Model-based fuzzing REST web services to detect vulnerabilities

Gerritsen, A (Arjan)

Award date:
2020

Link to publication

Model-based fuzzing REST web services to detect vulnerabilities

Author : A. Gerritsen Student number : 000000000000000000000

Date of presentation : November 27, 2020

Open Universiteit
www.ou.nl

Title	Model-based fuzzing REST web services to detect vulnerabilities
Author	Arjan Gerritsen
Student number	851368230
Institute	Open University of the Netherlands, Faculty of Science
Degree	Master’s Programme in Software Engineering
Chair	dr. ir. H. P. E. Vranken
Primary supervisor	dr. ir. H. P. E. Vranken
Secondary supervisor	dr. G. Alpár
Course code	IM9906

Acknowledgements

This thesis could not have been realised without the help and support of several people. Therefore, I would like to share some words of thanks in this section.

Firstly, I wish to express my sincere appreciation to my supervisors, Harald and Greg. Thank you both for giving me the chance to do this research under your directions and I am grateful for the valuable time you invested during this period. Harald, thank you for having our three weekly consultations. All the feedback you gave me was incredible valuable and enhanced the quality of this research and thesis. It has been a real pleasure working with both of you.

Secondly, I would like to thank my employer, the Belastingdienst and my former employer Oblivion. Thank you for believing in me and investing the needed resources to make this study possible. Furthermore, I would like to thank my colleagues Pascal and Asif for debating software security topics and discussing my research with you.

Thirdly, I would like to pay special regards to my loved ones. My parents, you provided me with a solid base, a safe place to grow up, and always emphasised the importance of education. That has shaped me into who I am today, I will always be grateful for that. My parents in law, you always were interested in my research, asked how things were going, and always remembered when I had an exam. Thank you for caring and sympathising so much. Finally, I would like to express my deepest gratitude to my wife, Karin. Your love for learning inspired me to start this journey and you also helped me immensely in arriving to where I am now. When the journey was tough, you motivated me to continue. Furthermore, you always supported me when I needed (academic) advise. I could not have done it without you.

Arjan Gerritsen
Apeldoorn, November 2020

Table of contents

Summary

An effective automated technique to detect vulnerabilities is fuzzing, i.e. a software program sends generated random or unanticipated data to a system under test (SUT). Currently, new hybrid forms of fuzzing, combining different techniques, are used to increase their effectiveness on complex software systems. Prior research showed that REST web services are a good candidate for applying these hybrid forms. This research examines a new hybrid form: model-based behavioural dictionary fuzzing, a guided fuzzer that uses a list of strings that contain values that are likely to detect vulnerabilities. This leads to the main research question: How can model-based behavioural dictionary fuzzing be applied effectively on REST web services to detect vulnerabilities?

Firstly, a systematic literature review was conducted. The results showed that the four most important types of vulnerabilities in web services are: injection, broken authentication, broken access control, and cross-site scripting. Secondly, RESTFuzzer was developed by executing experimental prototyping. This fuzzer was tested on WordPress and SutSqlI, a self-developed SUT that intentionally contains SQL injection vulnerabilities. RESTFuzzer proved to be capable of executing model-based behavioural dictionary fuzzing, which means sending a high percentage of valid requests, detecting vulnerabilities, and executing fast. Also, RESTFuzzer is effective, which means achieving sufficient amount of code coverage.

Overall, this research contributes to the security awareness and emphasises the importance to put REST web service vulnerabilities more prominent on the software security research agenda. Furthermore, it shows that fuzzers can be used as a capable and effective security testing tool to detect vulnerabilities in REST web services. The open source applications RESTFuzzer and SutSqll can be applied in practice and in further research. Moreover, this research demonstrates the effectiveness of using a new hybrid form of fuzzing, a combination of model-based and dictionary fuzzing on REST web services.

Samenvatting

Een effectieve geautomatiseerde techniek om kwetsbaarheden te detecteren is fuzzing, dat wil zeggen dat een softwareprogramma willekeurige of onverwachte data naar een system under test (SUT) stuurt. Tegenwoordig worden nieuwe hybride vormen van fuzzing bestaande uit een combinatie van verschillende technieken gebruikt om de effectiviteit te verhogen wanneer deze worden toegepast op complexe software systemen. Eerder onderzoek heeft aangetoond dat deze hybride vormen van fuzzing geschikt zijn om uit te voeren op REST webservices. Dit onderzoek richt zich op een nieuwe hybride vorm van model-based en dictionary fuzzing, een gestuurde fuzzer die een lijst van strings gebruikt met waarden die waarschijnlijk kwetsbaarheden kunnen detecteren. Dit leidt tot de vraag: Hoe kan model-based behavioural dictonary fuzzing effectief toegepast worden op REST webservices om kwetsbaarheden te detecteren?

In het eerste deel van dit onderzoek is een systematische literatuurstudie uitgevoerd. De resultaten laten zien dat de vier belangrijkste typen kwetsbaarheden in webservices zijn: injection, broken authentication, broken access control en cross-site scripting. In het tweede deel van dit onderzoek is door middel van experimental prototyping RESTfuzzer ontwikkeld. Deze fuzzer is getest op WordPress en SutSqlI, een zelfontwikkelde SUT met de naam SutSqlI, die opzettelijk toegevoegde SQL injection kwetsbaarheden bevat. RESTFuzzer bleek geschikt te zijn voor het uitvoeren van model-based behavioural dictionary fuzzing, dat betekent het sturen van een hoog percentage van valide requests, detecteren van kwetsbaarheden en snel kunnen uitvoeren. Daarnaast is RESTFuzzer effectief, er wordt een voldoende mate van code coverage behaald.

Dit onderzoek draagt bij aan security bewustzijn en benadrukt daarbij het belang om REST kwetsbaarheden prominenter op de software security onderzoeksagenda te zetten. Verder laat dit onderzoek zien dat fuzzers gebruikt kunnen worden als een geschikte en effectieve security testing tool om kwetsbaarheden in REST webservices te detecteren. De open source applicaties RESTFuzzer en SutSqll kunnen toegepast worden in de praktijk en in vervolgonderzoek. Bovendien heeft dit onderzoek laten zien dat een nieuwe hybride vorm van fuzzing, een combinatie van model-based en dictionary fuzzing, effectief toegepast kan worden op REST webservices.

1 Introduction

Nowadays, software security is of crucial importance, because computers are present in every aspect of personal life. Computers are no longer only personal computers or bulky laptops on a desk in a workplace or at home. In contrast, computers are everywhere. For example, they are on the wrist as smartwatches, in the bathroom as smart scales, in the pocket as smartphones, and in the living room as intelligent speakers. The number of vulnerabilities in software increases for various reasons.

Firstly, the number of computers has increased enormously. Due to the decrease in size and power consumption, computers can be deployed for many goals at many locations. Furthermore, computers become more affordable to produce. This results in for example, the Internet of Things (IOT), which is a large network of small and low power devices, that are able to exchange data through the Internet.

Secondly, devices are connected to the Internet, all the time. Most devices require some kind of data transfer between the device and a server where data is collected and processed. Furthermore, many devices need to be reachable from the Internet to offer some kind of functionality. For example, devices for home automation need to be controlled from a website or an application installed on a smartphone. Therefore, all devices are really easy to access from the Internet, also by malicious actors who intend to steal data or disrupt availability of services.

Thirdly, the extensive amount of data that is collected by devices might be of value for malicious actors. A large amount of the gathered data is personal and privacy sensitive. Data can also contain e-mail addresses, usernames, passwords, or credit card information. This information can be sold on the Internet. Therefore, malicious actors will put in much effort to steal data.

Fourthly, the complexity and extensibility of software has increased. Software solutions are growing in size and therefore become more complex. Due to this complexity it is harder to oversee all the consequences of choices made by software developers when writing source code, also with regard to security. Also extensibility has increased, systems are more and more connected to each other, to exchange data between client and server or with other systems. An activity that often occurs over the Internet.

1.1 Consequences of vulnerabilities in software

Exploitation by a malicious actor of vulnerabilities in software can cause serious damage. Various threats to confidentiality exists, which means that (sensitive) information is accessible by unauthorized people. For example, individuals can be affected by losing control to whom they want to share their privacy-sensitive information with. Moreover, companies can be damaged directly by information leakage, like e-mails that include valuable information for competitors. In addition, security and/or safety of a country can be threatened when government agencies lose important information [1]. Also, integrity and availability can be threatened. The integrity is not guaranteed when data can be modified by a malicious actor and availability is at risk when an application or data is unavailable.

1.2 REST web services susceptible to exploitation of vulnerabilities

Representational State Transfer (REST) web services is a type of software that is susceptible to exploitation of vulnerabilities [2, 3]. REST is a software architectural style that comprises a set of constraints for the development of stateless web services to use in system-to-system communication in a network environment [4]. More on REST can be read in section 2.1. Especially Structured Query Language (SQL) injection vulnerabilities are found in web services [5].

In addition, the use of REST web services is growing significantly. This becomes visible in the popularity of service-oriented architectures (SOA), in which system-to-system communication is often realised via REST web services, in web development with the use of microservices [6], and Asynchronous JavaScript And XML (AJAX) web applications, in which communication between frontend and backend is achieved with REST web services. Due to the complexity, connectivity, and extensibility of these web services, the number of vulnerabilities is growing [7]. To minimise the number of vulnerabilities in production software, precautionary measures are needed.

1.3 Precautionary measures

Precautionary measures are needed to ascertain that software meets certain security quality standards. For example, the OWASP Application Security Verification Standard (ASVS) [8]. This standard provides a foundation for security testing of web applications and also can be used by development teams as a guideline for developing more secure software. Software development teams incorporate these measures to increase software security. In addition, researchers and developers will learn from malicious actors. Techniques used by malicious actors can also be converted into precautionary measures to increase quality of software solutions and decrease the amount of vulnerabilities present in production software.

Different types of precautionary measures are available, e.g. abuse cases can be drafted, static software analyses can be used, and security tests can be executed [7]. As stated by McGraw [7], multiple precautions at different stages of the software development life cycle (SDLC), so called touchpoints, should be applied. Executing security tests can only be applied to functioning software. Furthermore, security testing is a frequently applied and important precautionary measure.

For security testing, model-based security testing (MBST) can be applied as an automated precautionary measure. This technique can be used to reduce the number of vulnerabilities that end up in production software, because MBST helps to automate security related tests. At first, model-based testing (MBT) was used to automate the generation of functional test cases based on a model. In this context a model is a simpler representation of the system under test (SUT). In this way the model will be easy to maintain and test cases can be generated and executed automatically every time a change in the software is made. For more information on MBST, see section 2.3.

Another form of security testing is attack and penetration testing. This is a valuable tool for discovering security flaws in web services. It is of special interest to execute these attack and penetration tests automatically, because that enables executing tests more often during the development process, in contrast to manual tests that are labour intensive and therefore expensive. Integrating attack and penetration tests in the SDLC will lead to discovering vulnerabilities early in the development process and results in cheaper and higher quality web services [7]. Moreover, automating attack and penetration tests standardises the testing process, which does benefit the quality of the test suite, in contrast to manual unstructured tests [9]. In addition, this also applies to precautionary measures regarding security testing in general.

An effective automated tool used for attack and penetration testing is fuzzing [10]. A fuzzer is a software program that can send random or unanticipated generated data to a SUT. The first fuzzer, originated from 1990, was used to test parts of the operating system UNIX [11]. The complexity of software has increased significantly since then. More on fuzzing can be read in section 2.4. With the increasing complexity of REST web services, the number of possible paths to go through the software (search space) increases. The coverage of a fuzzer using generation of random input will be low on applications with large search spaces. Consequently, the original type of fuzzing is not very effective when applied to complex software solutions. Therefore, new (hybrid) forms of fuzzing are being researched by combining fuzzing with different existing techniques to increase their effectiveness, e.g. dynamic symbolic execution, coverage guide, grammar representation, scheduling algorithms, dynamic taint analysis, static analysis, and machine learning [10].

One of these hybrid forms of fuzzing is model-based (behavioural) fuzzing, which is part of the research field MBST [9]. This approach uses a model that guides the fuzzer to go effectively through the search space and deals with software complexity [12]. Effectively, in this context, is the goal to cover as much of the search space in as few tests as possible. Furthermore, the largest possible part of the program should be executed while fuzzing the SUT, with as few tests as possible. This can be achieved by sending input that is not rejected by input validation. In order to achieve that, the fuzzer is able to generate data that is conform the input specification. Model-based or behavioural fuzzing can also be combined with data fuzzing [13], for example with dictionary fuzzing. Dictionaries are collections of strings. When sent to a SUT, these strings are likely to detect or exploit a vulnerability. Examples from a dictionary containing strings to discover SQL injection vulnerabilities in MySQL databases are: `1'1`, `1 exec sp_ (or exec xp_)`, and `1 and 1=1` [14]. A successful exploit of such a SQL injection vulnerability can result in access to data or the ability to modify or delete data by an attacker.

Currently, not much research has been done on model-based (behavioural) dictionary fuzzing on REST web services. Atlidakis et al. [15] described in their research how a model can be derived from an OpenAPI Specification (OAS) and how this model can be used to fuzz REST web services on a locally installed web application (GitLab [16]). Atlidakis et al. recommended more research to find more systematic answers: what kinds of and how many bugs can be found with model-based fuzzing web services, and how severe will these security issues be?

1.4 Research questions

To conclude, the never-ending race between software developers and malicious actors continues. Software developers will try to create better quality software, so software contains fewer vulnerabilities. On the other hand, malicious actors will use new techniques, e.g. fuzzing, to exploit vulnerabilities present in software. In response, researchers will use those same techniques to develop tools for developers that helps creating more secure software.

This thesis describes a research that builds on the study of Atlidakis et al. [15] by applying model-based fuzzing techniques on REST web services. The main research question (RQ) is: **How can model-based behavioural dictionary fuzzing be applied effectively on REST web services to detect vulnerabilities?** To answer this research question, the following sub-questions are formulated:

- **RQ1: What types of vulnerabilities can be detected in REST web services?**
  A systematic literature review will be conducted to answer this question.

- **RQ2: How can a model-based behavioural fuzzer be developed that is capable of detecting vulnerabilities in REST web services?**
  A prototype will be developed capable of model-based behavioural fuzzing. Several properties of that prototype will be tested in an experiment on a selected SUT to determine its capability.

- **RQ3: How can a model-based behavioural dictionary fuzzer be developed that is capable of detecting SQL injection vulnerabilities in REST web services?**
  The prototype will be extended to also support model-based behavioural dictionary fuzzing. Several properties of that prototype will be tested in an experiment on a selected and self-developed SUT to determine its capability.

- **RQ4: How effective is a model-based behavioural dictionary fuzzer in detecting SQL injection vulnerabilities in REST web services?**
  Code coverage will be measured in an experiment when the prototype executes model-based behavioural dictionary fuzzing on a selected SUT.

1.5 Chapter overview

This thesis consists of six chapters. Chapter 2 presents the technical background of this research. The main technical concepts of this thesis, REST web services, OpenAPI specification, model-based security testing, and fuzzing, will be elaborated. Two studies were executed to answer the research question and its sub-questions. Chapter 3 presents the first study of this thesis which focuses on the vulnerability types found in REST web services (RQ1). The applied method, systematic literature review, is described. Furthermore, the results are presented to provide an overview of the types of vulnerabilities found in REST web services. Chapter 4 presents the second study of this thesis which focuses on the development of a model-based (behavioural) dictionary fuzzer (RQ2, RQ3, and RQ4). It describes how the method, experimental prototyping, was applied to develop the prototype. This will be followed by a description of the architecture and the main components of the developed prototype. This chapter ends with the presentation of the results of executing different types of fuzzing on a SUT. Chapter 5 provides a general discussion of this thesis related to the research questions. Also, implications for further research and practices are given. Finally, chapter 6 contains a reflection on the research process.

2 Technical background

Knowledge on the applied techniques and used concepts is needed to understand this thesis. Therefore, this section describes those techniques and the main concepts used in this research. Firstly, REST web services, the targeted type of software for this research is described. Secondly, OpenAPI specification (OAS), a description languages (DL) is elaborated. Thirdly, model-based security testing is highlighted. Fourthly, different forms of fuzzing are presented.

2.1 REST web services

A web service is a software solution developed to enable machine-to-machine communication in a network environment. More specifically, a REST web service is a software architectural style for distributed systems that comprises a collection of rules to describe how web services should operate, introduced by Roy Fielding in 2000 [4]. Web services that are compliant with these rules are also called RESTful web services. REST web services provision an application programming interface (API) to create, access, modify, and delete (web) resources through a uniform and prearranged collection of stateless operations in a network environment [4]. These APIs or interfaces can be described by a DL [17, 18]. A DL is the specification for a machine interpretable file that describes the resources provided by a web service and the relationships between those resources [19].

Resources are an important aspect of REST web services, which are entities, that can be identified, modified, or deleted in a network environment by their Universal Resource Locators (URLs). REST web services often use the Hypertext Transfer Protocol (HTTP) to communicate. The communication protocol for REST is not only restricted to HTTP, but most of the time when REST is mentioned, REST over HTTP is actually meant, which also applies to this thesis. When using REST web services, an HTTP request is sent to a Universal Resource Identifier (URI) on a server, to which an HTTP response with an optional payload is returned to the client. This payload can be formatted in Extensible Markup Language (XML) or JavaScript Object Notation (JSON). The response indicates what result is achieved, this can be a notification that a resource is stored, modified, deleted or one or more resources itself can be returned. Information about the result is indicated by the HTTP response status code, e.g. 201 for a resource that is created or 404 when the requested resource cannot be found. The create, read, update, and delete (CRUD) operations are performed with consecutively the POST, GET, PUT, and DELETE HTTP methods. These operations are illustrated with an example resource user in Table 1.

Table 1: Examples of CRUD actions on resource user on a REST web service.

Action	URI	HTTP method
Create a user	/users	POST
Read all users	/users	GET
Read user with id 1	/users/1	GET
Update user with id 1	/users/1	PUT
Delete user with id 1	/users/1	DELETE

The goals of REST are: creating a scalable solution that has simple interfaces, performing well, being able to adapt to changing needs, delivering visible communication between service agents, enhancing the portability of components, and being reliable [4]. These characteristics are realised by the six guidelines that REST prescribes [4]. Firstly, the *client-server architecture style* enforces separation of concerns. A single component can be replaced, just by honouring the existing interfaces. Secondly, the *statelessness of requests*, which entails that when an operation is performed on the server all information should be included in a single request. This enhances the visibility, reliability, and scalability of a system. Thirdly, *clients are able to cache the results*. Per response, the server indicates if that specific response can be cached by the client. Fourthly, a *layered system* is required. A client cannot see beyond the server, multiple layers can be hidden behind it. As a result, scalability is attained, e.g. load balancers can be placed between the layers. Fifthly, *uniform interfaces*, the foundation of REST, ensure that implementations are decoupled from the provided services. Therefore, these components can evolve separately. Sixthly, an optional constraint is presented, namely *code-on-demand*. This allows the response from the server to contain executable code, to temporarily enhance or add functionality to a service.

2.2 OpenAPI specification

The OpenAPI specification (OAS) is a description language (DL). To simplify the development of REST web services, DLs are developed. These DLs help with the automated processing and human understanding of REST APIs. DLs are available to provide a structured view of what information can be used through that API and how to manipulate and obtain that information. For example, these formal languages can aid in providing generated documentation for programmers or tools that can be built around a DL to aid in developing APIs.

Different DLs were developed through time. The Web Services Description Language 2.0 (WSDL) [20] and Web Application Description Language (WADL) [19] were developed and mostly used with Simple Object Application Protocol (SOAP) web services. These DLs are not used much by the industry for describing REST web services. Therefore, new DLs are developed, e.g.: RESTful API Modeling Language (RAML) [21], Open Data Protocol (OData) [22], and OAS [23].

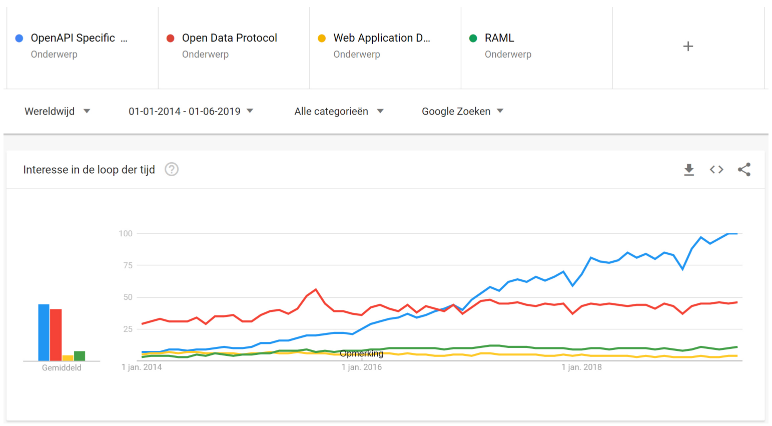


Figure 1: Google trends screenshot displaying the popularity of DLs: OAS (blue), OData (red), WADL (yellow), and RAML (green).

Although, various DLs are available, different sources indicate that OAS is the preferred choice at the moment [24, 25, 26]. Their research is based on, among other things, the GitHub ranking system. All GitHub users can give a star to a project, so active and much appreciated projects get higher ratings. Furthermore, the number of Google searches gives an indication on how often a technique is used. This also applies to the number of hits on Stack Overflow, a public Internet forum that allows users to ask and answer questions on various technical topics. Moreover, Google trends [27], Figure 1, shows an increasing interest in OAS, in contrast to other DLs.

