests, we prefer stateful result with a shortest list of previous tests.

```
{
    "properties": {
        "fqdn": {
            "oneOf": [
                {"type": "string"},
                {"enum": ["sys1.fuzz.com"]
                                       }
            ]
        },
        "id": {
            "type": "string",
            "enum": [
                "4d1b0da2-957a-11ea"
                "e0803fce-9564-11ea",
            ]
        },
        "name": {
            "type": "string"
        },
    }
    "required": ["id", "fqdn"],
    "type": "object"
}
```

Listing 5.3: Stateful input schema

```
Case(
    endpoint=Endpoint(
        path='/systems',
        method='PATCH',
        app=None,
        base_url='http://localhost:8080',
        path_parameters=None,
        headers=None,
        cookies=None,
        ...,
    ),
    path_parameters=None,
    headers=None,
    cookies=None,
    query=None,
    body={
        'fqdn': '\x06\U0004bdd9\x12',
        'id': '4d1b0da2-957a-11ea',
        'name': ',\U0003a5d2\x0f\x16'
    },
    form_data=None
)
```

Listing 5.4: Stateful Schemathesis Case

5.5 Configuration Updating

Configuration updating is updating the test model, definitions of input schemas and the state of the test run. Update of the test run's state in this phase modifies requirements inside the `State`.

Requirements update is needed to add new values for the required parameter and requirement's confidence. Confidence is rising exponentially if the request is rejected, thus the requirement seems to be *required non-fuzzable*. Otherwise, if the request returns 2xx

```
@attr.s
class Requirement:
    confidence: int =attr.ib(default=0)

    @property
    def is_fuzzable(self)
        return self.confidence <CONFIDENCE_THRESHOLD:

# inside the test runner
if requirement.is_fuzzable:
    if response.status_code in SUCCESS_STATUSES:
        requirement.confidence //=2
    if response.status_code ==404:
        requirement.confidence +=5
        requirement.confidence *=2
```

Listing 5.5: Confidence of a requirement

HTTP status code and the requirement is fuzzable, we will further decrease its confidence to retain the fuzzability of the requirement. Once the requirement becomes not fuzzable it stays *required non-fuzzable*. The confidence decision is presented in Listing 5.5. The values affecting confidence increase and decrease were determined experimentally.

Once we have values for requirements, we can update the model. The model update is an update of the input schema of an endpoint where the fuzzability of required parameters is decided based on their confidence level. The modification is in adding an `enum` property to the schema. This property ensures that the values will be selected from the enumeration. For fuzzable parameters, we add a choice to generate data based on the data type or to use a value from an enumeration by using JSON Schema `oneOf` keyword for combining multiple schemas. Listing 5.6 displays what the modification of the schema would look like.

```
# initial schema
"id": {
    "type": "string"
}
# schema with stateful values---not fuzzable
"id": {
    "type": "string",
    "enum": ["4d1b0da2-957a-11ea-bb37-0242ac130002"]
}
# schema with stateful values---fuzzable
"id": {
    "oneOf": [
        {"type": "string"},
        {
            "type": "string",
            "enum": ["4d1b0da2-957a-11ea-bb37-0242ac130002"]
        }
    ]
}
```

Listing 5.6: Schema modification

Chapter 6

Evaluation

We evaluated the implemented fuzzer against multiple services of Red Hat Insights[1] found at https://cloud.redhat.com, Vulnerability Management as a Service (VMaaS)[2], also made by Red Hat, and our testing application. In this chapter, we present the performed experiments and their results. The first set of experiments, presented in Section 6.1 evaluate the effect of proposed features and support reasoning behind our implementation decisions. We evaluate influence of the proposed features (such as skipping 404s or depenedency sorting) on different metrics such as the number of discovered bugs or the code coverage. Afterwards, we evaluate the fuzzer with settings obtained from the previous experiments on our testing application (Section 6.2. Finally, we perform fuzz testing of real-world REST APIs and present the bugs found in the tested applications in Section 6.3.

6.1 Evaluations of Proposed Features

We conducted several experiments to increase the quality of the implemented fuzzer. We tried multiple features and their effect on code coverage or on the number of tests. Then, the experiments were compared to Schemathesis results without stateful testing and a combination of stateful and basic testing. Experiments were performed locally using docker-compose and mocked dependent services of VMaaS and Vulnerability Insights API. We collected code coverage, found bugs, and count of executed tests. Conclusion of conducted experiments is discussed in Section 6.1.5.