There are different tools available that are built on DLs. Swagger [28] is an example that builds on the OAS. For example, Swagger UI [29] is a library that can parse an OAS file and provide a graphical user interface (GUI) in the browser. This enables viewing documentation, dynamically executing REST calls, and displaying the responses from those calls. Furthermore, server-side tools are available, e.g. Swagger Core [30]. When developing a REST server in a Java server environment, annotations can be used to generate an OAS file from source code. This ensures documentation is up-to-date and not much work to maintain.

2.3 Model-based security testing

When considering security requirements of a software system, security testing can be applied. In case that model-based testing (MBT) focuses on detecting security vulnerabilities, it is called model-based security testing (MBST) [9]. Two primary methods are available, namely security functional testing and security vulnerability testing [31]. Security functional testing can be applied to verify the functionality, efficiency, and availability of the developed security functionalities. Security vulnerability (or penetration) testing is executed to detect and exploit security vulnerabilities in a system by executing an authorised simulated attack on a SUT [32].

At first, MBT was introduced to automate the process of generating test cases. Testing software, in general, can be defined as an activity with the purpose to detect errors while running a software program [33]. The first software testing activities were executed by hand. Because of the increasing complexity of software and the pressure to release software faster, automated testing is being applied more often. Due to the automated generation of test cases that MBT offers, the labour-intensive and error-prone manual processes can be replaced by MBT. As a result, the number of errors in the software will be reduced and the efficiency of the development process will be increased.

In MBT, a (collection of) model(s) is used to automatically generate test cases [9]. The model should be a simpler representation of the SUT, in other words, more abstract. This enhances the ability to check, modify, and maintain the model. [34]. The models used in MBT can be classified into three groups, namely: formal, semi-formal, and informal models [35]. The formal models have a mathematical foundation. Use of these models in the software development industry is very limited, because scaling up to large systems is problematic. In addition, informal models are less applicable due to the absence of a uniform definition, which makes it difficult to model the complexity of modern applications. For these reasons most models are semi-formal, for example UML diagrams or DLs for REST web services. They cannot be used to prove properties mathematically, but they allow structured automated testing of complex applications.

For automated MBT three different components should be present in the model: an interface description, a behavioural model, and deployment information [12]. The interface description is a definition of the available interfaces of the SUT. Such a description should consist of the actions that can be executed on the SUT. Also, input (types) and output (types) should be included in the description. This enables successful execution of those actions and correct processing of the received data. Often Unified Modelling Language (UML) is used to acquire this kind of information, e.g. a class diagram can be used to identify data structures. Furthermore, how the SUT behaves should be captured in a behavioural model. Such a model describes how the SUT interacts with other sources, such as users of the software or other software systems. What actions can occur, and in what order, by a certain actor, are made visible in behavioural models, in UML a sequence diagram can be used for that purpose. Finally, deployment information is needed. This information comprises of information where the SUT is deployed, in the case of a REST web service, the base URL or a link to a specification file conforming to a DL. Deployment data can be stored in the model itself or is added to the test, via configuration, when the test case generation is executed.

In MB(S)T test cases are generated systematically from a model. How these test cases are generated depends on the model and the generation method. The method for MBST of REST web services applied by Atlidakis et al. [15] uses the OAS of a REST web service to create a model. The information on what REST web services are available and what input is needed to successfully execute those web services calls are extracted from the OAS. Furthermore, the output information from the OAS is used to determine the order in which calls can be executed. For example, inserting a blog post can only succeed with an mandatory author, it has to be an existing resource in the SUT. So, an author must be created before a blog post can be created. These dependencies can be derived from the OAS and the REST conventions. All this data is part of the model representing that SUT.

Previous research focused on MBT of REST web services, e.g. Fertig and Braun [36] stated that different types of test cases are needed for REST web services, namely: functional, security, performance, behaviour, and compliance testing. In their study a domain specific language (DSL) was developed to generate test cases. Furthermore, Pinhiero et al. [37] used a protocol state machine to generate test cases for REST web services. In UML, the state machine is a type of diagram that visualises the different states of an object while being executed. Behavioural and protocol state machines are available to express state transitions. The behavioural state machines are used to model single entities, while for REST web services (interaction between multiple resources) a protocol state diagram can be used.

2.4 Fuzzing

Fuzzing or fuzz testing is an automated software testing technique that may reveal faults in software by sending (unanticipated) data to the SUT and monitor how the system responds to this input [38]. This is a form of negative testing [39]. An example of unanticipated data with negative testing is when date typed values are expected (an age in a create user form), but alphanumeric or extreme long strings are sent. Fuzzing is related to boundary value analysis (BVA), a technique in which ranges of values are tested to verify if the systems reacts as expected, focusing on edge cases. In contrast to BVA, fuzzing will not only test edge cases, but aims at any input that can trigger unexpected or insecure behaviour [38].

Fuzzing can be applied in two different ways to find different types of vulnerabilities. Data fuzzing, which is applied the most, consists of sending invalid input to test the correct functioning of input validation mechanisms and parts of the SUT that process the entered data. This form of fuzzing is for example very usable for the detection of injection vulnerabilities. Another form is behavioural fuzzing where the emphasis is on detecting state related vulnerabilities. These can be caused by executing actions in an order that has not been taken into account by the development team, e.g.: an order could be placed, without verifying if the payment was successfully performed. To detect these vulnerabilities a subset of all possible sequences is performed to trigger faults in the SUT [13]. Security vulnerabilities that require manipulation of the state of an application can be tested with this form of model-based (behavioural) fuzzing [15].

There are different ways to classify fuzzing techniques [10]. One way is the categorisation by considering the amount of knowledge the fuzzer has of the SUT, namely: black-box, grey-box, or white-box. A black-box fuzzer does not consider internal logic. In contrast, white-box fuzzers contain a considerable amount of detailed knowledge. Grey-box fuzzers are positioned somewhere in between: some knowledge of the SUT is present, but not in detail. In addition, fuzzers can be typed as mutation-based or generation-based, consequently, the type of data generation is considered [38]. Mutation-based fuzzers start with an input seed and keep mutating that data while presenting it to the SUT. On the other hand, generation-based fuzzers generate data without a seed. These fuzzers can generate data randomly or from a (formal) specification, e.g. a model, or a grammar.

Through the years, the effectiveness of fuzzers has increased significantly, so more vulnerabilities can be detected [10]. The first fuzzer, developed by Barton Miller [11], consisted of a software program that generated random input data. With the increased complexity of software, this approach will not be able to cover much of the search space of complex software solutions. Therefore, much effort is spent on improving the main part of the fuzzing system, the test case generator, which is responsible for creating test cases. Different approaches are available to increase efficiency. The test case generator itself can be improved: instead of using randomly generated tests, a grammar can be used to generate test cases. Generation input by a grammar ensures that the generated input is more likely to bypass validation and still can cause an exception to occur. Furthermore, the test case generator can be provided with runtime information on how the SUT responds to the generated tests, e.g. dynamic symbolic execution, coverage guide, or dynamic taint analysis. Feeding the fuzzer with runtime feedback guides the fuzzer, to improve the code coverage while executing the SUT. Finally, static analysis can be done, also to improve the process of guiding the fuzzer. This can be achieved with source code analysis or creating a model [15].

An additional benefit of applying fully automated fuzzing to detect security vulnerabilities is the possibility to integrate this solution in the SDLC. This can be realised with tools that support the SDLC with continuous integration and continuous deployment, e.g. Jenkins [40]. A common practice is that, after automated building and deploying an application, functional tests are executed automatically. In this phase, also tests for security testing can be executed. Vulnerabilities are detected earlier by facilitating these tests in the SDLC. This will lead to the development of more secure software and potential vulnerabilities can be detected and eliminated before the software is deployed and taken into production [7].

Measuring the effectiveness of fuzzers is often realised by measuring code coverage [10, 41, 42]. This is an essential metric to determine the effectiveness of a particular fuzzing implementation, because it gives an indication on what percentage of the source code is executed [41]. The number of instructions, basic blocks, or routines that are executed during the execution of the test are counted and the percentage of the total of these source code components, determine the coverage [42]. A disadvantage of this method is that it does not indicate how much of the vulnerabilities in the SUT are found. Existing open source projects can be fuzzed and the amount of detected vulnerabilities gives a good impression of the effectiveness in detecting vulnerabilities of the fuzzer. Although, some criticism to this method must be brought under attention, because it is unknown how many vulnerabilities are present in the SUT. Therefore, the results are difficult to compare and judge.

2.4.1 Model-based (behavioural) fuzzing

Research showed that combining different techniques with fuzzing results in effective solutions to find security vulnerabilities. Using the advantages of the combined techniques will lead to improved effectiveness of future fuzzers [10]. For example, Atlidakis et al. [15] created REST-ler, a tool that combines MBT with behavioural fuzzing. Test cases are automatically generated, without human intervention, and also this solution guides the fuzzer efficiently through the search space, by using a model. This solution executes REST calls in different sequences to achieve behavioural fuzzing.

2.4.2 Model-based (behavioural) dictionary fuzzing

Behavioural fuzzing can also be combined with data fuzzing [13], for example with dictionary fuzzing. Such dictionaries contain a list of strings, that will possibly exploit a vulnerability when presented to a SUT. An example is FuzzDB [14], which contains different types of dictionaries, e.g. for the detection of SQL or JSON injection vulnerabilities. Another example of dictionary fuzzing is using dictionaries containing frequently used usernames and passwords to gain entry to a system in a brute force manner. Attack and penetration tools have built-in functionality to execute dictionary fuzzing, such as Zed Attack Proxy (ZAP) [43] or Burp Suite [44].

3 Vulnerability types in REST web services

This chapter presents the first study which focusses on the vulnerability types found in REST web services. The applied method is elaborated in the next section. Subsequently, the results are presented.

3.1 Method: systematic literature review

A systematic literature review was executed to create an overview of the types of vulnerabilities that can be detected in REST web services (RQ1). Therefore, a systematic procedure was executed to identify, appraise and synthesise relevant studies [45]. The advantage of this procedure is that it restricts bias by reducing the amount of data to its essence in verifiable steps.

3.1.1 Search strategy for scientific databases

Firstly, the keywords for this search were determined. An initial keyword list was constructed consisting of the key terms of RQ1. Additional related terms were added from the IEEE Thesaurus 2019 [46] and the ACM Computing Classification System, revision 2012 [47]. Iteratively, the search terms were tested and corresponding results were reviewed. Search terms were dropped when they were too short to be included by some of the search engines (e.g. the abbreviations REST and API) or did not result in relevant results regarding answering RQ1. Table 2 gives an overview of all the keywords that were investigated. The selected search terms were: “web service" and “vulnerability".

Table 2: Overview of the keywords investigated for the systematic literature review.

Initial keyword	IEEE Thesaurus 2019	ACM computing classification, 2012 Revision	Own insight
representational state transfer (REST)	[BT] Software architecture	-	[BT] Client-server systems [BT] Microarchitecture
application programming interface (API)	[BT] Computer interfaces [RT] Software defined networking	-	-
web service OR web services	[BT] Internet Middleware [RT] Asynchronous communication [RT] Cloud computing [RT] Service computing [NT] Message service [NT] Service-oriented architecture [NT] Simple Object Access Protocol (SOAP)	[NT] Simple Object Access Protocol (SOAP) [NT] RESTful web services [RT] Web Services Description Language (WSDL) [RT] Universal Description Discovery and Integration (UDDI) [RT] Service discovery and interfaces	[BT] Interface
vulnerability OR vulnerabilities	-	[RT] Vulnerability management [BT] Penetration testing [BT] Vulnerability scanners	[RT] Susceptibility [RT] Weakness [RT] Threat [RT] Risk [RT] Exploit

BT = broader term | RT = related term | NT = narrow term

Secondly, IEEE Xplore [48], ACM digital library [49], and ScienceDirect [50] were systematically searched using the selected search terms. The search was restricted to the years 2003 up to and including 2019. The concept REST was introduced in 2002, therefore no vulnerabilities were found before 2003. Furthermore, to reduce the number of hits, only the abstracts were searched, since that is the summary of an article (i.e. a short description of the purpose, method and main findings). For comparability, the same search strategy was used for all three databases. Some variation in the formulation of the queries was required, due to technical differences between the databases:

- For IEEEXplore this resulted in the query **(“web services" IN abstract) OR (“web service" IN abstract) AND (vulnerabilit* IN abstract)** and produced 76 results. Double quotes were used to exclude non-relevant results, as “web application". To acquire the single and plural form of web service in the results double quotes had to be used. A wildcard, as used with vulnerability to get single and plural forms did not work for this search engine in combination with double quotes.
- The ACM digital library was searched with query **recordAbstract(+“web services" +vulnerability)** and produced 42 results. Changing from plural to singular forms of the search terms resulted in the same number of results. Therefore, no extra arguments needed to be added to the query.
- ScienceDirect was queried for **“web services vulnerability"** in the abstracts and this produced 116 results. The executed search in ScienceDirect produced the same results, regarding the use of singular of plural forms of the search terms.

The titles and abstracts of these 234 results were exported and converted into a spreadsheet per database and imported in MaxQDA [51]. A study should “focus on web services" and “focus on one or more types of vulnerabilities" to be included in this review, so all titles and abstracts were assessed for these two tags. Only articles that got both tags were selected for further analysis. A total of 23 articles satisfied these constraints. These articles were downloaded and the pdf documents were added to the MaxQDA database.

3.1.2 Synthesis scientific articles

The selected articles were read completely and tagged, so all relevant information of each article was systematically and explicitly collected. The applied tagging system is displayed in Table 3. Reliability was enhanced because a tagging system was used. The content of each article was evaluated on the same criteria and this increased the reproducibility. For most tags deductive tagging was applied with the predefined tagging system. It is not always possible to create an exhaustive list on forehand, therefore inductive (i.e. open) tagging was used to determine the types of vulnerabilities and types of web services incrementally. This means that tags have arisen from the data of the original articles. An additional benefit of this approach is that the terminology of the authors was used which enhances the validity.

Then the inductive tags were grouped into meaningful categories using the OWASP Top 10 (2017) [52]. One additional category “uncategorised" was added for vulnerability types that do not fit existing categories. The OWASP Top 10 is a taxonomy of the ten most prevalent vulnerabilities in web applications. This list is based on a large amount of data from several organisations around the world and upon community feedback. Grouping these vulnerabilities in categories made it possible to compare and verify the results of literature (theory) with additional evidence of the National Vulnerability Database (practice), see section 3.1.3. The categories were prioritised based on the frequencies of corresponding vulnerabilities found in literature.

In Appendix A: Results literature review multiple tabulations (Tables 11, 12, and 13) are presented to enhance the transparency of this literature review [45]. These tabulations provide an overview of all the included articles (i.e. reference, title, authors, year of publication, keywords, research questions, type(s) of web service, type(s) of vulnerabilities, research method and a short description for each of these articles).

Table 3: The tagging system used with MaxQDA.

Tag group	Tags
Article inclusion	focus on web service, focus on one or more vulnerabilities
Meta information	title, authors, year of publication, keywords, research questions, extra information
Study types	case study/series, experiment (cause effect), literature review
Web service types	JSON-RPC, REST, RPC, SOAP, stateful, XML-RPC
Vulnerability types	access validation, authentication, authorization, unavailability, buffer overflow, code execution, command injection, confidentiality and integrity vulnerability, cross-site scripting (XSS), denial of service (DOS), error on interface, forcing error, invalid parser, invalid XML, logging vulnerability, parameter tampering, password in clear, script injection, server path disclosure, service traversal, session hijacking, session replay, spoofing, SQL injection, WSDL scanning, XML injection, XML rewriting attacks, XPath injection

3.1.3 Verification with additional evidence

The results of the literature review were verified with another source to support the findings from literature. Also, it supports that the outcome of this literature review also applies to REST web services and not only to web services in general. Additional validity evidence was collected from the National Vulnerability Database (NVD) [53]. The NVD is a repository containing vulnerabilities maintained by the National Institute of Standards and Technology (NIST) [54]. This database will give a complete and up-to-date overview of vulnerabilities found in practice. The NVD retrieves vulnerabilities from the Common Vulnerabilities and Exposures (CVE) database [55] and software security experts enrich those vulnerabilities (e.g. by adding impact metrics and vulnerability types). Various CVE Numbering Authorities (CNAs) are granted to add CVEs to the database. This process enhances the reliability of the data and therefore the NVD was a useful source for verification.

Querying the database via the website for the terms REST or RESTfull did not lead to acceptable results. Therefore, another solution was chosen. The vulnerabilities from this database are also presented in JSON feed files. All JSON feed files [56], with all vulnerability entries grouped per year, from 2002 until 2019 were downloaded. A small JAVA program was written to search and group the entries to create a comprehensive overview. In this program different actions were executed. Firstly, entries were included when the description matches this JAVA regular expression: `(?i:.*[^\\w](rest|restful)[^\\w].*)`. This resulted in inclusion of all the entries in which the descriptions contained the text 'rest' or 'restful' preceded and followed with a non-word character. The matching process was executed case insensitive and resulted in 277 vulnerability entries. After reviewing the results, a filter was applied to exclude all entries containing the string ' the rest of the '. As a result, 13 entries were excluded. Finally, the remaining entries were reviewed again by the researcher and another 5 were removed, because they were File Transfer Protocol (FTP) related items. In the context of FTP, Restart of Interrupted Transfer (REST) is a FTP command. Only information from the years 2009 until 2019 was selected, because before 2009 no REST related vulnerabilities were registered in the NVD.

Also, the Common Weakness Enumeration (CWE) [57] information was extracted from the JSON files. This information was used to group the vulnerabilities by type, see Table 14 in Appendix A: Results literature review. To compare these results with the results of the literature review, the discovered vulnerability types were also categorised in the OWASP Top 10 categories.

3.2 Results

The first research question is addressed in this section: What types of vulnerabilities can be detected in REST web services? In literature, 29 types of vulnerabilities were found based on 23 articles, see Tables 11, 12, and 13 in Appendix A: Results literature review for all details. Furthermore, in the NVD 49 types of vulnerabilities were found, see Table 14 in Appendix A: Results literature review. Table 4 provides an overview on vulnerability types in web services found in literature and the NVD with a categorisation based on the OWASP Top 10.

Table 4: Categorised vulnerability types found in literature and the NVD for (REST) web services.

OWASP Top 10 categories	Literature			NVD		
	Rank	Count	Vulnerability types	Rank	Count	Vulnerability types
A1 Injection	**1**	**32**		**4**	**13**	
		14	SQL injection		9	SQL injection
		7	XML injection		4	command injection
		5	XPath injection			
		2	code execution			
		1	script injection			
		1	invalid XML			
		1	invalid parser			
		1	parameter tampering			
A2 Broken authentication	**2**	**6**		**1**	**22**	
		3	credentials exposure		16	improper authentication

continued ...

OWASP Top 10 categories	Literature			NVD		
	Rank	Count	Vulnerability types	Rank	Count	Vulnerability types
		2	broken authentication		3	session fixation
		1	session replay		2	insufficiently protected credentials
					1	insufficient session expiration
A5 Broken access control	**3**	**3**		**2**	**22**	
		1	session hijacking		9	path traversal
		1	authorisation issues		7	access control issues
		1	inadequate or missing access validation		6	improper access control
A7 Cross-site scripting	**4**	**3**		**3**	**22**	
		3	cross-site scripting		22	cross-site scripting
A10 Insufficient logging & monitoring	**5**	**1**		**-**	**-**	
		1	logging vulnerability			
A4 4 XML external entity reference	**-**	**-**		**5**	**9**	
					9	information leak through XML external entity file disclosure
A3 Sensitive data exposure	**-**	**-**		**6**	**5**	
					2	missing encryption of sensitive data
					2	cleartext storage of sensitive information
					1	cleartext transmission of sensitive information
A8 Insecure deserialization	**-**	**-**		**7**	**3**	
					3	deserialisation of untrusted data
Uncategorised	**-**	**22**		**-**	**165**	
		5	denial of service		32	information exposure
		4	spoofing		22	improper input validation
		2	XML rewriting		20	permissions, privileges, and access controls
		2	buffer overflow		19	insufficient information
		1	unavailability		17	cross-site request forgery
		1	confidentiality and integrity vulnerability		4	credentials management
		1	error on interface		4	incorrect authorization
		1	server path disclosure		4	failure to sanitize data into a different plane
		1	forcing error		4	other
		1	services traversal		3	cryptographic issues
		1	information exposure		3	resource management errors
		1	WSDL scanning		3	inconsistent interpretation of HTTP requests
		1	XML signature wrapping		2	failure to constrain operations within the bounds of a memory buffer
					2	7PK - security features
					2	permission issues
					2	uncontrolled resource consumption
					2	incorrect permission assignment for critical resource
					2	uncontrolled memory allocation
					2	missing authorisation
					2	failure to control generation of code
					1	deprecated
					1	integer flow or wraparound
					1	improper privilege management
					1	incorrect default permissions
					1	improper verification of cryptographic signature
					1	race condition
					1	improper resource shutdown or release
					1	unrestricted upload of file with dangerous type
					1	information leak through log files
					1	improper link resolution before file access

continued ...

OWASP Top 10 categories	Literature			NVD		
	Rank	Count	Vulnerability types	Rank	Count	Vulnerability types
					1	exposed dangerous method or function
					1	allocation of resources without limits or throttling
					1	inclusion of functionality from untrusted control sphere

The four most important OWASP Top 10 categories of vulnerability types in web services based on literature will be presented in the next sections: injection, broken authentication, broken access control, and cross-site scripting. The top four ranked OWASP Top 10 categories found in the NVD support and verify the four categories found in literature, only in a different order, namely: broken authentication, broken access control, cross-site scripting and injection, see Table 4 for the details.