6.1.1 Skipping 404s

First, we need to test what we can gain from skipping requests after reaching the threshold of successive 404 responses. We performed this experiment on Vulnerability application, only, since the VMaaS's API is that it does not return the 404 HTTP status code when the resource does not exist. Instead it returns a 200 response with an empty JSON. Supplementary to the experiment on Vulnerability application, we conducted the same experiment on our testing application to illustrate greater impact on code coverage. Table 6.1 below shows the total time of the test run, code coverage, and the number of performed tests with different threshold values for successive 404 responses for Vulnerability application

[1] Red Hat Insights documentation - https://access.redhat.com/documentation/en-us/red_hat_insights

[2] VMaaS - https://github.com/RedHatInsights/vmaas/

and Table 6.2 for our testing application with an extra measurement—number of found bugs.

Threshold	# of tests	Test time	Coverage
None	27161	131.67s	71%
20	1000	87.84s	71%
10	988	82.53s	71%
5	984	82.22s	71%
1	980	81.66s	70%

Table 6.1: Skipping 404s on Vulnerability Insights

From the experiment on Vulnerability Insights, we can see that skipping tests for dependencies which result in successive 404s makex sense in terms of decreasing a number of tests and test time while maintaining the same level of code coverage. The experiment was performed against an application running locally (using docker-compose) with database synced from pre-production Red Hat Insights environment. The test time against a deployed application would be increased by tens of milliseconds per test, making the a bigger gap in test time with different threshold values. Skipping endpoint tests after a single 404 response results in decrease in only 1% of code coverage. To demonstrate how different it can be on another application, we performed the experiment on our testing application. Results can be seen in Table 6.2.

Threshold	# of tests	Test time	Coverage	# of bugs
None	307	7.58s	95%	1
5	304	5.39s	95%	1
1	205	10.65s	86%	0

Table 6.2: Skipping 404s on testing application

Code coverage of testing application dropped by 9% when we skipped endpoint test after the first 404 response. More importantly, this test run did not trigger the stateful bug in the application. Initially, it used an incorrect example value (system id) from the schema specification, thus the test of a dependent endpoint resulted in 404 response, and it did not provide correct example value for the target endpoint. This is also responsible for the increased test time. Testing with `threshold=1` executed a lot of tests against target endpoints resulting in 404s. On the other hand, higher threshold resulted in getting correct required parameters and a quick failure of target endpoints. Hypothesis stops testing when it finds same failures in endpoint's test case. Based on this experiment, we decided to skip testing of dependent endpoints after reaching **5** successive 404 responses for a single endpoint, which gives us opportunity to find correct values of required non-fuzzable parameters without trying too many API calls.

6.1.2 Sorting Dependencies

The next feature that we tested is the order of tests for dependency endpoints. We compared a randomly shuffled set of dependencies with an ordered set. Dependencies are sorted by two keys, ascending by the number of required values and descending by dependency count. This approach is prioritizing dependencies having less required parameters and dependencies providing more required values for other endpoints. Comparison can be seen in Table 6.3 based on the number of tests, code coverage, and the number of found bugs. Experiment is conducted against the Vulnerability Insights application. We used stateful approach only, with 150 generated examples per endpoint.

	# of tests	**Coverage**	**# of bugs**
Sorted	4500	80%	1
Shuffled	4300	78%	2

Table 6.3: Sorting dependencies

Sorted dependencies explored slightly bigger portion of the application code. Finding one less bug compared to shuffled dependencies is not a big problem. We have to keep in mind that inputs are randomly generated, and most probably, it triggered one more bug because it tested endpoint with generated input instead of using the obtained value. The most important metric for this experiment is the code coverage that is higher for the sorted set of dependencies. All in all, sorting of dependencies seems to be a better option and we can still combine stateful testing with generating random inputs for required fuzzable parameters to trigger bugs caused by random input. This is explained further in Section 6.1.4.

6.1.3 Using Examples from the Specification

There are cases when we are unable to obtain required values from dependencies. Then, we have two options:

- Test the target endpoint with random inputs.
- Use an example values defined in schema specification.

Both options were tested and compared by the number of tests, code coverage, and the number of bugs on the Vulnerability and VMaaS services. Results are shown in Table 6.4 for Vulnerability and in Table 6.5 for VMaaS.

	# of tests	**Coverage**	**# of bugs**
Random	4500	80%	1
Examples	1400	70%	0

Table 6.4: Using example/random values for required parameters on Vulnerability as an fallback to stateful value

	# of tests	**Coverage**	**# of bugs**
Random	15000	67%	0
Examples	15000	72%	0

Table 6.5: Using example/random values for required parameters on VMaaS as an fallback to stateful value

As we can see, the results are contradictory. Obtaining examples from schema as a fallback when we are unable to find the required value, decreases the coverage of Vulnerability, but it increases the coverage of VMaaS. One of the reasons is that the specification of Vulnerability contains non-existing system identifiers. If the identifier of a system is not found in the dependencies, one specified in the schema is used, but it is always invalid (since the identifiers are genereted when a system is uploaded to the service). Thus, it leads to a 404 response. Required parameters of VMaaS endpoints consists mostly of CVE[3] id, erratum[4] id, or RPM package's NEVRA (Name Epoch Version Release Architecture)[5]. All of these identifiers, issued by Red Hat, are always present in the VMaaS database, thus the example values in the schema are valid. Our goal is to explore deeper states of the application, so generally, we would want to use every possible chance to obtain the required value, even the one from the specification. The bug in Vulnerability is found thanks to a randomly generated input. A better solution to find this bug is to use a combination of stateful and random testing.