Twenty articles in this literature review focus solely on SOAP web services, one on stateful, and two on a combination of web services namely SOAP, REST, and RPC protocols. The results of the NVD are only vulnerabilities related to REST web services and have shown that the amount of REST web services vulnerabilities in the NVD increases over time, both in numbers and as a percentage of the complete number of registered vulnerabilities (see Table 14 in Appendix A: Results literature review). In the next paragraphs a short description of the corresponding OWASP category will be given. Thereafter, the most mentioned vulnerability types within that category will be defined, possible consequences of exploitation will be described, and will be illustrated with vulnerabilities in REST web services found in the NVD.

3.2.1 Injection

The category *injection* contains eight types of vulnerabilities found in literature, see Table 4. Having injection vulnerabilities can lead to sending non-validated data to an interpreter by means of a command or query. This can result in execution of commands or bypassing authorisation by a malicious actor [52]. The most frequent mentioned vulnerability types in literature are SQL injection, XML injection, and XPath injection. Especially, SQL injection is a risky vulnerability in web services, because it can change the SQL queries that are executed on the database [58, 59, 60]. For example, in the NVD, a SQL injection vulnerability was registered for Apache Fineract. This vulnerability made it possible to inject SQL via its REST web service, by directly appending existing queries via the parameters ‘orderBy’ and ‘sortOrder’ [61]. Consequently, SQL injection can lead to the theft of user information [62]. With XML injection, parts of XML messages are added and/or modified, by appending or alternating user input, which results in XML messages that are harmful [63, 64]. This can result in the collection and manipulation of data [65]. Comparable to SQL injection, when XPath injection is applied, malicious actors can retrieve or manipulate data by changing queries. Only with XPath injection the targets are XML oriented databases, in which data is stored in XML documents [58].

3.2.2 Broken authentication

The category *broken authentication* consists of three types of vulnerabilities found in literature, see Table 4. These vulnerability types are related to the flawed implementation of authentication and session management. These types of vulnerabilities allow malicious actors to obtain passwords, keys, or session tokens. Also, other erroneous implementations can lead to take over user identities for some time or enduringly [52]. The most frequently mentioned vulnerabilities in literature are credential exposure and broken authentication (similar to the category name). Credential exposure occurs when the response of the web service reveals credentials of the user [5]. An example is 'password in clear', when the web service reveals the password of the user, because it is sent unencrypted [58] or valid usernames are revealed by inspecting behaviour of the response from logging in on a web service [66]. During an authentication attack, the mechanism and methods used to authenticate the software system are attacked [62]. Examples of such an attack are: password exposure or phishing attacks [62]. The main risk of broken authentication is that third parties, without the identity of the user, have access to the system [62, 67]. In the NVD, a case of broken authentication led to the possibility for an unauthenticated malicious actor to bypass the authentication of Cisco Elastic Services Controller. By sending a specially created request to the REST web service, an attacker could execute administrative privileged actions [68].

3.2.3 Broken access control

The category *broken access control* comprises three types of vulnerabilities in literature, see Table 4. All these vulnerabilities are related to incorrect implementation of access management. Exploitation of these types of vulnerabilities lead to accessing program functionalities or data by an unauthorised user [52]. These vulnerability types are session hijacking, authorisation issues, and inadequate or missing access validation. With session hijacking a malicious actor monitors and reads network traffic. Then the session key or cookies are stolen to access a valid web session. The consequence is that a malicious actor gets unauthorised access to the web server [59]. Authorisation issues, by other authors called, inadequate or missing access validation are problems with access policies, used to express which authenticated users can access whatever resources [66, 67]. These issues are caused by resources that are missing proper access control mechanisms [66]. This can result in data leakage [66]. A vulnerability of the type broken access control registered in the NVD led to enumeration of usernames in some versions of Jira. An incorrect authorisation check on the REST web service (/rest/api/2/user/picker) disclosed if the supplied username was known to the system [69]. Valid usernames acquired by this form of enumeration can then be used to guess their corresponding passwords. Guessing the password has a greater chance of success now one of the two variables is confirmed correct. Furthermore, a list of earlier leaked username and password combinations can be consulted for occurrences of these known usernames.

3.2.4 Cross-site scripting

The category *cross-site scripting* includes the vulnerability type ’cross-site scripting’ in literature, see Table 4. According to the OWASP Top 10, it refers to the inclusion of malicious code into a web page caused by improper validation or escaping. A cross-site scripting (XSS) vulnerability is successfully exploited when the malicious code injected by a malicious actor is executed in the browser of the victim [52]. As described by Salas and Martins [70], the main purpose of XSS: ‘is to store, modify, or delete requests, misleading the servers and the user of the web services’. Consequently, sensitive information might be stolen and the systems integrity might be compromised [70]. Also, unauthorised access may be achieved [66] or the user can be redirected to a special constructed website by the attacker [62]. An example of a XSS vulnerability, found in the NVD, is a version of Silver Peak EdgeConnect SD-WAN was susceptible for XSS. Via a crafted URL, malicious JavaScript could be executed in the browser by users of the REST web service [71].

3.2.5 Uncategorised

In literature, *uncategorised* contains thirteen types of vulnerabilities that could not be classified to one of the OWASP Top 10 categories, see Table 4. The most commonly mentioned vulnerability types are denial of service and spoofing. A denial of service (DOS) attack means that a malicious actor makes an attempt to overload the resources of the victim. This can lead to a reduced or a denied access for valid users to the victim’s web service [72]. As a result, this leads to ineffectiveness, a loss of profit, or even reputational damage of the victim’s organisation [72]. REST web services are also susceptible for DOS vulnerabilities. For example, in the NVD was registered that systems installed with specific versions of the cluster database management component of Cisco Expressway Series Software and Cisco TelePresence Video Communication Server Software were vulnerable to DOS attacks. By sending a crafted request to the REST web service a restart of the system could be forced, resulting in a temporary unavailability of the service [73].

In literature, several forms of spoofing were distinguished. The general characteristic of these forms is that a malicious actor pretends to be someone or something else [74]. Principal spoofing is a form of spoofing in which a malicious actor pretends to be a user to access a web service [75]. Other forms of spoofing are WSDL spoofing (i.e. serving a modified WSDL) and security policy spoofing (i.e. removing or modifying security requirements) [64].

4 Developing a model-based (behavioural) dictionary fuzzer

This chapter presents the second research of this thesis which focuses on the development of a model-based (behavioural) dictionary fuzzer to detect (SQL injection) vulnerabilities. The results of the literature review showed that the most occurring type of vulnerabilities in REST web services is injection and then in particular SQL injection. Therefore, the focus in this research will be mainly on detecting SQL injection vulnerabilities.

This section is structured in four parts. Firstly, the research method is presented. In this subsection is explained how the prototype is developed. Secondly, a description of the architecture is given. The chosen three layered architecture is explained and how it relates to the prototype is presented. Thirdly, the main components of the developed prototype are elaborated. The responsibility of each component is given and the relation between the components is explained. Finally, the results of this research are presented.

4.1 Method: experimental prototyping

Experimental prototyping was executed to answer three research questions: How can a model-based behavioural fuzzer be developed that is capable of detecting vulnerabilities in REST web services? (RQ2), How can a model-based behavioural dictionary fuzzer be developed that is capable of detecting SQL injection vulnerabilities in REST web services? (RQ3), and How effective is a model-based behavioural dictionary fuzzer in detecting SQL injection vulnerabilities in REST web services? (RQ4).

Experimental prototyping is a process in which one or more prototypes are developed to be evaluated in an experiment [76, 77]. A prototype is according to Naumann & Jenkins “a system that captures the essential features of a later system (. . .) intentionally incomplete, is to be modified, expanded, supplemented, or supplanted” [78, p. 30]. Thus, in this study, a prototype was developed that was extended vertically, meaning adding functionality through all layers of the system [77]. The prototype needs an environment to test its capabilities on, therefore an experiment was conducted on a SUT [76]. This process of experimental prototyping is characterised by exploring, verifying and validation of the performance of the prototype [76]. The design and implementation of the prototype was adapted based on the results of these tests [76, 79, 80], which enhances the effectiveness of the prototype. This process was executed iteratively, which is visualised in Figure 2. The process of experimental prototyping was stopped when the desired result was reached.

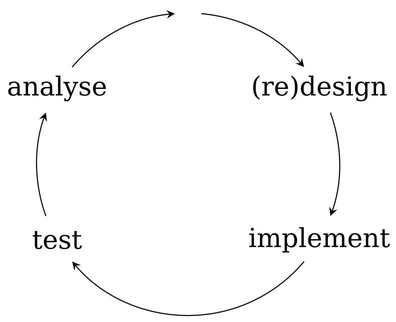


Figure 2: The process of experimental prototyping applied in this study to develop RESTFuzzer [1].

In this study, the desired result is a model-based fuzzer and model-based dictionary fuzzer. These fuzzers should be capable of and effective in detecting vulnerabilities. Capable is defined as sending a high percentage of valid requests, the ability to detect vulnerabilities, and operate fast. Effective is defined as executing a large part of the functionality of the SUT. Therefore, these four aspects were taken into consideration:

- These fuzzers can send a high percentage valid requests to the SUT, which means that the HTTP response codes are in the 200 range [15, 39]. Comparable research of Atlidakis et al. [15] showed that a model-based fuzzer was able to send 80% valid requests. Therefore, in this study anything above 75% will be classified as a high percentage. Consequently, a valid request conforms to the validation rules imposed by the SUT. For REST web services and HTTP traffic in general the status code of the response informs the sender about the result of processing the request. Therefore, response codes can be used to reflect the quality of the requests sent to the SUT. Thus, sending a high percentage valid requests is important because it influences the extent to which a fuzzer succeeds in detecting vulnerabilities in REST web services.

- These fuzzers can detect vulnerabilities. This is difficult to prove because it is unknown whether there are detectable vulnerabilities in a SUT, this also depends on the quality of the SUT. It is possible to know what vulnerabilities do exist in a SUT. Vulnerabilities can be injected in an existing SUT or a SUT with known vulnerabilities can be developed or used. There are open source applications available with built-in vulnerabilities for educational purposes, e.g. OWASP WebGoat [81]. Such a REST web service with vulnerabilities and an OAS was not available.

- These fuzzers can operate fast. This means the number of requests should be sufficient to be able to execute fuzzing. The success of fuzzers relies on executing many requests, therefore the performance of a fuzzer impacts the capability of the fuzzer.

[1]Derived from the iterative prototype experiment cycle by Tronvoll et al. [76].

- These fuzzers are effective in executing a large part of the functionality. Insight is given by measuring source code coverage. Code coverage is an important metric for measuring effectiveness of fuzzers, for more detail see section 2.4. Furthermore, previous research showed that code coverage is an important predictor for the amount of vulnerabilities found, namely, an increase of 1% code coverage leads to finding approximately 1% more vulnerabilities [39].

As part of the experimental prototyping, a SUT was selected, of which the selection process is described in section 4.1.1. Moreover, a SUT with vulnerabilities was developed, see section 4.1.2. The development of the prototype was divided into three phases. Firstly, the prototype was able to extract the REST model description from the OAS and execute model-based behavioural fuzzing, this is described in section 4.1.3. Secondly, the existing prototype was extended to be able to execute model-based behavioural dictionary fuzzing, see section 4.1.4. Thirdly, the effectiveness of the developed prototype was measured, this is elaborated in section 4.1.5. In each of the phases an iterative approach was used ((re)design, implement, test, and analyse) to improve the results of the prototype, see Figure 2. Furthermore, the architecture of the prototype was designed and the main components were defined. The elaboration of the architecture, its components and techniques that were applied are presented in section 4.2 and 4.3.

4.1.1 Selection and installation of the SUT

To be able to execute the experiment a SUT was needed to test the developed prototype on. A query was executed on Google [82] to find a suitable SUT. The following search terms were used: Open source, CMS, Swagger, OpenApi Specification, OAS, REST, and API.

Next, the acquired results were filtered by applying a list of five requirements: the SUT should contain REST web services, a description language (DL) should be available for the REST web services of the SUT, preferably OAS, the source code of the SUT should be available for inspection/modification, source code coverage measurement should be possible, and it should be possible to install the SUT on a personal computer. Accordingly, a SUT was suitable and selected for the experiment. This search from Google yielded five possible SUTs, displayed in Table 5 in order of most used, according to the website BuiltWith [83].

Table 5: Overview systems under test considered for the experiment.

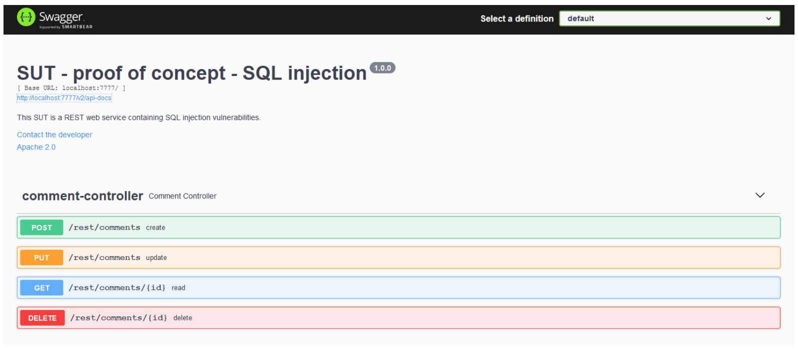
Reference	System under test	Programming language	Live websites according to BuiltWith [83]
[84]	WordPress	PHP	27,021,750
[85]	Drupal	PHP	568,141
[86]	Magento	PHP	190,731
[87]	Mediawiki	PHP	91,817
[88]	Bloomreach	Java	1,254

All these software systems require a web server to serve the (generated) HTML to a client browser, a database to store data, and a way to run the code. XAMPP [89] makes it possible to easily run an Apache webserver, a MariaDB [90] database and a PHP module for Apache. The requirements for all five SUTs were met in theory. The list of possible SUTs was prioritised on impact or in measurable terms: most frequently used. To keep schedule on track and since installing and getting a SUT to completely work requires a lot of time, one SUT was selected. Therefore, the first item on the prioritised list was chosen: WordPress. A running environment was realised to use the SUT for actual development and testing of the prototype. The plugin WP API SwaggerUI [91] had to be installed, to make the OAS available for WordPress. The SUT was successfully installed and running, as a result the OAS schema was available at: http://localhost/wordpress/rest-api/schema. Eventually, an installation of the SUT was also realised on another computer with a Linux distribution (Ubuntu). This was done because of the increased performance, which doubled, in comparison to the development environment. Therefore, all the experiments were conducted on the Ubuntu installation. The (hardware) specifications of the development environment and the test setup are presented in Table 15 and Table 16 in Appendix D: Information related to the experiments.

4.1.2 Developing a SUT containing SQL vulnerabilities

Also, in this study, a SUT was developed that intentionally contained SQL injection vulnerabilities. This SUT will be referred to as system under test SQL injection (SutSqlI) from now on. SutSqlI can be used to validate the capability of detecting vulnerabilities of the prototype. Therefore, a REST web service with create, read, update, and delete (CRUD) actions for comment entities was developed.

SutSqlI is a Spring Boot application [92] and is written in Java and built with Maven [93]. To supply an OAS v2, the third party library SpringFox [94] was used. In addition, SpringFox also supplies a Swagger UI [29], see Figure 3. This makes it possible to execute REST actions of this SUT via the browser.



SUT - proof of concept - SQL injection 1.0.0

[Base URL: localhost:7777/]
http://localhost:7777/v2/api-docs

This SUT is a REST web service containing SQL injection vulnerabilities.

Contact the developer
Apache 2.0

comment-controller Comment Controller

POST /rest/comments create
PUT /rest/comments update
GET /rest/comments/{id} read
DELETE /rest/comments/{id} delete

Figure 3: A screenshot of the Swagger user interface interpreting the OpenAPI specification of SutSqlI containing SQL vulnerabilities.

No PreparedStatements were used to interact with the database, which causes the SQL injection vulnerability in SutSqlI. PreparedStatements are database statements containing placeholders for variables used in queries. These placeholders are commonly filled with user input. The use of PrepraredStatements allows safely including user input in the query by escaping possible unsafe characters, like ' or ;. In this SUT plain queries were used for creating, updating, reading and deleting comments, therefore this SUT is vulnerable to SQL injection. See Listing 1 for the Java DAO class that was created to interact with the database.

```
 1  @Service
 2  public class CommentService {
 3
 4    // variable(s)
 5    @Autowired
 6    private JdbcTemplate jdbcTemplate;
 7
 8    public Comment create(Comment comment) {
 9      jdbcTemplate.execute("INSERT INTO comments (description) values ('"+ comment.getDescription() +"');");
10      Long id = jdbcTemplate.queryForObject("SELECT LAST_INSERT_ID();", Long.class);
11      comment.setId(id);
12      return comment;
13    }
14
15    public Comment read(Long id) {
16      Comment comment = jdbcTemplate.queryForObject("SELECT * FROM COMMENT WHERE ID = " + id + ";", Comment.class);
17      return comment;
18   }
19
20    public Comment update(Comment comment) {
21      jdbcTemplate.execute("UPDATE comments SET description = '"+ comment.getDescription() +"' WHERE id = " + comment.getId() +";");
22      return comment;
23    }
24
25    public void delete(Long id) {
26      jdbcTemplate.execute("DELETE FROM comments WHERE id = " + id +";");
27    }
28  }
```

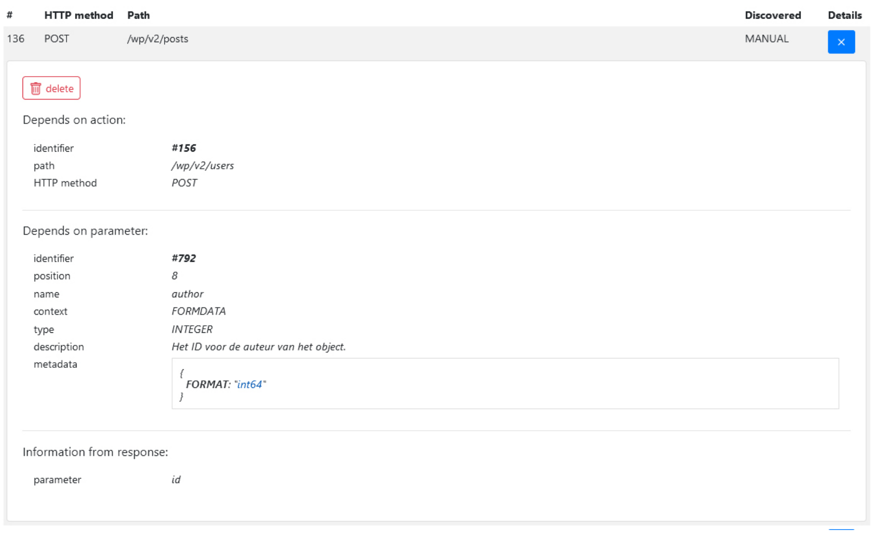
Listing 1: Service class of SutSqlI for interacting with the database.

Note that queries on lines sixteen and twenty-six are not exploitable, due to the fact a Long type is supplied as argument. Therefore, a string value supplied by a malicious actor can never be injected in those queries, because the conversion to a Long number will fail. The exploitation in this example can occur in the queries on lines nine and twenty-one. The description of the comment object is a string that can be injected in the create and update query by a malicious user.

4.1.3 Model-based (behavioural) fuzzing

Model-based fuzzing was implemented in the prototype in different phases. The development started with the implementation of a framework to support model-based fuzzing. In this phase the project was set up and a mechanism to execute long running asynchronous tasks was added. Then, the extractor was built.

Extracting REST model description from the OpenApi Specification The initial version of the extractor was able to obtain basic information from the OAS of the SUT and save that into the database. In the next phases this was incrementally enhanced to gather more information from the OAS. Furthermore, a dependency extractor was developed as a separate process that starts after extracting the data from the OAS. Also, manual adding dependencies between REST web services in the GUI was implemented. For example, creating a new post depends on the existence of an author. The automatic dependency extractor could not recognise the author parameter as being a user entity (according to REST conventions user_id was expected). Therefore, this mapping had to be added manually, see Figure 4. By implementing this mechanism also dependencies can be added that do not follow the REST conventions.



#	HTTP method	Path	Discovered	Details
136	POST	/wp/v2/posts	MANUAL	×

delete

Depends on action:

identifier	***#156***
path	*/wp/v2/users*
HTTP method	*POST*

Depends on parameter:

identifier	***#792***
position	*8*
name	*author*
context	*FORMDATA*
type	*INTEGER*
description	*Het ID voor de auteur van het object.*
metadata	*{ FORMAT: "int64" }*

Information from response:

parameter	*id*

Figure 4: A manual dependency registered in the prototype.

From basic fuzzer to model-based fuzzer Next, the basic fuzzer was developed. The basic fuzzer was able to send requests to the REST API of the SUT and responses were persisted to the database. This basic fuzzer was mainly used to discover what configuration features were needed to create valid requests. Therefore, a configuration mechanism was developed. The configuration started with the inclusion or exclusion of one or more REST actions based on a regular expression. This functionality was added to limit the number of requests by including only certain actions. Also, excluding certain actions prevented the fuzzing process to update the password of the user that was executing the fuzzing process. The configuration mechanism was enhanced with multiple features, in various increments, namely: exclusion of parameters, default values for parameters, and basic authentication. All these features were introduced to get better responses from the SUT, e.g. fewer responses with status codes 400 and 403.

Then, the actual model-based fuzzer was developed. Implementation started with the sequence generator. Since, research of Atlidakis et al [15] showed that execution of sequences in random order yielded the best result, this approach was also chosen for this prototype. However, the actual implementation for sequence generation differs. All possible sequences were generated in advance, then the order of the collection was randomised. The implementation of this fuzzer allowed to specify some parameters, e.g. the maximum number of requests sent for that fuzzing project.

The developed sequence generator starts with determining all possible combinations (sequences) of these actions. For example, given a SUT that has three different actions (*a*=3) and a maximum sequence length of two (*sMax*=2). Also, action a2 has a dependency on action a3, in other words action a3 has to be executed before action a2. In general, the number of possible sequences can be expressed as: $\sum_{s=1}^{sMax} a^s = numSeqs$, in which sMax is the maximum sequence length, a is the number of actions, and numSeqs is the number of possible sequences. This results for this example in $\sum_{s=1}^{2} 3^s = 12$ possible sequences. However, not all these sequences are valid (due to the dependency), the sequence generator only adds valid sequences and would end up with seven valid sequences, see Table 6.

Table 6: All possible valid and invalid combinations of actions (sequences) for the supplied example.

Sequences with length = 1		Sequences with length = 2	
valid	invalid	valid	invalid
[1]	-	[1,1] [1,3]	[1,2]
-	[2]	-	[2,1] [2,2] [2,3]
[3]	-	[3,1] [3,2] [3,3]	-

Experiment (WordPress) Two types of experiments were conducted. The first experiment was executing basic and model-based fuzzing on the SUT configured with a maximum of 5,000 requests with a basic configuration and an optimised configuration, see respectively Listing 4 and Listing 5 in Appendix D: Information related to the experiments. The basic configuration only ensured the fuzzing process keeps functioning by excluding REST web services that could lock out the fuzzer, by invalidating the username and password combination used by the fuzzer. The optimised configuration was established iteratively, by executing a task, analysing the results, and adapting the configuration. The same basic and optimised configuration were used in all conducted experiments. The goals of this experiment are to demonstrate the ability of the model-based fuzzer to generate and execute valid requests and to determine code coverage. The reason to maximise the number of requests to 5,000 was threefold. Firstly, the distribution of HTTP status codes turned out to be constant in various experiments. As a result, the percentages of valid requests were almost equal in those experiments after sending for example 5,000 or 10,000 requests. Therefore, 5,000 requests in one experiment was enough to establish the percentage of valid requests. Secondly, the code coverage percentage increased fast at the beginning of the test, but remained almost constant after a certain number of requests. Therefore, increasing the maximum number of requests would not give new insights. Thirdly, the experiment was executed various times to establish the optimised configuration, so execution time for 5,000 requests was acceptable and processing the code coverage data was doable for that amount of requests.

The second experiment was a long running model-based fuzzing task with optimised configuration and a maximum of 500,000 requests. Time was measured in both experiments to be able to present data about the speed of the different types of fuzzers. All invalid responses (with HTTP response code not in the 200 range) were analysed by the researcher to determine if a vulnerability was detected. The goal of this long running experiment was to detect vulnerabilities. Especially, response codes in the 500 range are worth investigating, since this indicates that a server error has occurred. This can indicate a functional error or a vulnerability. The number of 500,000 was chosen to cap the maximum running time of the experiment. Furthermore, it is not expected that running the experiment longer would trigger new functionality, since code coverage was almost constant after 5,000 requests.

4.1.4 Model-based (behavioural) dictionary fuzzing

Lastly, the model-based (behavioural) dictionary fuzzer was developed. Firstly, managing dictionaries was realised in the prototype. Support for any type of dictionary was built in. Various types of dictionaries can be found in FuzzDB [14], a GitHub open source project. For example, a JSON injection dictionary can contain: `{}`, `{"1":"0"}`, or `{"0":"\x00"}`. Via the GUI a SQL dictionary was created, the items were extracted from FuzzDB's SQL list. Only SQL dictionary fuzzing was applied in this experiment in order to answer RQ3, but the prototype can also be used with other types of dictionaries, e.g. path traversal or JSON.

A new fuzzing project was created in the prototype and configured as type model-based dictionary. The interface then displayed an option to select one or more dictionaries to include in the fuzzing attack. The MySQL exploit dictionary from FuzzDB [14] was used to detect SQL injection vulnerabilities, since the database used for the SUT is MariaDB. MariaDB is a fork of the original open source MySQL database, originated after MySQL was bought by Oracle.

The model-based dictionary fuzzer iterated all selected actions. A selection of the available REST web service calls can be made by specifying a configuration, see section 4.3.2. For each of these actions the parameters of these actions were also iterated. This number of parameters could be limited to a maximum via the configuration to decrease execution time. For every selected parameter a value from the dictionary was randomly selected. This process was repeated until all the items from the dictionary were selected or until the maximum value specified in the configuration was reached. Then each request was executed on the SUT and the response was captured and persisted in the database.

Experiment (WordPress) Two types of experiments were conducted with the model-based dictionary fuzzer on the REST web services of WordPress. The first experiment was executing basic and model-based dictionary fuzzing on the SUT configured with a maximum of 5,000 requests with a basic configuration and an optimised configuration. The second experiment was a long running model-based dictionary fuzzing task with optimised configuration and a maximum of 500,000 requests. Time was measured in both experiments to be able to present data about the speed of the different types of fuzzers. All invalid responses (with HTTP response code not in the 200 range) were analysed manually by the researcher, to determine if a vulnerability was detected.

Experiment (SutSqlI) An experiment was conducted on SutSqlI to prove these SQL vulnerabilities can be detected by the developed prototype. By using the model-based capabilities the read, update, and delete action can be executed. For those actions to be successfully executed an existing comment must be used. Also, for this experiment the MySQL exploit dictionary from FuzzDB [14] was used to detect SQL injection vulnerabilities. The number of requests will be capped at 5,000. But due to the relative simplicity of this application, that number won't be reached. All possible combinations of actions, parameters and dictionary items were executed in this experiment.

4.1.5 Measuring effectiveness

To determine the effectiveness of the prototype in this study code coverage was measured. Measuring code coverage is a method to get insight in how many and sometimes also which parts of the source code were executed during the execution of the fuzzing process [95]. It is a method often used in literature [39]. Two related terms are often used with software security: vulnerability coverage and code coverage. Vulnerability coverage can be described as the number of vulnerabilities found in the source code. Code coverage is, in the context of static analysis, the amount of code that is analysed or scanned to detect vulnerabilities in a system. For example, this is applied by Vieira et al. [5]. In this experiment code coverage is defined as measuring the amount of lines of code that are executed, since analysis is carried out on a running software application. This approach was also applied by Atlidakis et al. [15]. Also, a distinction was made between code coverage for the attack surface and the complete application. The attack surface is a part of the application that is directly accessible by malicious actors [39].

The options for how to execute source code coverage measurement were limited. This was due to the fact that the SUT is built in PHP and served via an Apache web server. For a SUT built in Java or .NET, a OWASP supported solution, Code Pulse [96] is available. This is a tool that allows real-time code coverage monitoring, while executing penetration testing activities.

PHPUnit was the first option explored for the analysis of the SUT. This is a framework used to execute functional tests for PHP oriented projects [97]. After some experimenting this option was abandoned. The problem was that this method only gave insight after the fuzzing process was completely executed. For good insight, the code coverage over time was needed.

The underlying mechanism PHPUnit uses to acquire its code coverage information from is Xdebug. Xdebug is a debugger and profiler tool for PHP. It is mostly used by an integrated development environment (IDE) to debug a software solution to detect where and what things go wrong in a program. Xdebug can give detailed information on what PHP files are included during execution and which lines of these files are executed. Therefore, Xdebug was explored and eventually also chosen to measure the code coverage during the fuzzing process.

Firstly, a PHP file was created, see Listing 2. The file was loaded automatically before any other PHP files were loaded, with the `auto_prepend_file` function in the global configuration file for the PHP module (PHP.ini). This was needed for Xdebug to load correctly with the given filter. In this file three functions are available. From the WordPress code, these functions were called. The first function `startCoverage` on line 6 was called before the REST call was executed. After the response was generated and served to the prototype the function `dumpCoverageInfo` was called, to save the coverage data to a file on the hard disc drive. All the REST API related functionality in WordPress could be found in directory `<WORDPRESS_ROOT>/wp-includes/rest-api/`, the main class was located in `class-wp-rest-server.php`. The functions to create and save the source code coverage data were injected in that class.

```
 1  <?php
 2    xdebug_set_filter( /* this block of code was eventually deleted */
 3      XDEBUG_FILTER_CODE_COVERAGE, XDEBUG_PATH_WHITELIST, [ ’c:\\xampp\\apps\\wordpress\\htdocs\\wp-includes\\rest-api\\’ ]
 4    );
 5
 6    function startCoverage() {
 7      xdebug_start_code_coverage(XDEBUG_CC_UNUSED);
 8    }
 9
10    function dumpCoverageInfo() {
11      $file = fopen(getFilename(), "w");
12      $data = xdebug_get_code_coverage();
13      fwrite($file, json_encode($data));
14      fclose($file);
15    }
16
17    function getFilename() {
18      $now = DateTime::createFromFormat(’U.u’, microtime(true));
19          return "c:/xdebug_dump/" . $now->format("Y_m_d_H_i_s_v");
20    }
21  ?>
```

Listing 2: PHP code to obtain code coverage data from Xdebug and save it to the hard disc drive.

As a result, for each request executed on the REST API of the SUT a file was created with code coverage data from Xdebug. In these files, the names of all included PHP files were listed. For each of these files a multi-dimensional array held all line numbers and per line a value `1` or `-1` to respectively indicate if a line was executed or not. This data only contained information on the code coverage of a single request. To get insight in the code coverage over time, while the fuzzing process was executed, this data needed to be summarised. For this goal the Reporter, see section 4.3.2, was developed.

Whitelisting could be achieved by activating lines from 2 until 4 from Listing 2. This resulted in keeping track of the source code coverage of only the files defined on line 6. These files can be marked as the attack surface of the SUT. Applying whitelisting would have resulted in only measuring code coverage for the attack surface of the application. An advantage of this approach was shorter response times, as omitting the whitelisting resulted in gaining a performance penalty. Also, the coverage data files were smaller in size on disc (∼500kB versus ∼50kB). Reports generated by the Reporter also took considerably less time to create. Finally, no whitelisting was applied in the experiment. So, all PHP files were included in the code coverage measurement. Performance issues were mitigated by installing WordPress on a Ubuntu operation system, which boosted performance. The number of requests per second were doubled.

Experiment (WordPress) For both, the model-based and model-based dictionary fuzzer, code coverage was measured in an experiment. This way effectiveness of both types of fuzzers could be compared. The coverage for all loaded PHP classes and for only the attack surface (PHP classes in the `<WORDPRESS_ROOT>/wp-includes/rest-api/` directory) was presented. This distinction was made in the reporter tool. The maximum requests to send was capped at 5,000 and the optimised configuration was used.

4.2 Architecture

This section describes the architecture of the prototype.

4.2.1 Global overview

The developed prototype is a web application, i.e. a Java Maven [93] project. Maven is a tool that helps with building the project and managing third party libraries used in a Java based project. The project consists of two modules, namely: backend and frontend. The backend module is among other things responsible for: running background tasks (e.g. the actual fuzzing process), saving and providing data from the database (e.g. extracted OAS information, requests send, and responses received) and extracting a REST model description (RMD) from an OAS file. Furthermore, the frontend module delivers a graphical user interface (GUI) for the users of this prototype. In contrast to a command line tool, a GUI offers easy access to and operation of all functionalities. The process and results of the fuzzing process are presented to the user in clear overviews, to help with the interpretation of the data.

The architecture of this prototype is based on a three layered architecture, often used in Java enterprise applications. In this architecture the following principle layers are distinguished: presentation, domain and data source [98]. The structure of the prototype related to this three layered architecture principle is displayed in Figure 5. The presentation layer is realised in the frontend module, this comprises everything related to the user interface. The domain and data source layer are accommodated in the backend module. A service layer is often wrapped around the domain related components, in which security and transaction control can be imposed [98], which is also applied in the prototype. The domain layer in the prototype comprises of the modules: service layer, task executor, RMD extractor, fuzzer, and reporter. All business logic is encapsulated in those components. The data source layer is partly present in the data layer. Hibernate [99] is used, which is an Object/Relational Mapping (ORM) framework to manage database connections, persisting, and populating objects. Therefore, in the data layer, only domain models and service classes are present for communication between database and application. Much of the (database) functionality needed in the data layer is provided by Hibernate.

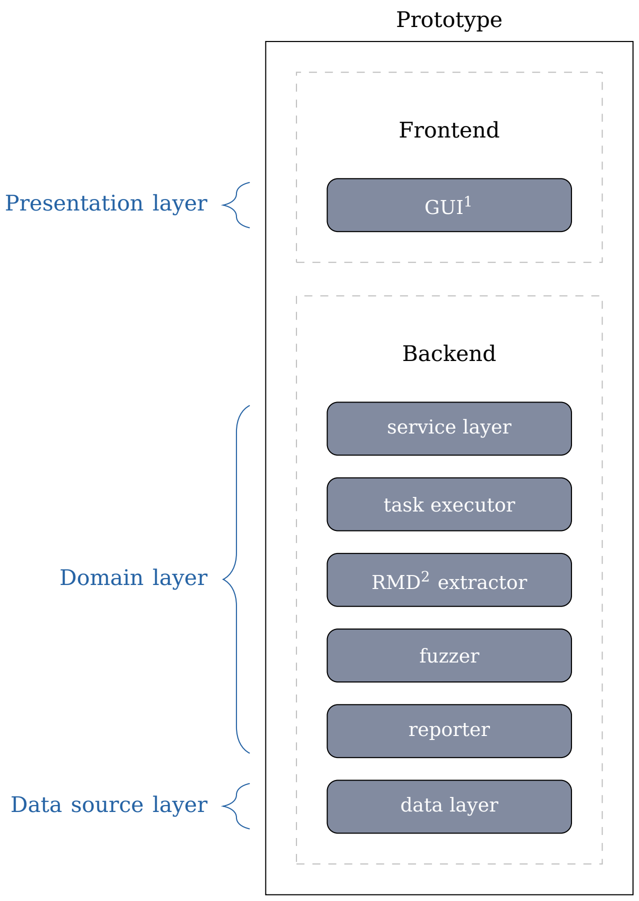


Figure 5: An overview of the modules (frontend and backend) and components (GUI[1], service layer, task executor, RMD[2] extractor, fuzzer, reporter and data layer) of the prototype and their positioning in the three layered architecture.

The backend is written in the programming language Java. Spring Boot [92] is used as framework to aid in focussing on writing features instead of infrastructure. The backend consists of six components: service layer, data layer, task executor, RMD extractor, fuzzer, and reporter. The two components, service layer and data layer, which are architectural components, are elaborated in the next two subsections.

4.2.2 Service layer (RPC-JSON)

The responsibility of the service layer is to exchange data with the frontend. A HTTP request from the frontend is sent to the service layer by means of executing a remote procedure call (RPC) in the JSON format. The service layer delegates any requests for data to the data layer. For interaction between the service layer and data layer, data transfer objects (DTOs) sent by the clients (browsers) are converted to domain objects and vice versa.

The Spring framework supplies all the building blocks to built REST web services and these are also used for this prototype. Since, the web services of this prototype are not fully compliant to the REST conventions, they are called RPC calls using JSON. The service layer consists of different types of building blocks, namely: controllers, mappers, and DTOs. Entry points for the controller are defined in controllers. A class is annotated with `@RestController()` and `@RequestMapping("<PATH>")` to notify the Spring framework this class serves as REST controller and what the base path is for this controller. All accessible methods in the class for the web service are annotated with `@RequestMapping(path = "<PATH>", method = RequestMethod.<METHOD>)`. This makes the method available for a certain URL and HTTP method. Furthermore, mappers and DTOs are used. Mappers are Java classes that convert DTOs to domain objects and vice versa. The DTOs contain information in a form usable for the frontend. These DTOs are sent via the Internet in JSON format and the conversion to JSON is handled by the Spring framework.

4.2.3 Data layer

The data layer is responsible for storing and retrieving the data from the database. This layer comprises three components: data access objects (DAOs), domain classes, and factory classes. The DAOs are interfaces extending Spring's CrudRepositories. These CrudRepository classes support CRUD (create, read, update, and delete) functionality. The extending DAOs are provided with `@Query` annotations to execute complex queries. These DAOs are implementations of a design pattern that hides the underlying database and its dependencies. Using another database type, would only require changes in the DAOs or in this case no changes at all, because Spring's CrudRepositories are used, which are compatible with many different database types. Eventually these are also facades for Hibernate, the unlaying ORM. Therefore, only minors tweaks are needed if database dependant queries are used.

[1] Graphical User Interface
[2] REST model description

The domain objects are Java classes that describe the database structure (fieldnames, types, validations, and relations to other objects/tables). Java annotations are used to map Java types to database types (i.e. `java.util.String` → `varchar(32)` and `java.util.Boolean` → `tinyint(1)`). Other annotations are used to impose constraints (i.e. `@NotNull` or `@NotEmpty`), or to indicate the type of relations objects have (i.e. `@OneToMany` or `@ManyToMany`).

Also, factory classes that implement the Factory design pattern are used. This enables the decoupled instantiating of objects, instead of using the class constructors directly [100]. As a result, the SOLID (Single-responsibility principle, Open–closed principle, Liskov substitution principle, Interface segregation principle, and Dependency inversion principle) principle of programming are applied [101].

4.3 Components

This section describes the main components of the developed prototype. A global overview of interaction between backend, frontend and SUT is displayed in Figure 6. A more detailed description of these two modules and its components is presented in the next paragraphs.

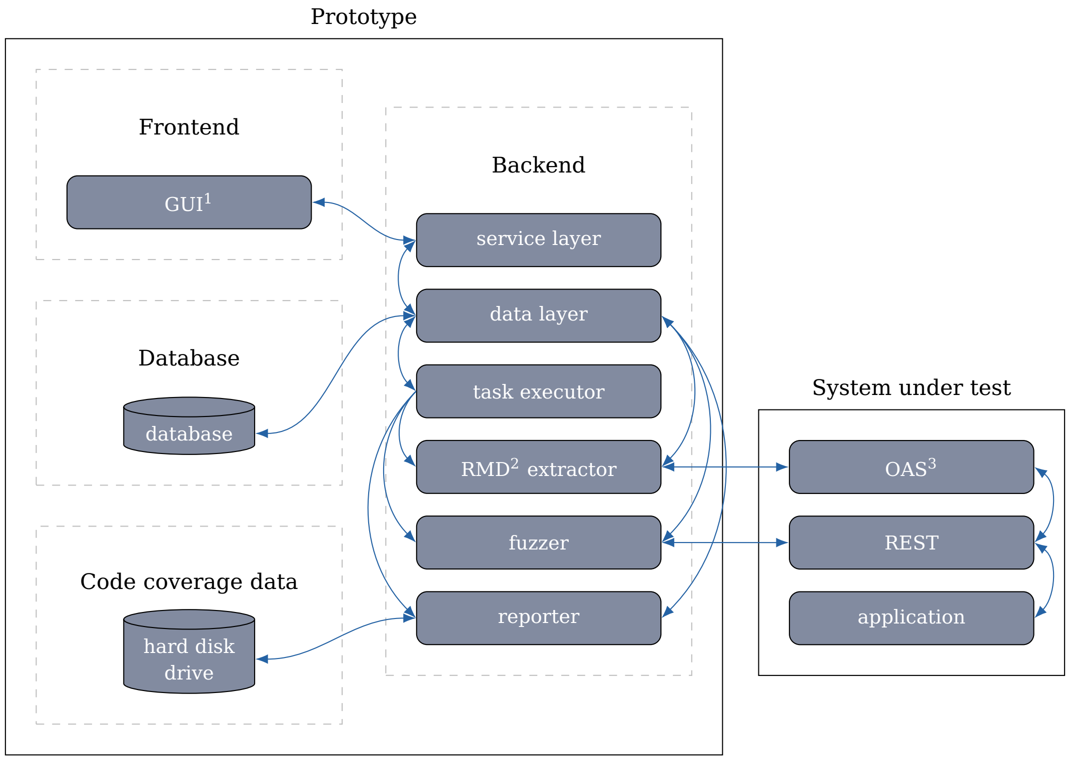


Figure 6: An overview of the modules and components of the prototype and their interaction with a SUT.

[1]Graphical User Interface
[2]REST model description
[3]OpenAPI Specification

4.3.1 Frontend module

The frontend module consists of one component, the graphical user interface (GUI). In the next section a global overview of the functionality of the GUI and how it was realised is presented.

Graphical user interface The functionality of the prototype is grouped into four parts in the GUI: *System under test*, *Fuzzing*, *Reports* and *Tasks*. These parts are accessible from the main menu via links with corresponding names. Executing a fuzzing task starts with adding a new SUT to the system. A SUT is added by entering the URL to the OAS. Then, in the detail panel from the added SUT, an extract task can be started. All relevant information from the OAS is extracted and saved into the database. While this process is running, the task is displayed in the tasks menu. After completion, an overview from actions, parameters, and dependencies is available from the detail panel of the selected SUT, see Figure 21 in Appendix C: Screenshots RESTFuzzer. From the tab that is displaying the automatically extracted dependencies, also manual dependencies can be added.

The next step is creating a fuzzing project in the GUI. A prerequisite is that a configuration is available. So, first a configuration should be added to the system, see Figure 22 in Appendix C: Screenshots RESTFuzzer. A configuration consists of a name and a JSON formatted block holding the actual configuration. An empty block is displayed for the user to supplement. After the configuration is added, the fuzzing project can be added. A fuzzing project consists of a description, type (basic, dictionary, model-based, or model-based dictionary), configuration, maximum number of requests, and a SUT. Depending on the type of fuzzer that is chosen extra options are displayed. If a dictionary typed fuzzer is selected, one or more dictionaries must be selected. The maintenance of dictionaries is also available via the GUI, see Figure 23 in Appendix C: Screenshots RESTFuzzer.