6.1.4 Combination of Random and Stateful Testing

As we have mentioned in the previous experiments, a combination of stateful and random testing should increase code coverage and find more bugs. To back up our idea, we conducted an experiment combining a stateful and a random testing on the Vulnerability Insights and VMaaS applications. For stateful and random testing, we used 150 Hypothesis examples and for a combination of both approaches, we decreased the number of examples to 100, to create roughly the same number of tests. Tables 6.6 and 6.7 show the collected number of tests, code coverage, and the number of bugs for Vulnerability and VMaaS, respectively.

Before the experiment itself, let us illustrate the combination of both approaches on our testing application. In the example, we use two endpoints, `GET /systems/{id}` and its dependency `GET /systems`. Our focus is to test `GET /systems/{id}`. Using the stateful approach, we test the endpoint using the `id` value obtained from dependency combined with randomly generated query parameter. One of the tests could look like `GET /systems/9afd-12aa?query="`. On the other hand, random test can test the endpoint `GET /systems/{id}` with a randomly generated value, such as `GET /systems/%00`. Both options can find different bugs, but usually, stateful testing won't find the bug found by the random testing because it will use the value from dependency. Random testing won't find the stateful bug if the format of the required parameter is as complicated as in our case. Therefore, a combination of both techniques should provide better results.

	# of tests	Coverage	# of bugs
Random	2500	78%	6
Stateful	4500	80%	1
Combination	4400	82%	5

Table 6.6: Combination of random and stateful testing of Vulnerability

From the Vulnerability results, we see that the combination of both approaches explores the biggest portion of application source code. However, it finds one less bug than random testing. Although, we have to notice that inputs are randomly generated and different test runs may trigger different bugs. Combination of random and stateful testing of VMaaS does not provide any improvement. This is mainly caused by the specification of VMaaS API response. For example, some endpoints require a `nevra` parameter, but none of the

[3] Common Vulnerabilities and Exposures - https://www.redhat.com/en/topics/security/what-is-cve

[4] Red Hat Errata - https://access.redhat.com/articles/2130961

[5] RPM - https://rpm.org/

	# of tests	Coverage	# of bugs
Random	2400	65%	0
Stateful	15000	72%	0
Combination	17000	72%	0

Table 6.7: Combination of random and stateful testing of VMaaS

endpoints provide this value named as `nevra`. The values are rather returned as the NEVRA of the exact package inside the `package_list` value. Thus, a lot of required parameters are already randomly generated and we do not see any improvement by using combination of both techniques for VMaaS.

6.1.5 Conclusion of Experiments

From the experiments above, we can see that fuzzing of each application is specific. Our goal is to create a tool as general as possible, thus, we decided to combine stateful and random testing since it explores more statements of the tested application. Random testing is hitting bugs triggered by randomly generated inputs, making it a necessary supplement of stateful tests.

We can also see that sorted dependencies slightly increases code coverage, thus we used it in our fuzzer as well.

Using example values defined in the schema as a fallback when no required value is found can, on the other hand, even decrease code coverage. The reason is that if the example value is invalid, the fuzzer may result in a loop of 404 responses. An example of an invalid value in the API specification is the system id that does not exist in the application. However, in terms of avoiding the 404 status code, any value is better than the generated one. That can be seen by a higher increase in code coverage for VMaaS than a decrease in coverage for Vulnerability, making the example value fallback a better option in terms of universality of the fuzzer.

Skipping dependency testing after hitting a threshold of successive 404 responses is also useful to decrease the number of tests and the time of testing while exploring the same code.

Stateful fuzzing is very dependent on specification correctness and consistency. An incorrect schema will halt the entire testing, invalid example values can poison obtained required values, and inconsistent naming of parameters can spoil dependency inference.

6.2 Fuzzing the Testing Application

With respect to the experiments concluded above, we tested our implementation of stateful REST API fuzzer on the testing application that contains a fault seen only when tested statefully. The fault is a *typo* introduced in the `PATCH /systems` endpoint. To hit the fault, a user needs to use the correct value for the required non-fuzzable parameter `id`. The faulty code is shown in Listing 6.1. The comparison between testing of the applicattion with Schemathesis 1.2.0 and with our implementation can be found in Table 6.8.