When the project is saved into the system, a fuzzing task can be started. While the long running tasks is executing, progress can be monitored via the tasks link in the menu. After completing the task, the detail page of the project refreshes automatically and the the results are displayed. A tab with responses and requests will be displayed, presenting all the requests that are sent by the fuzzer and all the corresponding responses received are also visible in another tab. If model-based fuzzing is executed an extra tab with sequences is displayed, see Figure 24 in Appendix C: Screenshots RESTFuzzer. It is possible to view a sequence, see what requests were sent in that sequence and what the corresponding responses were. From a response, the corresponding request is visible. Filter options are available in each tab to display only the information that is interesting, e.g. only the responses in the HTTP 200 range.

Also, two types of reports can be created via the GUI, see Figure 25 in Appendix C: Screenshots RESTFuzzer. The first report gives insight in the HTTP response codes that were received after executing a fuzzing task. The second report gives insight in the code coverage. A report consists of a description, type (responses or code coverage), project, interval, grid x-as, and grid y-as. After adding a new report to the system via the GUI, a long running task can be started to generate the report. The result is a LaTeX formatted file containing graphs to visualise the type of HTTP response codes and code coverage during a fuzzing task.

The long running tasks can be monitored in the GUI. An overview with queued and running tasks is presented first, see Figure 26 in Appendix C: Screenshots RESTFuzzer. This overview is refreshed automatically every 1.5 seconds, so all information is up to date. The other overview presents all the tasks that are finished. Each row of both overviews can be clicked on, to display the detailed information of that task.

Some background information on the applied techniques in the GUI is described in this and the next paragraphs. Most modern web applications offer a rich user experience using JavaScript libraries. A GUI allows users to get a clear overview of data present in the system and easy operation of that system, for screenshots of the GUI see Appendix C: Screenshots RESTFuzzer.

The first frontend applications that provided rich user experiences were built with HTML, cascade style sheets (CSS), and JavaScript. The disadvantage of this approach is that these applications were very hard to maintain and contained security issues, e.g. XSS. Therefore JavaScript libraries, like Prototype or JQuery, were developed. These bundled most frequently used functions, e.g. toggle the visibility of HTML elements. This enhances maintainability and security, if the library was frequently updated. As a result, many new libraries were introduced and used in projects, which resulted again in complex, insecure, and hard to maintain applications. So, new libraries were introduced to maintain and bundle all assets (HTML, JavaScript, and images) to decrease maintenance effort. For this purpose Webpack [102] was used for this prototype. At the same time many frameworks that apply structure in frontend applications are introduced, such as React, Angular, or Vue.js [103]. Vue.js is chosen for the development of this GUI.

Bootstrap [104] is used for the layout, this is a component based library that allows to built responsive (mobile first) websites and web applications. BootstrapVue [105] provides Vue.js components that make use of the Bootstrap layout. This allows fast development and results in a uniform interface. Therefore, BootstrapVue is used for this prototype. Frontend applications also have state, that needs to be administrated. The Veux [106] state management library is used to store all data (state) centralized, with rules ensuring that state can only be manipulated in a way, that always results in the same output. Furthermore, the frontend needs to communicate with the backend, this is realised with RPC using JSON. A JavaScript library named Axios [107] is chosen to realise that functionality.

4.3.2 Backend module

In contrast to the frontend module, the backend consists of many components. These components will be elaborated in the next sections.

Task executor The role of the Task executor is to execute long running tasks asynchronously in the background. In a web application an action starts with sending a request and ends with receiving the response. Accepted lead times are in the range of a few seconds. Some tasks can take up much time, for example a fuzzing task can run for days. Therefore, those tasks have to be executed in a separated thread on the server. Progress information is persisted to the database by long running asynchronous tasks, so that progress can be monitored from the web application, by accessing progress information from the frontend.

The task executor is implemented with Spring's `@Scheduled` and `@Async` annotations. A service class using a method annotated with `@Scheduled` allows repeated execution at a given interval. The prototype checks every 10 seconds in the database if there are jobs to be run using cron like notation `@Scheduled(cron = "*/10 * * * * * *")`. Cron is a time based job scheduler on Unix typed operation systems. The syntax supports specifying values for the following attributes: second, minute, hour, day of the week, day of the month, month, and year. A star value means that a job is executed for every allowable value of that attribute. Also, the division symbol can be used to run each time that division yields a residue of zero. Furthermore, a TaskExecution interface is used, which is implemented by every executable task. A contract is drafted for new tasks by using this interface. A Java class named TaskExecutionFactory is used to create running tasks from the saved tasks from the database. Also, a TaskExecutor class using the `@Async` annotation makes it possible to run jobs in their own Java threads.

RMD extractor The task of the RMD extractor component is twofold. Firstly, the OAS schema files are read and its content is converted to usable objects for the prototype and those objects are persisted in the database. Secondly, relations between web services are derived from the extracted OAS data. The RMD information is saved in the prototype database to the table names starting with the rmd_ prefix. The entity relationship diagram (ERD), an overview of the database structure, is presented in Figure 20 in Appendix B: Architecture diagram. The diagram is created with DBeaver [108], a universal database manager.

There are different versions of the OAS at the moment of writing, namely versions v2 and v3. For the actual parsing of the OAS schema file a third party open source library is used, namely: Swagger Parser [109]. This library only supports v2 and is used because the selected SUT implements version v2 of the OAS schema. In the OAS parser a wrapper is developed that includes this library, therefore the communication takes place via wrapper. Accordingly, it will be easy to extend the wrapper to also support v3.

Fuzzer The fuzzer component is responsible for the creation of test cases (HTTP requests). These test cases are executed by sending the HTTP requests to the SUT, and capture the results (HTTP responses). Functionalities of the fuzzer are divided into smaller logic units of functionality. There is a Java package containing Java classes representing all types of available fuzzers: FuzzerBasic, FuzzerModelBased, FuzzerDictionary, and FuzzerModelBasedDictionary. Furthermore, there is an interface Fuzzer.java, which describes the interface these classes have to implement. All fuzzers extend FuzzerBase which contains functionality that is common for all types of fuzzers and related data is persisted in the database tables with the prefix fuz_, for details see Figure 20 in Appendix B: Architecture diagram. Additionally, there are utility classes that all provide a small piece of functionality, namely: MetaDataUtil, RequestUtil, ParameterUtil, SequenceUtil, and ExecutorUtil. In the next paragraphs these types of fuzzers and units of functionality are further explained.

FuzzerBasic The basic fuzzer is a fuzzer, capable of fuzzing without using the dependency information from the prototype. It uses some information from the extracted OAS, like which actions are available and what parameters these actions have.

FuzzerModelBased This model-based fuzzer can execute model-based fuzzing, by using the dependency information from the OAS and the manual added dependencies. This extracted model enhances the fuzzing process in comparison to the FuzzerBasic which does not use this information.

FuzzerDictionary Dictionary fuzzing can be executed with this type of fuzzer. This form of fuzzing uses items from a dictionary that are likely to trigger some form of vulnerability, e.g. SQL or JSON injection. The dictionaries entered in the prototype are used.

FuzzerModelBasedDictionary The FuzzerModelBasedDictionary is capable executing of model-based dictionary fuzzing. Dependency information from the OAS and manual added information is used, to create valid requests. For every action, parameter, and dictionary item a request can be sent (depending on the configuration), in which every time a parameter is filled with a value from the dictionary.

MetaDataUtil The responsibility of this utility class is to validate and convert the metadata supplied with the fuzzing task. The metadata is persisted in the database formatted in JSON. This JSON data has to be converted before it can be processed by the prototype. Various types of fuzzers require different metadata to work with. The metadata required for all fuzzers is described in a configuration block. All fuzzer type specific data is declared outside the configuration block. A sample configuration for a model-based type fuzzer is presented in Listing 3.

```
{
  configuration: {
    authentication: {
      method: "BASIC", username: "username", password: "password"
    },
    includeActions: [
      {
        path: ".*", httpMethod: ".*"
      }
    ],
    excludeActions: [],
    excludeParameters: [
      {
        action: {
          path: ".*", httpMethod: ".*"
        },
        parameter: {
          name: "meta"
        }
      },
      {
        action: {
          path: ".*", httpMethod: ".*"
        },
        parameter: {
          name: "template"
        }
      }
    ],
    defaults: []
  },
  maxSequenceLength: 3,
  maxNumRequests: 2500
}
```

Listing 3: An example configuration describing how a model-based fuzzer should behave.

The `configuration` block consists of four different types of information. Firstly, the `authentication` block contains information on how to authenticate to the REST web service. The prototype supports basic authentication, which is configured by specifying `method: "BASIC"`. Furthermore, a username and password are required to execute basic authentication. An empty block `{}` can be used to specify that no authentication should be used. Secondly, `includeActions` is specified. This array of actions is included for this fuzzing project. A `path` and `httpMethod` are required to provide, because those are the properties that define an action. Both parameters are interpreted as regular expressions, so in this case (`.*`) all actions known for this SUT are included. Thirdly, actions that should be excluded from the fuzzing project are specified by using `excludeActions`. This should also be an array as used with `includeActions`. Fourthly, exclusion of parameters could be configured by using `excludeParameters`. This should be an array of parameters. A parameter is identified by name and the action it belongs to. All these values are regular expressions as well. Therefore, specific parameters for specific actions can be excluded or specific parameters for all actions. Also, default values for parameters can be assigned, in the `defaults` block.

Besides the general configuration which applies to all types of fuzzers, there is metadata specific for only one type of fuzzer. In the example supplied above, a configuration for a model-based fuzzer is supplied. A model-based fuzzer is instructed to create sequences with a maximum length, in this example the length of three is configured by specifying `maxSequenceLength: 3`. Since, the amount of requests grows enormously with an increasing sequence length, also the number of requests for that test can be limited. For example to limit the number of requests to 2500 for that test, the configuration should include `maxNumRequests: 2500`.

RequestUtil Requests are generated by the RequestUtil from actions, which are extracted from the OpenAPI description. These actions contain a path (URI), an HTTP method, and a collection of parameters. The RequestUtil class creates a request database record from an action. The population of parameters is delegated to the ParameterUtil.

ParameterUtil The ParameterUtil uses information from the RMD to generate valid values for parameters in the request, which eventually are sent to the SUT. Parameters in the RMD contain information, which is used to generate valid values for these parameters. Next, four examples of how RMD information is used to construct valid parameter values are given. Firstly, four different types are identified: `ARRAY`, `BOOLEAN`, `STRING`, and `INTEGER`. Secondly, four different formats are identified for the type `STRING`, namely: `ip`, `uri`, `email`, and `datetime`. Thirdly, for the type `INTEGER` a minimum and maximum value are specified. Fourthly, an `ARRAY` can be defined as an `ENUM`, which contains predefined values. A value is selected randomly from these options.

Furthermore, when model-based fuzzing is executed, the ParameterUtil uses dependency information between parameters and actions to enhance the number of valid requests. So, when a sequence of requests is executed, values for parameters are retrieved from requests executed earlier in that specific sequence. Also, when dictionary fuzzing is executed, values for specific parameters are filled with values from a dictionary.

SequenceUtil The SequenceUtil is a Java class, only used with the model-based typed fuzzers. A sequence consists of a range of one or more generated requests. The RMD contains information on actions (e.g. what parameters they have and what types these parameters are). The SequenceUtil is responsible for generating valid sequences with a maximum given length.

The SequenceUtil starts with generating all possible sequences for the actions used for that specific fuzzing project. The generated sequence is verified by checking if the dependencies are met for every request in that sequence. That means that if a request is dependent on one or more other actions, that action or actions have to be executed in the same sequence before that specific action. The sequence is only added to the collection of valid generated sequences if these conditions are met.

Furthermore, a maximum number of requests is supplied to keep the time to execute the generated requests in manageable proportions. This is demonstrated by the next formula that expresses the number of possible requests (r) in relation to the number of actions (a) of a SUT and the maximum sequence length (sMax): $\sum_{s=1}^{sMax} a^s \cdot s = r$. For example, given a SUT with 25 actions (*a*=25) and a maximum sequence length of three (*sMax* = 3) results in executing $\sum_{s=1}^{3} 25^s \cdot 3 = 48,150$ requests. While this amount of requests is still executable in reasonable time, increasing the maximum sequence length to five (sMax = 5) results in $\sum_{s=1}^{5} 25^s \cdot s = 50,438,775$ requests. In these examples multiple dependencies between actions will be present, therefore some sequences will be invalid and therefore are excluded. But this does not influence the order of magnitude, and the exponential grow of the number of requests with an increasing maximum sequence length.

ExecutorUtil The ExecutorUtil is a wrapper around the open source library Apache HttpClient [110]. This library takes care of all connection related functionality. For example, pooling mechanisms are available to keep a connection open, so the same connection can be used for sending requests and receiving responses without the need to establish a new connection every time a request is sent. The responsibility of ExecutorUtil and some related helper classes are to send requests to the SUT and capture the response and delegate persisting to the database. A helper class (ExecutorUtilHelper) maps domain objects (FuzRequest) to objects that can be handled by the Apache HTTPClient (HttpUriRequest).

Reporter The reporter tool is developed to easily create reports. Two different types of reports can be generated, namely: code coverage and cumulative responses count grouped by HTTP status code. The input for the code coverage report are data dumps containing information about which lines of code are executed related to the PHP classes of the SUT. Information from the prototype's database is needed to generate a report for the cumulative responses count. More in-depth detail on how this data is gathered and aggregated is presented in section 4.1.5.

This reporter tool is incorporated in the prototype for the re-use of existing data access functionalities. Furthermore, the process to generate the code coverage report is a long running task and therefore the task executor of the prototype can be used very effectively. The generated reports are presented as line graphs in this thesis. Therefore, the output of these reporters are LaTeX snippets, which are inserted in the LaTeX file of this thesis. The Tikz and PGFPlots packages are used to produce these line charts. Examples of these reports, e.g. Figure 7 and Figure 10, can be found in section 4.4.

4.4 Results

In this section the results of the second part of this research, experimental prototyping are elaborated. The results for answering research questions 2, 3, and 4 are presented respectively.

4.4.1 Model-based (behavioural) fuzzing

The second research question is addressed in this section: How can a model-based behavioural fuzzer be developed that is capable of detecting vulnerabilities in REST web services? This research question is answered by addressing three aspects regarding the capability of the developed model-based behavioural fuzzer: valid requests, speed, and the ability to detect vulnerabilities. This section ends with a summary of the main findings. A description of the architecture and its components is presented in section 4.2 and 4.3.

Valid requests Regarding the aspect valid requests the results of four experiments are presented. The experiments with the basic fuzzer without model-based capabilities (basic and optimised configuration) were conducted to establish a baseline to compare the model-based fuzzer (basic and optimised configuration) with. The listings of the configurations are presented respectively in Listing 4 and Listing 5 in Appendix D: Information related to the experiments. The number of requests for these experiments were limited to 5,000, see section 4.1.3 for more information.

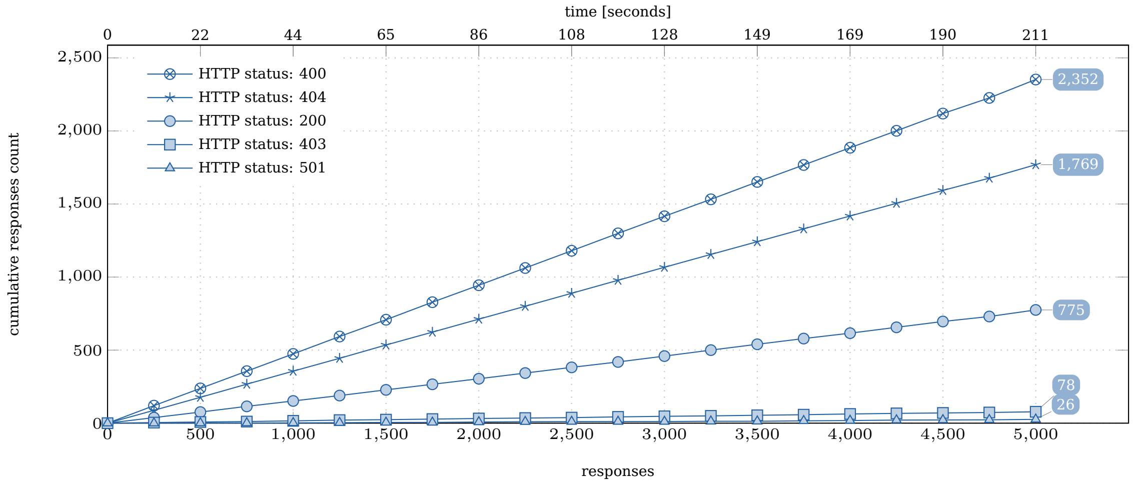


Figure 7: Overview cumulative response codes using the basic fuzzer on WordPress configured with basic configuration.

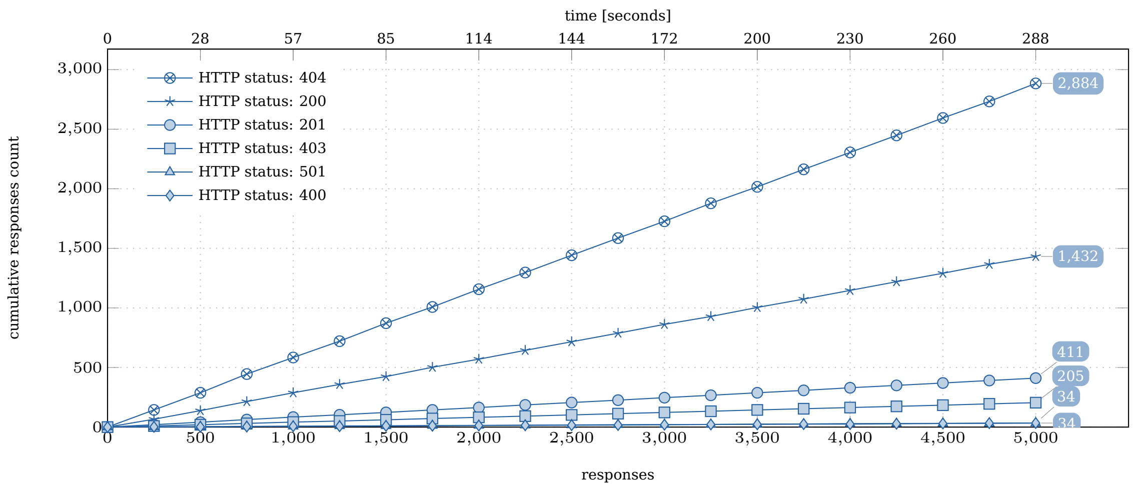


Figure 8: Overview cumulative response codes using the basic fuzzer on WordPress configured with optimised configuration.

The results of the first experiment using the basic fuzzer with basic configuration showed that only 15.50% of the requests were valid (HTTP status codes from responses in the 200 range). This percentage is the ratio of responses with status code HTTP 200 (n=775) to the total number of requests (n=5,000), see Figure 7. Furthermore, the second experiment using the optimised configuration resulted in a higher percentage of valid requests, namely 36.86% returning HTTP response codes in the 200 range. This percentage is the ratio of responses with status code HTTP 200 (n=1,432) and HTTP 201 (n=411) to the total number of requests (n=5,000), see Figure 8.

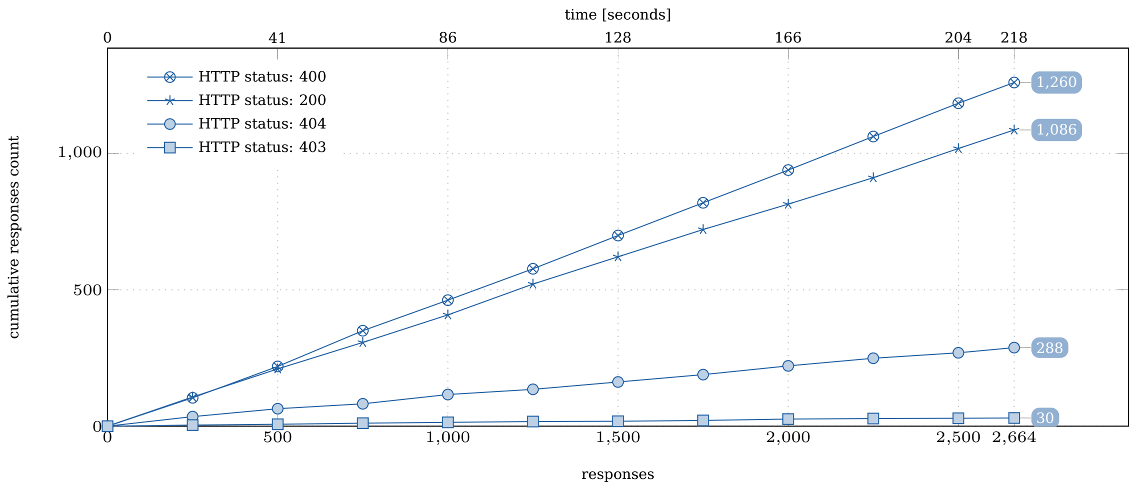


Figure 9: Overview cumulative response codes using the model-based fuzzer on WordPress configured with basic configuration.

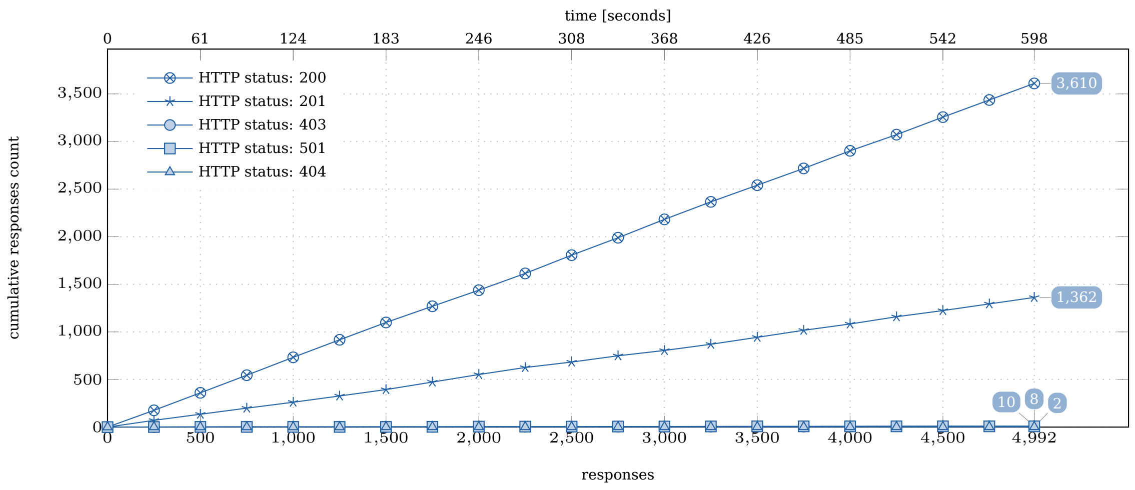


Figure 10: Overview cumulative response codes using the model-based fuzzer on WordPress configured with optimised configuration.

The results of the third experiment using the model-based fuzzer with the same basic configuration showed that 40.77% of the requests were valid (HTTP status codes from responses in the 200 range). This percentage is the ratio of responses with status code HTTP 200 (n=1,086) to the total number of requests (n=2,664), see Figure 9. Furthermore, the fourth experiment using the same optimised configuration resulted in a higher percentage of valid requests, namely 99.60% returning HTTP response codes in the 200 range. This percentage is the ratio of responses with status code HTTP 200 (n=3,610) and HTTP 201 (n=1,362) to the total number of requests (n=4,992), see Figure 10. In these two experiments not all of the 5,000 pre-calculated requests were executed, because sequences were aborted if one of the requests in that sequence did not return a response with an HTTP status code in the 200 range. So, from these four experiments can be concluded that the number of valid requests was significantly higher for the model-based fuzzer than the basic fuzzer. Furthermore, using the optimised configuration resulted in higher percentages of valid requests than the basic configuration.

Speed In these four experiments, regarding the aspect speed can be observed that the basic fuzzer with basic configuration executed 23.7 requests per second and with the optimised configuration 17.4 requests per second. Furthermore, the model-based fuzzer with the basic configuration executed 12.2 requests per second and with the optimised configuration 8.3 requests per second. So, from the data can be concluded that the basic fuzzer is about twice as fast as the model-based fuzzer. Also, the configuration had influence on the speed of the fuzzer, using the basic configuration resulted in faster execution.

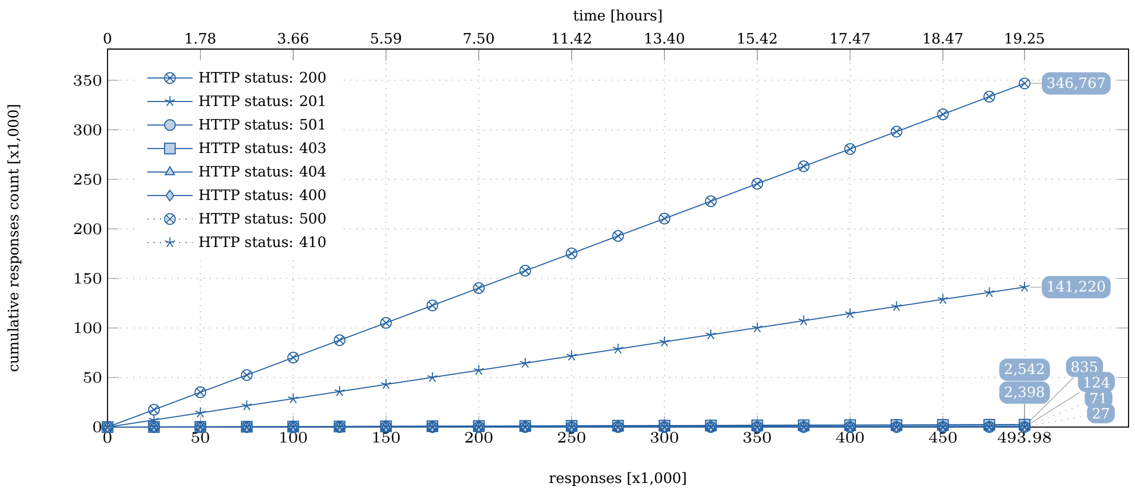


Figure 11: Overview cumulative response codes using the model-based fuzzer on WordPress configured with optimised configuration (maximum number of 500,000 requests).

Detect vulnerabilities A fifth experiment was executed with a long running model-based fuzzing task to detect vulnerabilities in WordPress. For this experiment the maximum number of requests was set to 500,000 requests. The optimised configuration was used and the maximum sequence length was set to 5. As a result, 493,984 requests were sent to WordPress in nineteen hours, fifteen minutes and fourteen seconds, see Figure 11. Furthermore, all responses that contained an HTTP status not in the 200 range were examined. A database query was composed to group all different response codes and bodies in the ranges not equal to 200, see Figure 27 in Appendix D: Information related to the experiments. All unique results (n=43) were manually inspected. No indication for revealing a vulnerability was found, all these responses were traceable to valid situations. For example, four different types of errors in the HTTP 500 range were detected: 501 (terms can't be placed in the recycle bin), 501 (users can't be placed in the recycle bin), 500 (username already exists), and 500 (this slug is already in use by another term).

In sum The model-based behavioural fuzzer is capable of sending a high percentage of valid requests, the speed of the developed prototype is sufficient, and no vulnerabilities in the REST web services of WordPress were detected. A summary of the results in this section is provided in Table 7.

Table 7: Summary of the results of the experiments with the basic and model-based behavioural fuzzer.

type of fuzzer	configuration	meta data	valid requests [%]	speed [requests per second]	vulnerabilities detected
basic	basic	5,000 requests WordPress	15.50	23.7	-
basic	optimised	5,000 requests WordPress	36.86	17.4	-
model-based	basic	5,000 requests WordPress	40.77	12.2	-
model-based	optimised	5,000 requests WordPress	99.60	8.3	-
model-based	optimised	500,000 requests WordPress	-	-	no

4.4.2 Model-based (behavioural) dictionary fuzzing

The third research question is addressed in this section: How can a model-based behavioural dictionary fuzzer be developed that is capable of detecting SQL injection vulnerabilities in REST web services? This research question is answered by addressing three aspects regarding the capability of the extended prototype, a model-based behavioural dictionary fuzzer: valid requests, speed, and the ability to detect SQL injection vulnerabilities.

Valid requests Regarding the aspect valid requests the results of four experiments are presented. The experiments with the dictionary fuzzer without model-based capabilities (basic and optimised configuration) were conducted to establish a baseline to compare the model-based dictionary fuzzer (basic and optimised configuration) with. The basic and optimised configuration were identical to the ones used for the basic and model-based fuzzer, see section 4.4.1. The projects were configured to use all items from the dictionary for all parameters for all the actions from the SUT.

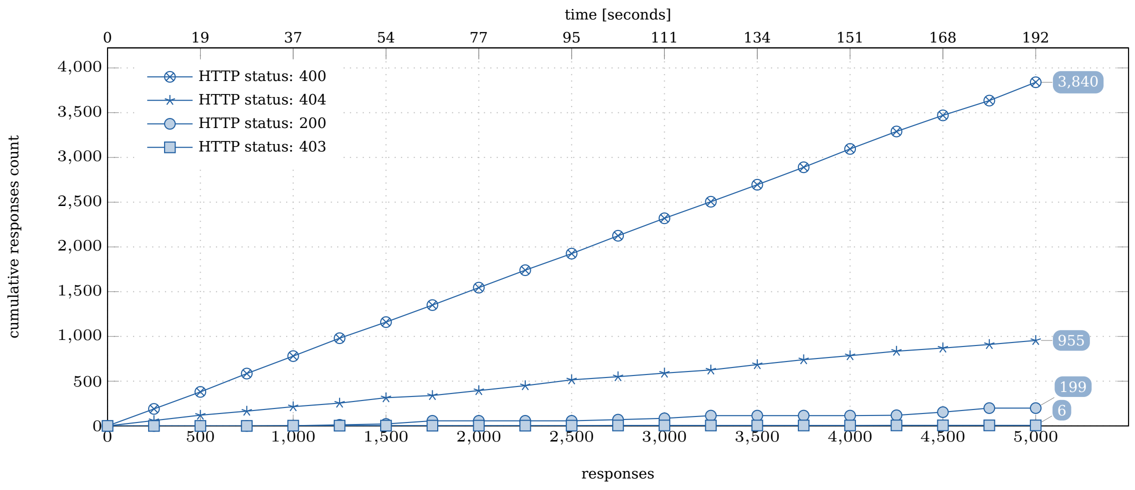


Figure 12: Overview cumulative response codes using the dictionary fuzzer on WordPress configured with basic configuration.

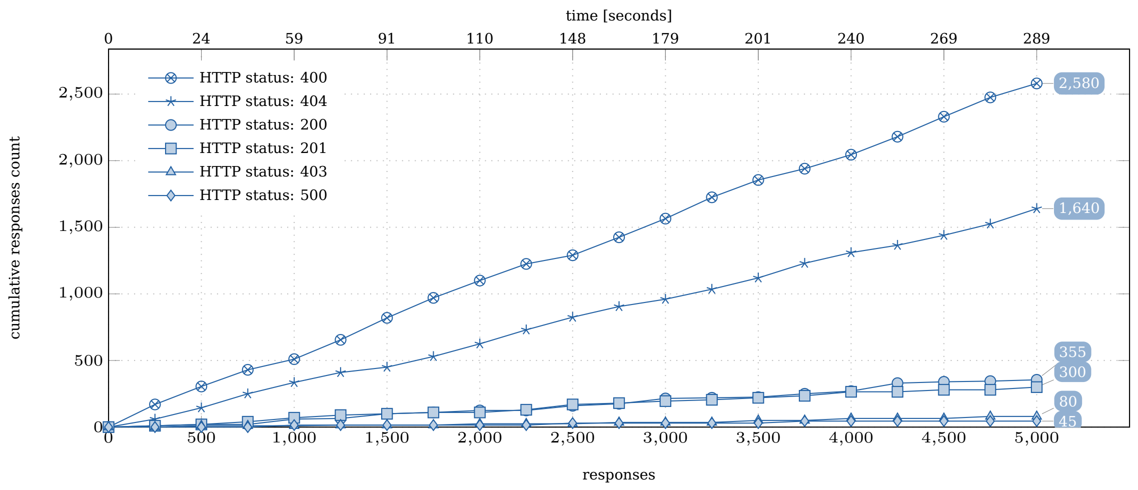


Figure 13: Overview cumulative response codes using the dictionary fuzzer on WordPress configured with optimised configuration.

The results of the first experiment using the dictionary fuzzer with basic configuration showed that only 3.98% of the requests were valid (HTTP status codes from responses in the 200 range). This percentage is the ratio of responses with status code HTTP 200 (n=199) to the total number of requests (n=5,000), see Figure 12. Furthermore, the second experiment using the optimised configuration resulted in a higher percentage of valid requests, namely 13.10% returning HTTP response codes in the 200 range. This percentage is the ratio of responses with status code HTTP 200 (n=355) and HTTP 201 (n=300) to the total number of requests (n=5,000), see Figure 13.

The results of the third experiment using the model-based dictionary fuzzer with the same basic configuration showed that 4.76% of the requests were valid (HTTP status codes from responses in the 200 range). This percentage is the ratio of responses with status code HTTP 200 (n=206) to the total number of requests (n=4,330), see Figure 14. In this experiment not all of the 5,000 requests were executed due to aborted sequences receiving erroneous status codes in the responses. Furthermore, the fourth experiment using the same optimised configuration resulted in a higher percentage of valid requests, namely 66.50% returning HTTP response codes in the 200 range. This percentage is the ratio of responses with status code HTTP 200 (n=567) and HTTP 201 (n=2,758) to the total number of requests (n=5,000), see Figure 15. So, from these four experiments can be concluded that the number of valid requests was significantly higher for the model-based dictionary fuzzer than the dictionary fuzzer. Furthermore, using the optimised configuration resulted in higher percentages of valid requests than the basic configuration.

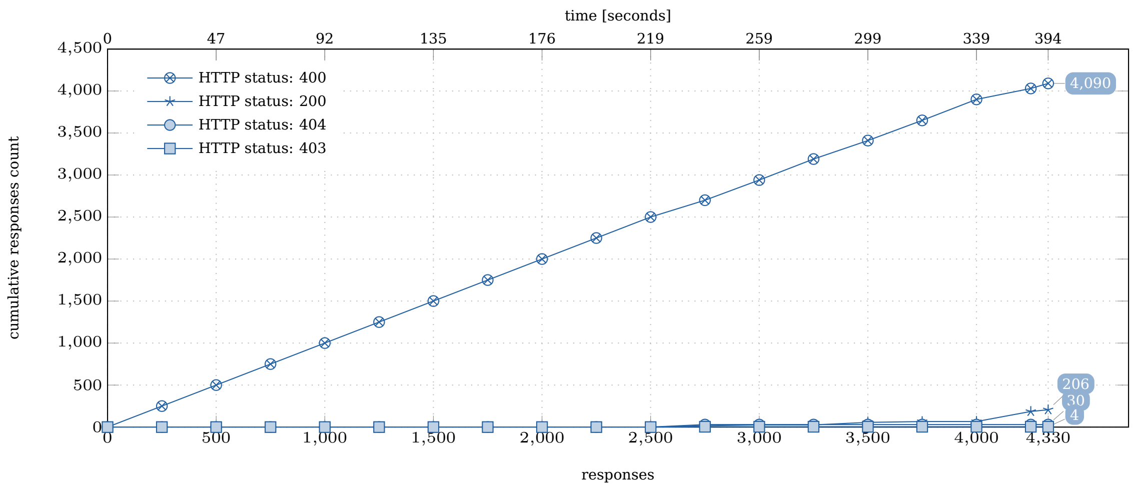


Figure 14: Overview cumulative response codes using model-based dictionary fuzzer on WordPress with basic configuration.

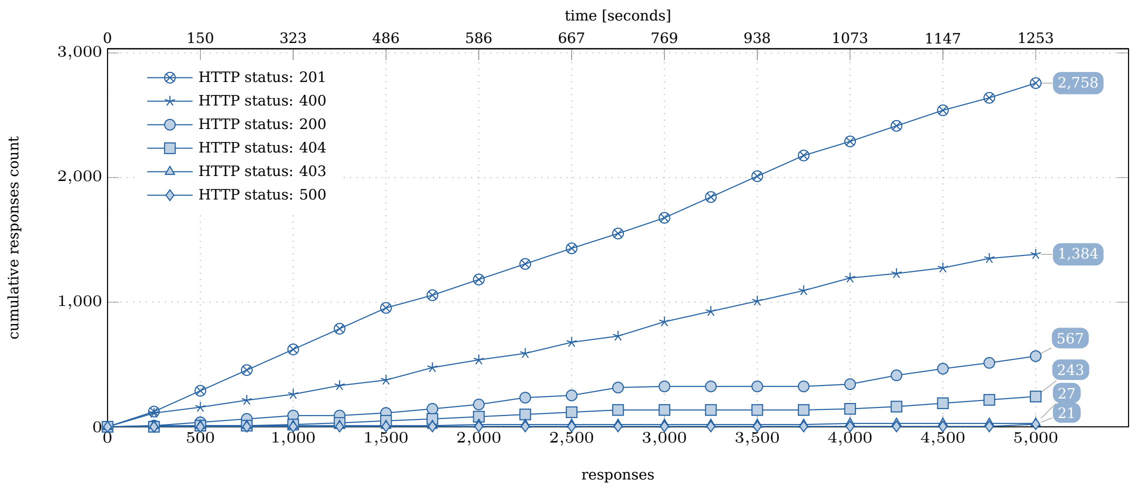


Figure 15: Overview cumulative response codes using model-based dictionary fuzzer on WordPress with optimised configuration.

Speed In these four experiments, regarding the aspect speed can be observed that the dictionary with basic configuration executed 26.0 requests per second and with the optimised configuration 17.3 requests per second. Furthermore, the model-based dictionary fuzzer with the basic configuration executed 11.0 requests per second and with the optimised configuration 4.0 requests per second. So, from the data can be concluded that the dictionary fuzzer is faster than the model-based dictionary fuzzer. Also, the configuration had influence on the speed of the fuzzer, using the basic configuration resulted in faster execution.

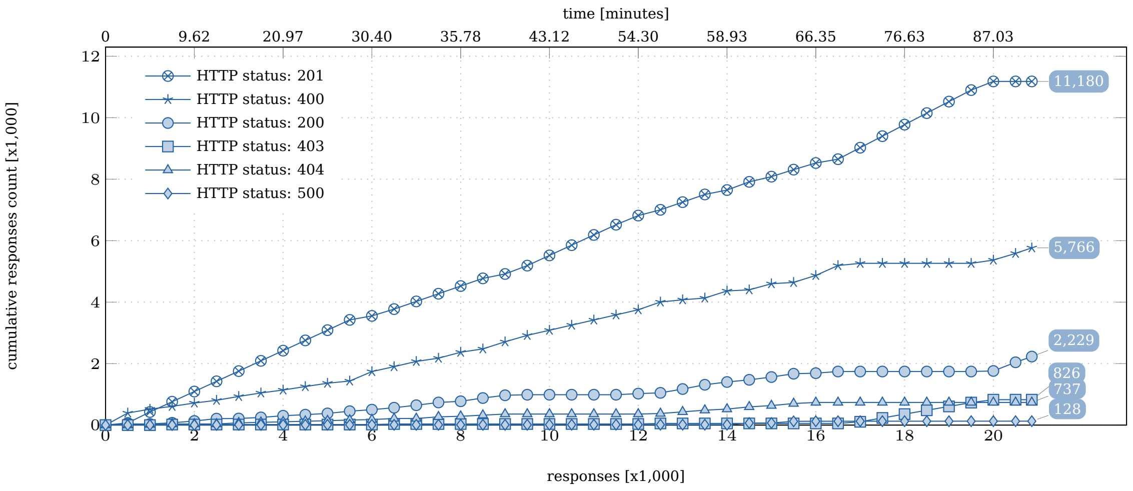
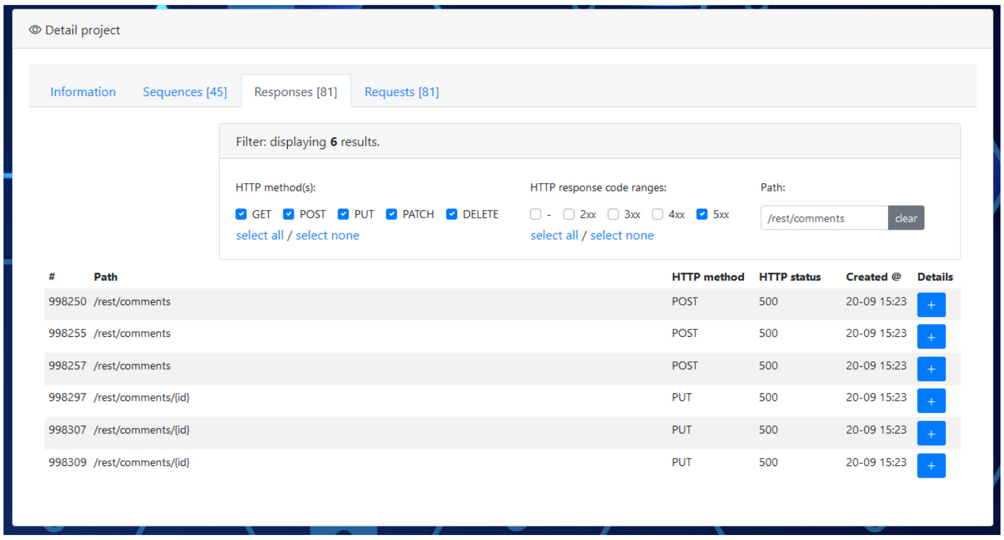


Figure 16: Overview cumulative response codes using the model-based dictionary fuzzer on WordPress configured with optimised configuration (maximum number of 500,000 requests).

Detect SQL injection vulnerabilities A fifth experiment was executed with a long running model-based fuzzing task to detect SQL injection vulnerabilities in WordPress. The maximum number of requests of 5,000 was increased to 500,000. All possible combinations of actions, parameters, and dictionary items resulted in sending 20,866 requests to WordPress, see Figure 16. The number of response codes in the 400 range was high (n=7,329). This is expected due to sending parameters that were often not compatible with expected parameters, e.g. sending string values containing characters that could exploit SQL injection vulnerabilities when numbers were expected. A database query was used to inspect status codes and bodies of the responses that were not in the 200 range (n=78), to check if vulnerabilities were uncovered. The results are presented in Figure 28 in Appendix D: Information related to the experiments. No indications were found that vulnerabilities were detected by the fuzzing process with the model-based dictionary fuzzer.

Verification to detect SQL injection vulnerabilities The sixth experiment was executed to validate the capability of the prototype to detect SQL vulnerabilities, because no SQL vulnerabilities were found in the REST web services of WordPress. An additional SUT, SutSqlI, was used for this purpose. This SUT was registered in the prototype and information from the OAS was extracted and dependencies were determined, for more information see Figures 29, 30, 31, and 32 in Appendix D: Information related to the experiments.


Detail project

Information | Sequences [45] | Responses [81] | Requests [81]

Filter: displaying **6** results.