As seen in Table 6.8, stateful testing hit the bug that testing with Schemathesis 1.2.0 did not. The reason is that our solution found valid identifiers of systems which were used in the following API calls. Specifically, it did not find the bug by just providing a correct required non-fuzzable parameter value, but by accessing a deeper state of the application.

```
def patch_system(body):
    """Modify system’s name."""
    system_id =body.get("id")
    name =body.get("name")
    fqdn =body.get("fqdn")
    if not SYSTEMS.get(system_id):
        return {"error": f"System ‘{system_id}‘ not found."}, 404
    # change system’s attribute fqdn to fqnd
    SYSTEMS[system_id] ={"name": name, "fqnd": fqdn}
    return {"updated": system_id}, 200
```

Listing 6.1: The fault in testing application

	# of tests	Test time	Coverage	# of bugs
Schemathesis 1.2.0	201	7.58s	86%	0
Stateful	304	5.39s	95%	1
Combination	413	12.47s	100%	1

Table 6.8: Fuzzing of testing application

An improved exploration of the application state space is also supported by increase of code coverage. The testing sequence causing a bug was the following:

1. `GET /systems` to obtain identifiers of systems. It is a dependency of the target `GET /systems{id}` endpoint, providing a valid value for `id` required non-fuzzable parameter.
2. `PATCH /systems` to modify a system resource. The resource is modified by the faulty method shown in Listing 6.1. The requests against this endpoint pass, as it correctly returns the 200 status code.
3. `GET /systems{id}` to access a recently modified resource, found by `GET /systems` and modified by `PATCH /systems`. Because `PATCH /systems` modifies in-memory database incorrectly, the API call to this endpoint results in *500 Internal Server Error*.

The issue is not possible to find without using stateful data and it is also not reproducible without knowledge of previously tested endpoints. Combination of random and stateful testing provides better results even if it takes longer which underlines the results presented in Section 6.1.4.

6.3 Fuzzing Real-world REST APIs

After experimental evaluation of implemented features and determining the optimal parameters of our fuzzer, we move to testing real-world REST APIs. First, we test Red Hat Insights services of `https://cloud.redhat.com` in pre-production OpenShift Dedicated[6] environment and present the bugs found in respective applications. Then, we compare our fuzzer to QuickREST [22], a proof-of-concept tool tested against Gitlab API.

[6] OpenShift Dedicated - `https://www.openshift.com/products/dedicated/`

6.3.1 Testing Red Hat Insights

In this section, we summarize discovered bugs in the pre-production environment of https://cloud.redhat.com. We deployed our fuzzer to test the same environment as Red Hat Insights to reduce the latency of API calls and decreasing the test time. A combination of stateful and random fuzz testing was used to test seven applications in total, namely, Advisor, Compliance, Drift, Inventory, Patch, Remediations, and Vulnerability. All these applications were tested using 100 Hypothesis examples per endpoint.

Before we move to the testing of the Red Hat Insights application, we need to deploy the fuzzer to the same environment as the tested application to minimize latency. Initially, we wanted to deploy the fuzzer alongside a tested application in the ephemeral environment in OpenShift. However, we struggled with the deployment of applications due to occasional problems with their deployment. Instead, we decided to test applications in the pre-production environment. To increase the speed of requests, we deployed a Schemathesis container to the same OpenShift Dedicated cluster.

Advisor The first tested application was Advisor. Advisor is a service evaluating systems against a set of rules. During the testing, our fuzzer executed 8000 tests, found 7 bugs and 3 other issues. Two endpoints, `GET /system/uuid/reports/` and `DELETE /api/insights/v1/ack/{rule_id}/`, are not sanitizing input values for required parameters. Resulting in *500 Internal Server Error* if `uuid=0` or `rule_id='%00'`. Other bugs were found after using incorrect values for query parameters, such as `rule_id=[0\x00']` or `display_name='\x00'` for 5 other endpoints.

One interesting bug was found thanks to our stateful extension in `GET /rule/{rule_id}` endpoint. If a user provides a valid `rule_id` parameter and an invalid value for `tags` query, it will result in 500 status code. Interesting part is that it happen only if `GET /ack/{rule_id}` return 404. The lack of *acknowledgement* for a *rule* was caused by previously deleting the *acknowledgement* by `DELETE /ack/{rule_id}` endpoint.

Other issues found in Advisor seem to be intermittent and not reproducible. Even the fuzzer marked these examples as flaky. They were, according to Hypothesis, unreliable. The exact output is „`Falsified on the first call but did not on a subsequent one"`.

Compliance Next, we tested the Compliance service. It is assessing the system's compliance to OpenSCAP[7] security policy. Our fuzzer executed 300 tests and found 9 bugs. Found bugs are very similar to these in Advisor service. Two are caused by the lack of path parameters sanitization, triggered by `id='%00'`. Others are caused by missing sanitization of query parameters, specifically of the `limit` query parameters used for limiting the number of returned data. The parameter expects an *integer* value and results in 500 status code when a *boolean* value is used. It affects 7 endpoints.

We have also seen some flakiness. Some endpoints exceeded 500ms request timeout, but they responded quicker on the subsequent request. We have noticed one strange situation when `id="` was considered a valid requirement. It is caused by Compliance having endpoints `GET /profiles/{id}` and `GET /profiles/`. Using empty string for `id` parameter was considered as making a request to `GET /profiles/`. Thus, it returned 200 status code, some data in JSON as well, and we marked the requirement as a valid example.

[7] OpenSCAP - https://www.open-scap.org/

Drift Service comparing multiple systems is called Drift. Only 200 tests were generated and we hit one bug. If the request body of `POST /comparison_report` is empty, it returns 500 HTTP status code.

Inventory Another tested service was Inventory. This is the service which all other services depend on. It stores all uploaded systems to Red Hat Insights. Despite running more than 3600 tests, we found just 2 bugs. One is due to the pagination of responses—when a user requests a non-existing page by providing a big integer value, it results in an error. Another error is found for a particular request body shown in Listing 6.2 generated by an input schema of the `POST /hosts` endpoint.

```
[
    {
        ’account’: ’’,
        ’reporter’: ’’,
        ’stale\_timestamp’: ’2000-01-01T00:00:00Z’,
        ’’: -9223372036854775809
    }
]
```

Listing 6.2: Inventory service bug - request body

It is necessary to mention that all required non-fuzzable parameters used generated values. The cause is lying within the service specification of required `host_id_list` parameter of the type *string*. In the specification, it is described as *a comma separated list of host IDs*. Other endpoints return identifiers of hosts as `id`, but the fuzzer cannot know that the identifiers should be concatenated with comma delimiter to create a valid value for `host_id_list` parameter.

Patch The service used for patching systems by applying needed Red Hat Product Advisories is called Patch. We found the same kind of issues as in the previous services caused by sanitizing of path parameters. Using `’%00’` for `inventory_id` or `advisory_id` parameters, triggers *Internal Server Error*. It affects 5 endpoints during the test run of 1000 test cases.

Remediations Patching is handled by the Remediations service by creating Ansible playbook[8] for the affected systems. We found two bugs in 3100 tests. Both are caused by providing a *floating point* number instead of an *integer* for `offset` query parameter of the `GET /remediations` and `GET /remediations/{id}/playbook_runs` endpoints.

Vulnerability Finally, we have tested the Vulnerability service that is reporting CVEs and provides mitigation plans for affected systems. As seen in previous sections, Vulnerability was tested by 4500 tests and it resulted in 6 errors. All of the bugs are caused by an insufficient sanitizing of path parameters. *Internal Server Error* is triggered by providing `’%00’` as a value for `cve_id` or `inventory_id` parameters.

However, we see a similar problem as in Inventory. One parameter, `cve_id`, is not returned by any endpoint and `inventory_id` is returned only by other endpoint having `cve_id` as a required non-fuzzable path parameter. Luckily, `cve_id` example in schema was a CVE affecting some systems, hence we obtained a required non-fuzzable `inventory_id` parameter value.

[8] Ansible playbook - https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html

6.3.2 Testing Gitlab API

Authors of QuickREST [22] were able to find some issues in Gitlab API. We have followed their steps and tried to reproduce the same issues. It is necessary to note that QuickREST is a proof-of-concept tool and not a production-ready tool by any means.

First, we set up the vulnerable Gitlab version based on their reproducer steps found in [21]. Then, we executed their experiments and after a few test runs, we hit the bug mentioned in their scripts.

Sadly, Gitlab is not providing the API specification in the OpenAPI format. Instead, the documentation can be found in a human-readable format in their API documentation[9]. In order to run experiments, authors of QuickREST created a subset of Gitlab API defined in OpenAPI 2.0 (Swagger) format manually.

To reproduce the Gitlab stateful bug, we used the same OpenAPI specification as the authors of QuickREST. However, we didn't find any issues. The reason is the way reproduce the bug. The bug is a deadlock in Gitlab and it is reproducible by alternating two requests very quickly. Our tool is unable to make such a test run, since we are making requests endpoint-wise, thus we are first executing all tests for the dependent endpoints and then tests for the target endpoint. QuickREST is able to trigger this bug not only by the ability to create test cases with alternated endpoint tests but mainly thanks to the definition of Gitlab API input schema for this experiment. When we tried to run the same QuickREST experiment script against a schema extended with another endpoint, the script did not find the bug.