HTTP method(s): ✓ GET ✓ POST ✓ PUT ✓ PATCH ✓ DELETE — select all / select none

HTTP response code ranges: ☐ - ☐ 2xx ☐ 3xx ☐ 4xx ✓ 5xx — select all / select none

Path: /rest/comments — clear

#	Path	HTTP method	HTTP status	Created @	Details
998250	/rest/comments	POST	500	20-09 15:23	+
998255	/rest/comments	POST	500	20-09 15:23	+
998257	/rest/comments	POST	500	20-09 15:23	+
998297	/rest/comments/{id}	PUT	500	20-09 15:23	+
998307	/rest/comments/{id}	PUT	500	20-09 15:23	+
998309	/rest/comments/{id}	PUT	500	20-09 15:23	+

Figure 17: All responses of the SutSqlI experiment in the HTTP 500 range.

As a result of this experiment 81 requests were executed. The HTTP response code of six of the responses to those requests were in the HTTP 500 range, see Figure 17. These specific response bodies were examined. The observations are presented here. Firstly, three responses belonged to the REST action `POST /rest/comments` and three belonged to `PUT /rest/comments/{id}`. These actions correspond to the create and update functionality, which are vulnerable to SQL injection. The other actions (read and update) did not return a response in the HTTP 500 range. Secondly, when examining these bodies three of the nine values from the MySQL detection dictionary from FuzzDB triggered a SQL exception: `1’ and 1=(select count(*) from tablenames); --`, `1’1`, and `fake@ema’or’il.nl’=’il.nl`. Thirdly, applying values from the dictionary to the id parameter resulted in an HTTP 400 status code (Failed to convert value of type ’java.lang.String’ to required type ’java.lang.Long’; nested exception is java.lang.NumberFormatException: For input string: <value from dictionary>). This can be explained by the fact the code is converting the string into a number. Therefore, the id parameter is not susceptible to SQL injection. Fourthly, six of the nine values from the dictionary did not trigger an exception. In these cases the result was a valid response in HTTP 200 range.

Noteworthy is the speed achieved by the prototype in the experiment on SutSqlI. The prototype was able to send requests at a faster pace, namely at 40,0 requests per second. Therefore, the prototype is not the bottleneck in the experiments with WordPress in terms of sending the requests and processing the responses. The present vulnerabilities in both REST actions were detected by the prototype. In addition, the method of manual analysis by the researcher of the HTTP 500 responses was successful in determining that a vulnerability was detected.

In sum The model-based dictionary fuzzer is capable of sending a high percentage of valid requests and the speed of the developed prototype is sufficient. Furthermore, no vulnerabilities in the REST web services of WordPress were detected, but all the SQL injection vulnerabilities in SutSqlI were detected. A summary of these results is provided in Table 8.

Table 8: Summary of the results of the experiments for the dictionary and model-based dictionary fuzzer.

type	configuration	meta data	valid requests [%]	speed [requests per second]	vulnerabilities detected
dictionary	basic	5,000 requests WordPress	3.98	26.0	-
dictionary	optimised	5,000 requests WordPress	13.10	17.3	-
model-based dictionary	basic	5,000 requests WordPress	4.76	11.0	-
model-based dictionary	optimised	5,000 requests WordPress	66.50	4.0	-
model-based dictionary	optimised	500,000 requests WordPress	-	-	no
model-based dictionary	optimised	500,000 requests SutSqlI	-	-	yes

4.4.3 The effectiveness of model-based (behavioural) dictionary fuzzing

The last research question is addressed in this section: How effective is a model-based behavioural dictionary fuzzer in detecting SQL injection vulnerabilities in REST web services? This research question is answered by addressing one aspect regarding the effectiveness of the developed model-based dictionary fuzzer: code coverage.

Code coverage The results of the first experiment using the model-based fuzzer showed that after 5,000 requests were executed, a total of 6,819 lines of codes were at least executed once. This corresponds to 49.74% of all lines of code loaded for serving the requests. The results of this experiment are presented in Figure 18. The top line presents the executed lines of code of all the loaded PHP classes needed for serving these requests. The bottom line presents execution of a specific part of the application which includes the classes in the folder `\rest-api\`. These classes can be indicated as the attack surface of the REST web services of WordPress.

The results of the second experiment using the model-based dictionary fuzzer showed that after 5,000 requests were executed, a total of 6,234 lines of codes were at least executed once. This corresponds with 45.36% of all lines of code loaded for serving the requests. The greatest increase can be noticed at the beginning of the experiment. After sending 250 requests, already 5,005 lines of code (38.36%) were executed. So, most of the code is already executed after sending a small amount of requests. These results are presented in Figure 19.

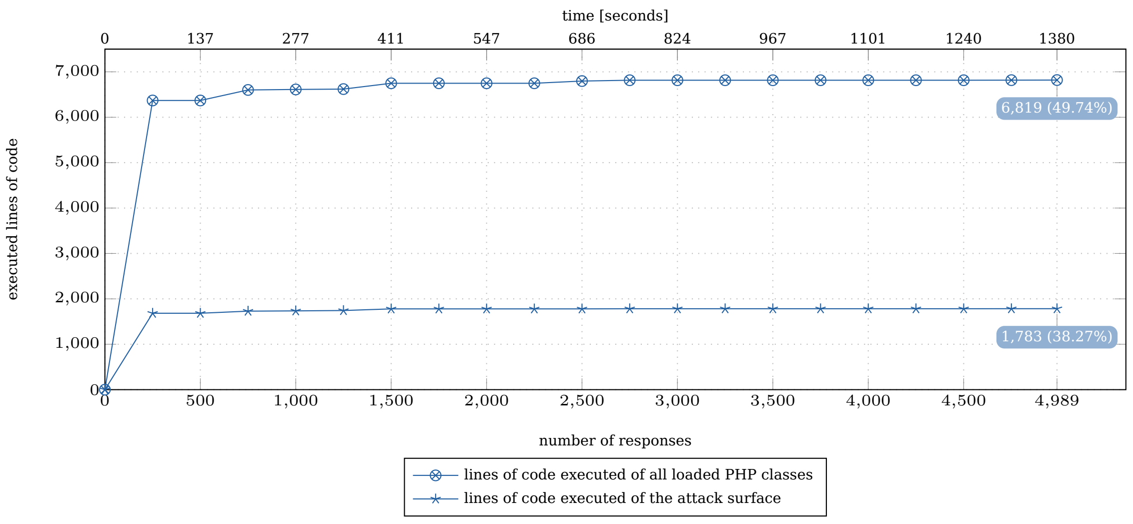


Figure 18: Overview code coverage executing the model-based fuzzer on WordPress.

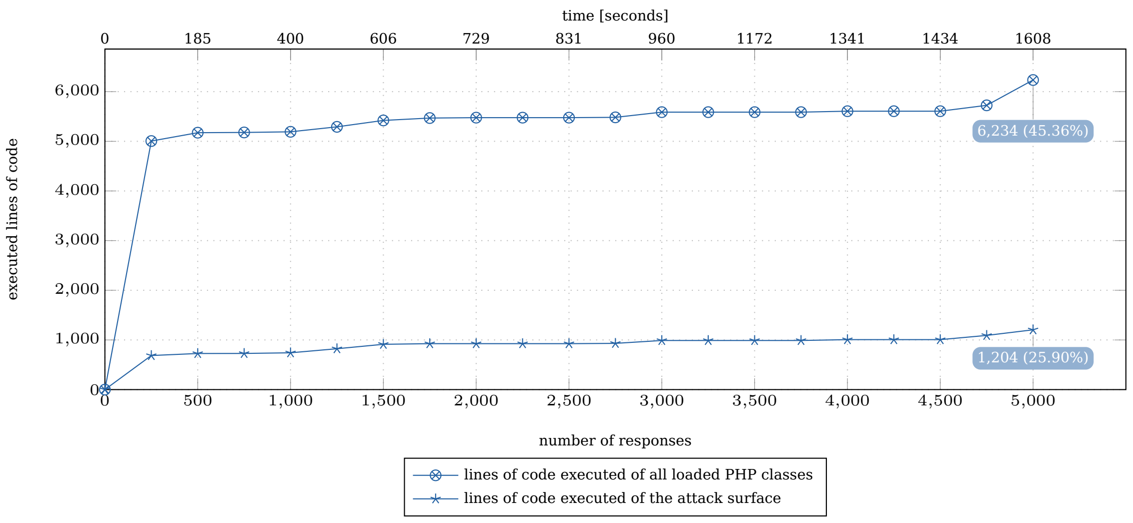


Figure 19: Overview code coverage executing the model-based dictionary fuzzer on WordPress.

Some observations can be made, in comparison with the experiment of the model-based fuzzer. Firstly, for both type of fuzzers, the increase of code coverage was reached for the greatest part at the beginning of the experiment. For the model-based experiment the amount of code coverage almost becomes constant. In contrast, the model-based dictionary fuzzer achieved some increase of code coverage over time. Secondly, the speed of both types of fuzzer decreased. The speed of the model-based and model-based dictionary fuzzer were decreased to 3,6 and 3,1 requests per second respectively. This is about 50% of the speed in comparison to the experiments in which code coverage data was not captured.

In sum These experiments have shown the prototype was quite effective in executing its tasks by achieving high percentages of code coverage. A summary of these results is provided in Table 9

Table 9: Summary of the results of the experiments for measuring code coverage of the model-based and model-based behavioural dictionary fuzzer.

type	configuration	meta data	code coverage total [%]	code coverage attack surface [%]
model-based dictionary	optimised	5,000 requests WordPress	45.36	25.90
model-based	optimised	5,000 requests WordPress	49.74	38.27

5 Discussion, conclusion and recommendations

This research focuses on the types of vulnerabilities that can be found in REST web services and how a model-based behavioural fuzzer that is capable of detecting these vulnerabilities can be developed. This is achieved by answering four research questions:

- RQ1: What types of vulnerabilities can be detected in REST web services?
- RQ2: How can a model-based behavioural fuzzer be developed that is capable of detecting vulnerabilities in REST web services?
- RQ3: How can a model-based behavioural dictionary fuzzer be developed that is capable of detecting SQL injection vulnerabilities in REST web services?
- RQ4: How effective is a model-based behavioural dictionary fuzzer in detecting SQL injection vulnerabilities in REST web services?

This section provides an overview and discussion of the main findings. The strengths and weaknesses of this study are described in relation to other studies. Furthermore, conclusions are drawn and implications are given for immediate practice and further research.

5.1 Vulnerability types in REST web services

In the first part of this study, a systematic literature review was conducted. The following three conclusions can be drawn.

Firstly, 29 types of vulnerabilities in web services were identified based on literature and 49 types of vulnerabilities were found in the NVD (focus solely on REST). The most commonly found vulnerability types in this study (SQL injection, XML injection, XPath injection, cross-site scripting, and authentication and access control related vulnerabilities) are also presented by studies executed by Iqbal et al. [62] and Mouli and Jevitha [74]. It is interesting to note that in this study more types of vulnerabilities were found in comparison to those other studies. This might be caused by the usage of more specific, search terms [62], a difference in focus (e.g. cloud computing [74], instead of, in this study REST web services), or a difference in the type of sources (e.g. only literature [62, 74], instead of in this study, also the NVD).

The merit of this study is that the focus is on REST web services. Most of the sources found in this literature review do not focus on REST, but on SOAP related web services. These results are likely to be generalisable to REST, because the results were verified with additional validity evidence of the NVD. Only REST related vulnerabilities were selected from the NVD. In other words, the same types of vulnerabilities were found in literature and in the NVD.

Furthermore, a difference in the amount of types of vulnerabilities between the NVD and literature is found. A possible explanation is that the vulnerability types (CWEs) in the NVD sometimes are very specific and somewhat overlapping, e.g.: CWE-284 Access Control (Authorization) Issues and CWE-863 Incorrect Authorization, while in literature these types are more likely to be grouped together. Also, some CWEs need to be reclassified, because they are obsolete (i.e. NVD-CWE-other or CWE-17 Deprecated).

Secondly, the results of this study have shown that the number of REST web services vulnerabilities in the NVD increases over time, both in numbers and as a percentage of the total number of registered vulnerabilities (see Table 14 in Appendix A: Results literature review. This finding is substantiated by the significantly increasing use of REST web services. As a result, the number of vulnerabilities is also growing, due to the complexity, connectivity, and extensibility of these web services [7]. In line with this development, it is worth mentioning, that this study also revealed that most studies focus solely on SOAP web services.

Thirdly, the vulnerabilities are categorised according to the OWASP Top 10. The four most important categories are: injection, broken authentication, broken access control, and cross-site scripting. The order of these four categories confirm the most important OWASP Top 10 categories of vulnerability types in web services [52]. This is not surprising, because the OWASP Top 10 is, comparable to this literature review, also based on the occurrence of vulnerabilities, aside the extent to which the vulnerabilities are exploitable, detectable, and the impact they have [52]. Although, the top four ranked OWASP Top 10 categories found in the NVD are conform the four categories found in literature, they are in a different order. This might be explained by not taking into account the extent to which these vulnerability types are exploitable, detectable and how much impact they have.

In conclusion, this research presented an overview of the (REST) web service vulnerability types categorised conform the OWASP Top 10. A practical implication of these findings is that it can help organisations to increase the general security awareness among software architects, developers, and managers in organisations using (REST) web services. This can be achieved by using this thesis as input for conversations or education about these types of vulnerabilities, the impact of those vulnerabilities for practice, and the precautionary measures the team should undertake. This awareness and ensuing actions will contribute to increase the quality of software. By reducing the amount of vulnerabilities that are present in software systems serious damage can be avoided (e.g. losing privacy-sensitive information or reputational damage for companies). This research makes these theoretical findings more accessible for further use in practice, because a well known and adopted categorisation system, the OWASP Top 10 [52] was used.

Moreover, this research highlighted that the number of REST web services vulnerabilities in the NVD increases over time. It also showed that most articles in literature focus solely on SOAP web services. This study also showed that the types of vulnerabilities in web services in general are generalisable to REST. Taken together, these findings implicate that further research should focus on methods for detection and prevention of vulnerabilities in REST web services and whether existing methods for other types of web services (i.e. SOAP) are to some level applicable to REST. In other words, this research emphasises the importance to put REST vulnerabilities more prominent on the software security research agenda. To continue this first part of research, the next part will focus on detecting vulnerabilities in REST web services and will mainly concentrate on the most commonly found type of vulnerabilities in literature: SQL injection vulnerabilities.

5.2 Developing a model-based (behavioural) dictionary fuzzer

In the second part of this study a prototype capable of executing different types of fuzzing was developed by applying experimental prototyping. Different types of fuzzing were implemented iteratively, namely: basic fuzzing, dictionary fuzzing, model-based fuzzing, and model-based dictionary fuzzing. Also, various experiments were executed with these different types of fuzzers on WordPress and a self-developed SUT named SutSqlI. These experiments demonstrated the capability and effectiveness of the developed prototype. The following six conclusions can be drawn.

Firstly, results showed that the developed model-based fuzzer was successful in sending a high percentage of valid requests. A request is considered valid when the response of the server contains an HTTP status code in the 200 range [39], which applies to all HTTP related traffic, so also to REST over HTTP. With model-based fuzzing the sequence of requests is valid when all requests in that sequence are valid [15]. The model-based fuzzer in this study was capable of sending a high percentage of 99.60% valid requests. This finding shows that the prototype is successfully in executing model-based fuzzing tasks. These results are explainable because type information and dependency (or relational) information were extracted from the OAS and used to create a behavioural model that allowed to satisfy validations imposed by the SUT. Research of Atlidakis et. al [15] underlined the merit of using dependency information for model-based fuzzing.

Secondly, the percentages valid requests from the model-based typed fuzzers were considerably higher than of the variants that did not use model-based fuzzing, see Table 10. This implies that model-based fuzzing has an advantage over the variants that did not use the model-based information for fuzzing REST web services. Noteworthy is the difference between the model-based variants in achieving generating valid requests. The model-based fuzzer achieves a higher percentage than the model-based dictionary fuzzer. This can be explained, because parameters were deliberately filled with invalid values from the dictionary. This resulted in many responses with HTTP status code 400, indicating input validation errors, which can be seen in figure 28 in Appendix D: Information related to the experiments. Additionally, the model-based capabilities are only required to create requests that are mostly valid (for example, the update of an entity can only occur on an existing entity), except the one parameter that is replaced by a value from the dictionary.

Table 10: Overview valid requests model-based versus non model-based typed fuzzers.

	basic	model-based	dictionary	model-bassed dictionary
basic configuration	15.50%	40.77%	3.98%	4.76%
optimised configuration	36.86%	99.60%	13.10%	66.50%

Thirdly, the prototype detected no vulnerabilities in WordPress. Not finding any vulnerabilities in WordPress can be explained by the following reasons. Recently, not many vulnerabilities of any kind were found in REST web services of WordPress [111]. Furthermore, the vulnerabilities that were found in the REST web services were not in the core functionality, but in plugins [55]. This is caused by the fact that WordPress takes security very seriously, as can be read in a white paper on the topic [112]. For example, WordPress has a quality security team, consisting of fifty software security experts, that monitor the software security quality. Furthermore, the OWASP Top 10 is used to present APIs, resources, and policies that are used to prevent those top ten types of vulnerabilities. Also, the bug bounty program HackerOne is used by WordPress, in which beta versions were subjected to security testing by researchers and software producers (since July 2016). Correspondingly, previous research presented that these programs can be effective in detecting vulnerabilities [111]. However, for practice, this does not imply that fuzzing is not necessary on software systems that have undergone severe security checks in the SDLC, because different means should be applied in various stages of the SDLC [7].

Fourthly, the developed prototype did detect SQL injection vulnerabilities in SutSqlI. This proves that RESTFuzzer is a capable fuzzer. The quality of the methodology of this study is strengthened by using a second source, SutSqlI, as a verification method to support the previous findings. Besides the already provided indirect evidence, like a high percentage of valid requests, there is now also direct proof of the capability of the fuzzer to detect SQL injection vulnerabilities.

Fifthly, the prototype can send requests fast enough to be capable of fuzzing. In the current experiment setup on WordPress, speeds from 3,1 to 23,7 requests per seconds were reached (without registering code coverage data). The current speed of the prototype were sufficient to execute long running tasks (500,000 requests) in reasonable time (approximately nineteen hours). Executing the fuzzing task on SutSqlI, resulted even in sending 40,0 requests per second. Therefore, the limitation in terms of speed was in the case of the experiment WordPress and not the prototype. Comparing results to other fuzzers in terms of speed is difficult, because the SUT also impacts the speed. To make a comparison, the research of Atlidakis et al. [15] was consulted, as the type of fuzzer and SUT is comparable to this research. From the graphs in their paper can be calculated that the speed of their fuzzer ranges from 2,9 to 4,5 requests per second. This is comparable to the speed of RESTFuzzer. A final remark on this topic was that the applied configuration had impact on the speed of the fuzzer, using the basic configuration instead of the optimised resulted in all situations a higher speed.

Sixthly, this research showed the fuzzer was effective in executing its tasks by achieving high percentages of code coverage. The model-based dictionary typed fuzzer executed 45.36% of all lines of code of the loaded PHP class files after sending 5,000 requests. The model-based fuzzer achieved an even higher percentage, namely: 49.74%. In previous research no clear threshold percentage was found for an acceptable or required code coverage. Although, in comparison to previous research of Takanen et al. [39] (e.g. ten different fuzzers achieve code coverage between approximately 20-42% while fuzzing a FTP server software program) the results achieved in this study seem to be solid enough for a fuzzer to be capable of detecting vulnerabilities. Furthermore, code coverage is also an important predictor for the number of vulnerabilities found [39].

Noteworthy is the higher percentage of code coverage for the model-based fuzzer. This can be caused by using the randomised application of sequences. As a result, much of the REST functionality is executed at the beginning of the experiment. In contrast, the model-based dictionary fuzzer iterates over all existing actions, parameters, and dictionary items, in a fixed order. Therefore, there is less spread in executing the functionality. Furthermore, the fast increase of code coverage in the beginning of both experiments is noticeable. This can be explained by the fact that a part of the source code is generic and is used in most of the actions, i.e. that generic code is executed the first time a REST action is processed.

Further research could focus on using other types of dictionaries in model-based dictionary fuzzing REST web services. In this research SQL injection dictionaries were explored. Since, injection is one of the most common type of vulnerabilities in REST web services, further research could focus on using JSON injection dictionaries. These forms of fuzzing might also be applicable to SOAP web services. The SOAP technique is somewhat obsolete, but many software applications still rely on this form of machine-to-machine communication. Therefore, it might be interesting to see if model-based dictionary fuzzing is also able to detect vulnerabilities in SOAP web services. Finally, integrating this or another form of fuzzing in the SDLC could also be an interesting topic for further research.

In conclusion, the product of this part of the research is a prototype (RESTFuzzer) capable of four different types of fuzzing, i.e. basic, dictionary, model-based, and model-based dictionary fuzzing on REST web services. Also, for validation purposes, a SUT (SutSqlI) was developed that intentionally contains SQL injection vulnerabilities. These products, RESTFuzzer and SutSqlI, can be used for further research as well as in practice. In this research the developed prototype was tested in practice, by conducting various experiments on the REST web services of WordPress and SutSqlI. The results showed that fuzzers can be used as a security testing tool, which is a methodological contribution as well as a practical one. Also, due to the modular architecture of the program new DLs can be supported or other versions of the OAS. Furthermore, a theoretical implication is that further research should focus on executing model-based fuzzing on more SUTs to determine if this form of model-based fuzzing can be successful in detecting vulnerabilities. Overall, this thesis contributes, not only to increase the awareness of vulnerabilities in REST web services, but provides also a capable and effective fuzzer (RESTFuzzer) and a SUT (SutSqlI) for practical use. Furthermore, it demonstrates the effectiveness of using a new hybrid form of fuzzing, a combination of model-based and dictionary fuzzing on REST web services. All this results in creating more secure software and eventually decreases the chance of exploitation of vulnerabilities by malicious actors, which can cause serious damage for individuals and companies.