6.3.3 Conclusion

All in all, our fuzzer found 32 bugs in 7 services. All bugs are caused by insufficient input validation, triggered by unicode, special strings, or different data types. One bug was found exclusively by stateful testing. The fuzzer needed to use a valid path parameter and a generated query parameter for a resource returning 404 by different API call, the reproducer can be seen in Listing 6.3. Luckily for Red Hat Insights, we did not find any stateful bug caused by improperly modified resource or a race condition.

```
______________________ GET: /api/insights/v1/rule/{rule_id}/systems/ ______________________
1. Received a response with 5xx status code: 500
Previous endpoints:
        1. GET: /api/insights/v1/rule/
        2. GET: /api/insights/v1/ack/{rule_id}/
        3. DELETE: /api/insights/v1/ack/{rule_id}/

Check :not_a_server_error
Path parameters :{’rule_id’: ’amd_sme_enabled|AMD_SME_ENABLED’}
Query :{’tags’: [’’]}

Run this Python code to reproduce this failure:
    requests.get(
        f"{base_url}/api/insights/v1/rule/amd_sme_enabled|AMD_SME_ENABLED/systems/",
        params={’tags’: [’’]}
    )
```

Listing 6.3: Stateful bug output

[9] Gitlab API Docs - https://docs.gitlab.com/ee/api/README.html

By testing Red Hat Insights services, we noticed how crucial the schema specification is for successful fuzzing. In the first place, a specification must be in a valid OpenAPI or Swagger format. For stateful testing, a schema has to be consistent, so that the required parameters can be found in responses. When the required parameter cannot be obtained from dependencies, it is nice to have OpenAPI features implemented by an application, like enumeration of accepted values[10] or regular expression pattern[11] for the ability to generate an example accepted by the service.

[10] OpenAPI Enums - https://swagger.io/docs/specification/data-models/enums/

[11] OpenAPI Pattern - https://swagger.io/docs/specification/data-models/data-types/#pattern

Chapter 7

Conclusion

In this thesis, we presented an approach to stateful fuzzing by inferring dependencies of REST API endpoints. The result is a pull request extending the Schemathesis project by adding capabilities for stateful testing. Schemathesis is internally used by the Insights QE team for fuzz testing of Red Hat Insights services. Once the pull request is merged, Schemathesis will become, to our best knowledge, the first production-ready tool for stateful fuzzing of REST API. Our designed fuzzer is minimizing 404 HTTP status responses since we are using inferred inputs for required parameters. Using the values from previous tests also allows us to create different testing scenarios and to explore deeper states of the application. Moreover, skipping dependencies returning 404 status codes helps to further decrease the number of rejected requests and minimize the number of executed tests. Dependency skipping with ordering of dependencies maximizes the probability of obtaining a correct input value for a required parameter. Furthermore, modification of an input schema determining a fuzzability of required parameters contributes to our solution of exploration vs. exploitation problem.

During testing of the fuzzer on the Red Hat Insights application, we found 32 bugs. In particular, we found one stateful bug that needs to execute API calls against three dependent endpoints before triggering the issue in the target endpoint. To reproduce this bug, it is necessary to obtain a set of valid parameter values. Then, proceed with finding the value from the set that has a certain property (rule acknowledgement). Subsequently, the fuzzer deletes the property of the resource, and finally, it tests the target endpoint with a valid parameter value to access a previously inferred resource and a randomly generated query parameter.

One of the possible improvements of our project can be in the order of test execution. Right now, we are executing tests endpoint-wise, but, as testing on the Gitlab API showed us, it would be useful to alternate endpoints and do not execute the whole test set for an endpoint at once. A possibility of mutation of obtained values for *optional fuzzed* or *required fuzzed* parameters may create inputs accepted by the service while being problematic to handle. Finally, adding a white-box capability to guide test creation by so far untested code paths can be great for the tool's universality.

Bibliography

[1] Arcuri, A. RESTful API Automated Test Case Generation. In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. July 2017, p. 9–20. ISBN 978-1-5386-0592-9.

[2] Atlidakis, V., Godefroid, P. and Polishchuk, M. RESTler: Stateful REST API Fuzzing. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May 2019, p. 748–758. ISSN 0270-5257.

[3] Berners Lee, T., Fielding, R. T. and Masinter, L. M. *Uniform Resource Identifier (URI): Generic Syntax* [RFC 3986]. RFC Editor, january 2005. Available at: https://rfc-editor.org/rfc/rfc3986.txt.

[4] Bozkurt, M., Harman, M. and Hassoun, Y. Testing & Verification In Service-Oriented Architecture: A Survey. *Software Testing, Verification and Reliability*. june 2013, vol. 23.

[5] Böhme, M., Pham, V. and Roychoudhury, A. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*. May 2019, vol. 45, no. 5, p. 489–506. ISSN 2326-3881.

[6] Böttinger, K., Godefroid, P. and Singh, R. Deep Reinforcement Fuzzing. In: *2018 IEEE Security and Privacy Workshops (SPW)*. May 2018, p. 116–122.

[7] Carion, U. *JSON Schema Language*. Internet-Draft draft-json-schema-language-02. Internet Engineering Task Force, july 2019. [cit. 2019-12-06]. Work in Progress. Available at: https://datatracker.ietf.org/doc/html/draft-json-schema-language-02.

[8] Chakrabarti, S. K. and Kumar, P. Test-the-REST: An Approach to Testing RESTful Web-Services. In: *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*. IEEE, Nov 2009, p. 302–308. Available at: http://ieeexplore.ieee.org/document/5359602/. ISBN 978-1-4244-5166-1.

[9] Chan, W. K. and Jiang, B. Fuse: An Architecture for Smart Contract Fuzz Testing Service. In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. Dec 2018, p. 707–708. ISSN 1530-1362.

[10] Chen, P. and Chen, H. Angora: Efficient Fuzzing by Principled Search. In: *2018 IEEE Symposium on Security and Privacy (SP)*. May 2018, p. 711–725. ISSN 2375-1207.

[11] Earle, C. B., Fredlund, L. Åke, Herranz Ángel and Mariño, J. Jsongen. In: *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang - Erlang '14*. New York, New York, USA: ACM Press, 2014, p. 33–41. Available at: http://dl.acm.org/citation.cfm?doid=2633448.2633454. ISBN 9781450330381.

[12] Ed douibi, H., Canovas Izquierdo, J. and Cabot, J. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. october 2018, p. 181–190.

[13] Fertig, T. and Braun, P. Model-driven Testing of RESTful APIs. In: *Proceedings of the 24th International Conference on World Wide Web - WWW '15 Companion*. New York, New York, USA: ACM Press, 2015, p. 1497–1502. Available at: http://dl.acm.org/citation.cfm?doid=2740908.2743045. ISBN 9781450334730.

[14] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation*. University of California, Irvine, 2000. Available at: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[15] Fielding, R. T. and Reschke, J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [RFC 7231]. RFC Editor, june 2014. Available at: https://rfc-editor.org/rfc/rfc7231.txt.

[16] Fowler, D. S., Bryans, J., Shaikh, S. A. and Wooderson, P. Fuzz Testing for Automotive Cyber-Security. In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. June 2018, p. 239–246. ISSN 2325-6664.

[17] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K. et al. CollAFL: Path Sensitive Fuzzing. In: *2018 IEEE Symposium on Security and Privacy (SP)*. May 2018, p. 679–696. ISSN 2375-1207.

[18] Godefroid, P., Levin, M. Y. and Molnar, D. Automated Whitebox Fuzz Testing. In:. November 2008. Available at: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

[19] Godefroid, P., Levin, M. Y. and Molnar, D. SAGE: Whitebox Fuzzing for Security Testing. *Queue*. New York, NY, USA: ACM. january 2012, vol. 10, no. 1, p. 20:20–20:27. Available at: http://doi.acm.org/10.1145/2090147.2094081. ISSN 1542-7730.

[20] Guha, S., Munagala, K. and Shi, P. Approximation Algorithms for Restless Bandit Problems. *CoRR*. 2007, abs/0711.3861. Available at: http://arxiv.org/abs/0711.3861.

[21] Karlsson, S., Causevic, A. and Sundmark, D. *QuickREST replication packages* [online]. GitHub [cit. 2020-05-27]. Available at: https://github.com/zclj/replication-packages/tree/master/ICST-2020.

[22] Karlsson, S., Causevic, A. and Sundmark, D. *QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs*. 2019.

[23] Kuznetsov, A., Yeromin, Y., Shapoval, O., Chernov, K., Popova, M. et al. Automated Software Vulnerability Testing Using Deep Learning Methods. In: *2019 IEEE 2nd Ukraine Conference on Electrical and Computer Engineering (UKRCON).* July 2019, p. 837–841.

[24] Liang, H., Pei, X., Jia, X., Shen, W. and Zhang, J. Fuzzing: State of the Art. *IEEE Transactions on Reliability.* Sep. 2018, vol. 67, no. 3, p. 1199–1218. ISSN 1558-1721.

[25] Manico, J. and Saad, E. *REST Security* [online]. OWASP Cheat Sheet Series, 2019 [cit. 2019-12-06]. Available at: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html.

[26] Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M. et al. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering.* 2019, p. 1–1. Available at: https://ieeexplore.ieee.org/document/8863940/. ISSN 2326-3881.

[27] Masse, M. *REST API design rulebook.* Sebastopol, CA: O'Reilly, 2012. ISBN 978-1-449-31050-9.

[28] Miller, B. P., Fredriksen, L. and So, B. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM.* New York, NY, USA: ACM. december 1990, vol. 33, no. 12, p. 32–44. Available at: http://doi.acm.org/10.1145/96267.96279. ISSN 0001-0782.

[29] Peng, H., Shoshitaishvili, Y. and Payer, M. T-Fuzz: Fuzzing by Program Transformation. In: *2018 IEEE Symposium on Security and Privacy (SP).* May 2018, p. 697–710. ISSN 2375-1207.

[30] Sargsyan, S., Kurmangaleev, S., Mehrabyan, M., Mishechkin, M., Ghukasyan, T. et al. Grammar-Based Fuzzing. In: *2018 Ivannikov Memorial Workshop (IVMEM).* May 2018, p. 32–35.

[31] Sutton, M., Greene, A. and Amini, P. *Fuzzing: brute force vulnerabilty discovery.* Upper Saddle River, NJ: Addison-Wesley, 2007. ISBN 9780321446114.

[32] Takanen, A., DeMott, J., Miller, C. and Kettunen, A. *Fuzzing for software security testing and quality assurance.* Second editionth ed. Norwood, MA: Artech House, 2018. ISBN 9781608078509.

[33] Wright, A., Andrews, H., Hutton, B. and Dennis, G. *JSON Schema: A Media Type for Describing JSON Documents.* Internet-Draft draft-handrews-json-schema-02. Internet Engineering Task Force, september 2019. [cit. 2019-12-06]. Work in Progress. Available at: https://datatracker.ietf.org/doc/html/draft-handrews-json-schema-02.

[34] Zeller, A., Gopinath, R., Böhme, M., Fraser, G. and Holler, C. Fuzzing: Breaking Things with Random Inputs. In: *The Fuzzing Book* [online]. Saarland University, 2019 [cit. 2019-06-21]. Available at: https://www.fuzzingbook.org/html/Fuzzer.html.

[35] Zhao, J. and Pang, L. Automated Fuzz Generators for High-Coverage Tests Based on Program Branch Predications. In: *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*. June 2018, p. 514–520.

Appendix A

Attached Files

Directory tree of the attached media is the following:

- **schemathesis** - Git repository of Schemathesis project. Our enhancement is found in `stateful` branch.
- **stateful_app** - Source code of testing application used in the thesis.
- **text** - Thesis in `pdf` format and LaTeXsource files.

Appendix B

Usage

The Schemathesis can be executed by either CLI or pytest runner and the application can be installed using `Poetry`[1] as follows:

```
cd schemathesis
git checkout stateful
python3 -m pip install poetry
poetry install
# switch to a virtual environment with installed schemathesis
poetry shell
```

Listing B.1: Schemathesis installation

Command Line Interface Schemathesis tests cases are executed using the `schemathesis` command:

```
schemathesis run https://example.com/api/swagger.json
```

Listing B.2: Running schemathesis

If an application requires authorization then one can use `-auth` option for *Basic Auth* and `-header` to specify custom headers to be sent with each request.

CLI supports passing options to `hypothesis.settings`, prefixed with `-hypothesis-`. For example set the number of generated examples:

```
schemathesis run --hypothesis-max-examples=1000 https://example.com/api/swagger.json
```

Listing B.3: Schemathesis - max hypothesis examples

To minimize the number of 404 errors and ability to catch issues caused by other endpoints accessing/modifying the same resource, one can run stateful tests by passing CLI option `-stateful`.

```
schemathesis run --stateful https://example.com/api/swagger.json
```

Listing B.4: Schemathesis - CLI stateful tests

[1] Poetry - https://python-poetry.org/

Pytest If you would like to run stateful tests, you need to provide `stateful=True` parameter either in schema preparation step

```
schema =schemathesis.from_uri("http://0.0.0.0:8080/swagger.json", stateful=True)
```

Listing B.5: Schemathesis - stateful schema definition

or parametrize tests with

```
@schema.parametrize(stateful=True)
def test_no_server_errors(case):
    ...
```

Listing B.6: Schemathesis - statef tests

For stateful test you also need to update the current state based on the actual test result. An example of a stateful test could look as follows:

```
import schemathesis

schema =schemathesis.from_uri("http://0.0.0.0:8080/swagger.json")

@schema.parametrize(stateful=True)
def test_no_server_errors(case):
    response =case.call()
    # Update state - gather examples for required properties of endpoints
    schemathesis.update_state(case, response)
    # You could use built-in checks
    case.validate_response(response)
    # Or assert the response manually
    assert response.status_code <500
```

Listing B.7: Schemathesis - stateful test test_api.py

Finally, you can run the tests using:

```
pytest test_api.py
```

Listing B.8: Schemathesis - test execution using pytest

A more comprehensive documentation can be found at https://github.com/kiwicom/schemathesis (except for stateful options).

Appendix C

Running Testing Application

Testing application containing example of stateful bug is written in Python depending on Connexion framework. To install it, you can make use of Python's virtualenv as follows:

```
cd stateful_app
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

Listing C.1: Stateful bug output

Run the application using python -m main. Application will be running on http://localhost:8080/. The OpenAPI specification in JSON format can be found at http://localhost:8080/openapi.json and the generated Swagger UI at http://localhost:8080/ui.