6 Reflection

In this section, the researcher reflects on the process and products. This means that lessons learned while conducting the research are shared.

6.1 Vulnerability types in REST web services

The first lesson learned from this systematic literature review was the importance of executing such a research method systematically. Therefore, the accurate documenting of all steps in the method section is absolutely necessary. Furthermore, a proposal is an adequate guideline to do research, but sometimes adjustments have to be made, along the way, in order to get sufficient answers to one or more of the research questions. As expected, not much could be found in literature on especially REST web services related to vulnerability types. Therefore, in order to answer the first research question, another source, the NVD, was consulted. This turned out to be a good source to verify results found in literature. Regarding the process can be concluded that the planning in terms of hours is pretty accurate, but spending the amount of hours per week, is often not feasible. Therefore, holidays and weekends are necessary to catch up for lost hours, in order to stay on track with the planning.

6.2 Developing a model-based (behavioural) dictionary fuzzer

The first lesson learned in the second part of this study was how to develop different types of fuzzers. The process of developing the software took more time than initially expected and calculated in the planning. Some of the extra time needed was caused by the fact that not all dependencies could be extracted automatically, due to not following the REST conventions completely. Additionally, manual adding of dependencies had to be developed. Also, some parts of the development just were underestimated, since some applied techniques and frameworks were new to me. Like, LaTeX was chosen as tool to create this document. It had great benefits which are really appreciated by me. In contrast, it should be emphasised that the learning curve of LaTeX is rather steep and more time was spent on some simple lay-out issues than calculated. The second lesson learned is regarding the selection of the SUT. The SUT was selected by applying a list of five requirements described in section 4.1.1. Looking back, a sixth requirement should have been added: plausibility of the presence of vulnerabilities. Then a less mature SUT than WordPress was selected in which it was more likely to detect vulnerabilities. Finally, experimental prototyping worked very well for developing a prototype for functionality that was not completely clear at the start of a project. Adjusting the software to newly found obstacles is fast, due to the short cycles of (re)design, implement, test, and analyse.

References

Peer-reviewed

[1] G. Yee, “Removing software vulnerabilities during design,” in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, Jul. 2018. DOI: 10.1109/compsac.2018.10284.

[2] N. Antunes and M. Vieira, “Designing vulnerability testing tools for web services: Approach, components, and tools,” *International Journal of Information Security*, vol. 16, no. 4, pp. 435–457, Aug. 2017, ISSN: 1615-5270. DOI: 10.1007/s10207-016-0334-0. [Online]. Available: https://doi.org/10.1007/s10207-016-0334-0.

[3] A. Masood and J. Java, “Static analysis for web service security - tools & techniques for a secure development life cycle,” in *2015 IEEE International Symposium on Technologies for Homeland Security (HST)*, IEEE, Apr. 2015, pp. 1–6. DOI: 10.1109/ths.2015.7225337.

[4] R. T. Fielding and R. N. Taylor, “Architectural styles and the design of network-based software architectures,” PhD thesis, University of California, 2000.

[5] M. Vieira, N. Antunes, and H. Madeira, “Using web security scanners to detect vulnerabilities in web services,” in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, IEEE, Jun. 2009. DOI: 10.1109/dsn.2009.5270294.

[6] N. Antunes and M. Vieira, “Penetration testing for web services,” *Computer*, vol. 47, no. 2, pp. 30–36, Feb. 2014. DOI: 10.1109/mc.2013.409.

[7] G. McGraw, *Software security: building security in*. Addison-Wesley Professional, 2006, vol. 1.

[9] I. Schieferdecker, J. Grossmann, and M. Schneider, “Model-based security testing,” *EPTCS 80*, pp. 1–12, Feb. 28, 2012. DOI: 10.4204/EPTCS.80.1. arXiv: http://arxiv.org/abs/1202.6118v1 [cs.SE].

[10] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, “A systematic review of fuzzing techniques,” *Computers & Security*, vol. 75, pp. 118–137, Jun. 2018. DOI: 10.1016/j.cose.2018.02.002.

[11] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. DOI: 10.1145/96267.96279.

[12] S. Herbold and A. Hoffmann, “Model-based testing as a service,” *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 3, pp. 271–279, 2017. DOI: 10.1007/s10009-017-0449-2.

[13] M. Schneider, J. Grossmann, I. Schieferdecker, and A. Pietschker, “Online model-based behavioral fuzzing,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, IEEE, Mar. 2013. DOI: 10.1109/icstw.2013.61.

[15] V. Atlidakis, P. Godefroid, and M. Polishchuk, “Restler: Stateful rest api fuzzing,” in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE ’19, Montreal, Quebec, Canada: IEEE Press, May 31, 2019, pp. 748–758. DOI: `10.1109/ICSE.2019.00083`. [Online]. Available: `https://dl.acm.org/citation.cfm?id=3339600`.

[17] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, “Web services,” in *Web Services*, Springer Berlin Heidelberg, 2004, pp. 123–149. DOI: `10.1007/978-3-662-10876-5_5`.

[24] M. N. Lucky, M. Cremaschi, B. Lodigiani, A. Menolascina, and F. D. Paoli, “Enriching API descriptions by adding API profiles through semantic annotation,” in *Service-Oriented Computing*, Springer International Publishing, 2016, pp. 780–794. DOI: `10.1007/978-3-319-46295-0_55`.

[25] R Tsouroplis, M. Petychakis, I. Alvertis, E Biliri, F. Lampathaki, and D. Askounis, “Community-based api builder to manage apis and their connections with cloud-based services,” *CEUR Workshop Proceedings*, vol. 1367, pp. 17–23, Jan. 1, 2015. [Online]. Available: `https://www.researchgate.net/publication/283771397_Community-based_API_builder_to_manage_APIs_and_their_connections_with_cloud-based_services`.

[31] G Tian-yang, S Yin-sheng, and F You-yuan, “Research on software security testing,” *Engineering and Technology*, vol. 69, pp. 647–651, 2010.

[32] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschner, “Model-based security testing: A taxonomy and systematic classification,” *Software Testing, Verification and Reliability*, vol. 26, no. 2, pp. 119–148, Jul. 2015. DOI: `10.1002/stvr.1580`.

[33] G. J. Myers, *The Art of Software Testing, Second Edition*. Wiley, 2004, ISBN: 0-471-46912-2.

[34] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, Apr. 2011. DOI: `10.1002/stvr.456`.

[35] M. Sarma, P. V. R. Murthy, S. Jell, and A. Ulrich, “Model-based testing in industry - a case study with two mbt tools,” in *Proceedings of the 5th Workshop on Automation of Software Test*, ACM Press, 2010, pp. 87–90. DOI: `10.1145/1808266.1808279`.

[36] T. Fertig and P. Braun, “Model-driven testing of restful apis,” in *Proceedings of the 24th International Conference on World Wide Web*, ACM Press, 2015, pp. 1497–1502. DOI: `10.1145/2740908.2743045`.

[37] P. V. P. Pinheiro, A. T. Endo, and A. Simao, “Model-based testing of restful web services using uml protocol state machines,” in *Brazilian Workshop on Systematic and Automated Software Testing*, 2013.

[38] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007, ISBN: 0321446119.

[39] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[41] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, “A taint based approach for smart fuzzing,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE, Apr. 2012, pp. 818–825. DOI: `10.1109/icst.2012.182`.

[42] M. O. Shudrak and V. V. Zolotarev, “Improving fuzzing using software complexity metrics,” in *Information Security and Cryptology - ICISC 2015*, Springer International Publishing, 2016, pp. 246–261. DOI: `10.1007/978-3-319-30840-1_16`. [Online]. Available: `https://www.researchgate.net/publication/312577578_improving_fuzzing_using_software_complexity_metrics`.

[45] M. Petticrew and H. Roberts, *Systematic Reviews in the Social Sciences*, M. Petticrew and H. Roberts, Eds. Blackwell Publishing Ltd, Jan. 2008. DOI: `10.1002/9780470754887`.

[58] S. Karumanchi and A. C. Squicciarini, “In the wild: A large scale study of web services,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC ’14*, ACM Press, 2014. DOI: `10.1145/2554850.2555010`.

[59] A. Ghourabi, T. Abbes, and A. Bouhoula, “Experimental analysis of attacks against web services and countermeasures,” in *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services - iiWAS ’10*, ACM Press, 2010. DOI: `10.1145/1967486.1967519`.

[60] N. Antunes and M. Vieira, “Detecting SQL injection vulnerabilities in web services,” in *2009 Fourth Latin-American Symposium on Dependable Computing*, IEEE, Sep. 2009. DOI: `10.1109/ladc.2009.21`.

[62] S. Iqbal, M. L. M. Kiah, B. Dhaghighi, M. Hussain, S. Khan, M. K. Khan, and K.-K. R. Choo, “On cloud security attacks: A taxonomy and intrusion detection and prevention as a service,” *Journal of Network and Computer Applications*, vol. 74, pp. 98–120, Oct. 2016. DOI: `10.1016/j.jnca.2016.08.016`.

[63] S. Jan, C. D. Nguyen, and L. C. Briand, “Automated and effective testing of web services for XML injection attacks,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, ACM Press, 2016. DOI: `10.1145/2931037.2931042`.

[64] M. Jensen, N. Gruschka, R. Herkenhoner, and N. Luttenberger, “SOA and web services: New technologies, new standards - new attacks,” in *Fifth European Conference on Web Services (ECOWS ’07)*, IEEE, Nov. 2007. DOI: `10.1109/ecows.2007.9`.

[65] M. I. P. Salas, P. L. D. Geus, and E. Martins, “Security testing methodology for evaluation of web services robustness - case: XML injection,” in *2015 IEEE World Congress on Services*, IEEE, Jun. 2015. DOI: `10.1109/services.2015.53`.

[66] W. Yu, P. Supthaweesuk, and D. Aravind, “Trustworthy web services based on testing,” in *IEEE International Workshop on Service-Oriented System Engineering (SOSE ’ 05)*, IEEE, 2005. DOI: 10.1109/sose.2005.38.

[67] S. Salva, P. Laurencot, and I. Rabhi, “An approach dedicated for web service security testing,” in *2010 Fifth International Conference on Software Engineering Advances*, IEEE, Aug. 2010. DOI: 10.1109/icsea.2010.84.

[70] M. Salas and E. Martins, “Security testing methodology for vulnerabilities detection of XSS in web services and ws-security,” *Electronic Notes in Theoretical Computer Science*, vol. 302, pp. 133–154, Feb. 2014. DOI: 10.1016/j.entcs.2014.01.024.

[72] T. Vissers, T. S. Somasundaram, L. Pieters, K. Govindarajan, and P. Hellinckx, “DDoS defense system for web services in a cloud environment,” *Future Generation Computer Systems*, vol. 37, pp. 37–45, Jul. 2014. DOI: 10.1016/j.future.2014.03.003.

[74] V. R. Mouli and K. Jevitha, “Web services attacks and security - a systematic literature review,” *Procedia Computer Science*, vol. 93, pp. 870–877, 2016. DOI: 10.1016/j.procs.2016.07.265.

[75] J. Muñoz-Arteaga, E. B. Fernandez, and H. Caudel-García, “Misuse pattern: Spoofing web services,” in *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs - AsianPLoP ’11*, ACM Press, 2011. DOI: 10.1145/2524629.2524643.

[76] S. A. Tronvoll, C. W. Elverum, and T. Welo, “Prototype experiments: Strategies and trade-offs,” *Procedia CIRP*, vol. 60, pp. 554–559, 2017. DOI: 10.1016/j.procir.2017.01.049.

[77] M. Carr and J. Verner, “Prototyping and software development approaches,” *Department of Information Systems, City University of Hong Kong, Hong Kong*, pp. 319–338, 1997. [Online]. Available: https://www.researchgate.net/publication/238117215_Prototyping_and_Software_Development_Approaches.

[78] J. D. Naumann and A. M. Jenkins, “Prototyping: The new paradigm for systems development,” *MIS Quarterly*, vol. 6, no. 3, p. 29, Sep. 1982. DOI: 10.2307/248654.

[79] R. Nacheva, “Prototyping approach in user interface development,” in -, Jun. 1, 2017. [Online]. Available: https://www.researchgate.net/publication/317414969_prototyping_approach_in_user_interface_development.

[80] G. D. Everett and R. McLeod Jr, *Software testing: testing across the entire software development life cycle.* John Wiley & Sons, 2007.

[95] H. Radwan and K. Prole, “Code pulse: Real-time code coverage for penetration testing activities,” in *2015 IEEE International Symposium on Technologies for Homeland Security (HST)*, IEEE, Apr. 2015. DOI: 10.1109/ths.2015.7225269.

[98] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison Wesley, Nov. 1, 2002, 533 pp., ISBN: 0321127420. [Online]. Available: `https://www.ebook.de/de/product/3253239/martin_fowler_patterns_of_enterprise_application_architecture.html`.

[100] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Prentice Hall, Dec. 1, 1995, ISBN: 0201633612. [Online]. Available: `https://www.ebook.de/de/product/3236753/erich_gamma_richard_helm_ralph_e_johnson_john_vlissides_design_patterns.html`.

[101] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000. [Online]. Available: `http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf`.

[111] D. Luna, L. Allodi, and M. Cremonini, "Productivity and patterns of activity in bug bounty programs," in *Proceedings of the 14th International Conference on Availability, Reliability and Security - ARES '19*, ACM Press, 2019. DOI: `10.1145/3339252.3341495`.

[113] K. Bhargavan, C. Fournet, A. D. Gordon, and G. O'Shea, "An advisor for web services security policies," in *Proceedings of the 2005 workshop on Secure web services - SWS '05*, ACM Press, 2005. DOI: `10.1145/1103022.1103024`.

[114] K. Bhargavan, C. Fournet, and A. D. Gordon, "Verifying policy-based security for web services," in *Proceedings of the 11th ACM conference on Computer and communications security - CCS '04*, ACM Press, 2004. DOI: `10.1145/1030083.1030120`.

[115] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for SQL injection vulnerabilities: An input mutation approach," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, ACM Press, 2014. DOI: `10.1145/2610384.2610403`.

[116] N. Antunes and M. Vieira, "Comparing the effectiveness of penetration testing and static code analysis on the detection of SQL injection vulnerabilities in web services," in *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, IEEE, Nov. 2009. DOI: `10.1109/prdc.2009.54`.

[117] Q. Li, J. Chen, Y. Zhan, C. Mao, and H. Wang, "Combinatorial mutation approach to web service vulnerability testing based on SOAP message mutations," in *2012 IEEE Ninth International Conference on e-Business Engineering*, IEEE, Sep. 2012. DOI: `10.1109/icebe.2012.34`.

[118] N. Antunes and M. Vieira, "Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 269–283, Mar. 2015. DOI: `10.1109/tsc.2014.2310221`.

[119] N. Laranjeiro, M. Vieira, and H. Madeira, "A learning-based approach to secure web services from SQL/XPath injection attacks," in *2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*, IEEE, Dec. 2010. DOI: `10.1109/prdc.2010.24`.

[120] N. Antunes and M. Vieira, “Evaluating and improving penetration testing in web services,” in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, IEEE, Nov. 2012. DOI: 10.1109/issre.2012.26.

[121] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, “Effective detection of SQL/XPath injection vulnerabilities in web services,” in *2009 IEEE International Conference on Services Computing*, IEEE, 2009. DOI: 10.1109/scc.2009.23.

Other

[8] OWASP. (Sep. 14, 2020). Owasp application security verification standard | owasp foundation, [Online]. Available: https://owasp.org/www-project-application-security-verification-standard/.

[14] A. Muntner. (May 26, 2019). Dictionary of attack patterns and primitives for black-box application fault injection and resource discovery., [Online]. Available: https://github.com/fuzzdb-project/fuzzdb.

[16] GitLab. (Feb. 2, 2019). Gitlab, [Online]. Available: https://about.gitlab.com/.

[18] W3C Working Goup. (Feb. 11, 2004). Web services architecture. W3C Working Goup, Ed., [Online]. Available: https://www.w3.org/TR/ws-arch/.

[19] Marc Hadley (Sun Microsystems, Inc). (Sep. 25, 2020). Web application description language, [Online]. Available: https://www.w3.org/Submission/wadl/.

[20] Erik Christensen (Microsoft), Francisco Curbera (IBM Research), Greg Meredith (Microsoft), Sanjiva Weerawarana (IBM Research). (Sep. 25, 2020). Web services description language (wsdl), [Online]. Available: https://www.w3.org/TR/wsdl.html.

[21] RAML Workgroup. (Apr. 14, 2019). Restful api modeling language. R. Workgroup, Ed., [Online]. Available: https://raml.org/.

[22] OData. (May 12, 2019). Odata - the best way to rest. OData, Ed., [Online]. Available: https://www.odata.org/.

[23] Open API Initiative. (Feb. 3, 2019). Open api specification, [Online]. Available: https://www.openapis.org/.

[26] L. Heritage. (Feb. 1, 2015). Api description languages: Which is the right one for me? [Online]. Available: https://www.slideshare.net/SOA_Software/api-description-languages-which-is-the-right-one-for-me.

[27] Google. (Jun. 30, 2019). Google trends, [Online]. Available: https://trends.google.nl/trends/explore?date=2014-01-01%202019-06-01&q=%2Fm%2F010ppjcy,%2Fm%2F0bhb11q,%2Fm%2F02q151f,%2Fm%2F01082xym&hl=nl&tz=-120.

[28] Swagger. (Feb. 3, 2019). Swagger, [Online]. Available: https://swagger.io/.

[29] Swagger. (Jun. 15, 2019). Swagger ui, [Online]. Available: https://swagger.io/tools/swagger-ui/.

[30] Swagger. (Jun. 15, 2019). Swagger core, [Online]. Available: https://github.com/swagger-api/swagger-core.

[40] Jenkins. (Aug. 17, 2019). Jenkins, [Online]. Available: https://jenkins.io/.

[43] OWASP. (Jul. 1, 2019). Owasp zed attack proxy project. OWASP, Ed., [Online]. Available: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

[44] PortSwigger. (Jul. 1, 2019). Burp suite scanner - portswigger. PortSwigger, Ed., [Online]. Available: https://portswigger.net/burp.

[46] IEEE. (Jun. 2, 2019). 2019 ieee thesaurus. IEEE, Ed., [Online]. Available: https://www.ieee.org/content/dam/ieee-org/ieee/web/org/pubs/ieee-thesaurus.pdf.

[47] ACM Computing. (Nov. 2, 2019). The 2012 acm computing classification system, [Online]. Available: https://www.acm.org/publications/class-2012.

[48] IEEE. (Nov. 2, 2019). Ieee xplore digital library, [Online]. Available: https://ieeexplore.ieee.org/.

[49] ACM Computing. (Nov. 2, 2019). Acm digital library, [Online]. Available: https://dl.acm.org/dl.cfm.

[50] ScienceDirect. (Nov. 2, 2019). Sciencedirect.com | science, health and medical journals, full text articles and books., [Online]. Available: https://www.sciencedirect.com/.

[51] MAXQDA. (Dec. 18, 2019). Maxqda | all-in-one tool for qualitative data analysis & mixed methods - maxqda - the art of data analysis, [Online]. Available: www.maxqda.com.

[52] OWASP. (Dec. 27, 2019). Owasp top 10 - 2017, [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_(en).pdf.pdf.

[53] NIST. (Jan. 3, 2020). Nvd - home, [Online]. Available: https://nvd.nist.gov/.

[54] NIST. (Dec. 27, 2019). National institute of standards and technology | nist, [Online]. Available: https://www.nist.gov/.

[55] CVE. (Jan. 5, 2020). Cve - common vulnerabilities and exposures (cve), [Online]. Available: http://cve.mitre.org/.

[56] NIST. (Oct. 27, 2019). Nvd - data feeds, [Online]. Available: https://nvd.nist.gov/vuln/data-feeds#JSON_FEED.

[57] CWE. (Nov. 10, 2019). Cwe - common weakness enumeration, [Online]. Available: https://cwe.mitre.org/.

[61] NVD. (Jan. 12, 2020). Nvd - cve-2018-1289, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-1289.

[68] NVD. (Jan. 12, 2012). Nvd - cve-2019-1867, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-1867.

[69] NVD. (Jan. 12, 2012). Nvd - cve-2019-3403, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-3403.

[71] NVD. (Jan. 12, 2012). Nvd - cve-2019-16104, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-16104.

[73] NVD. (Jan. 28, 2012). Nvd - cve-2017-12287, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-12287.

[81] OWASP. (Sep. 18, 2020). Owasp webgoat - learn the hack - stop the attack. OWASP, Ed., [Online]. Available: https://owasp.org/www-project-webgoat/.

[82] Google. (Jun. 10, 2019). Google, [Online]. Available: https://www.google.nl/.

[83] BuiltWith. (Mar. 1, 2020). Builtwith technology lookup, [Online]. Available: https://builtwith.com/.

[84] WordPress. (Feb. 29, 2020). Blog tool, publishing platform, and cms - wordpress.org, [Online]. Available: https://wordpress.org/.

[85] Drupal. (Feb. 29, 2020). Drupal - open source cms | drupal.org, [Online]. Available: https://www.drupal.org/.

[86] Magento. (Mar. 1, 2020). Ecommerce platforms | best ecommerce software for selling online | magento, [Online]. Available: https://magento.com/.

[87] MediaWiki. (Feb. 29, 2020). Mediawiki, [Online]. Available: https://www.mediawiki.org/wiki/MediaWiki.

[88] Bloomreach. (Mar. 1, 2020). Java-based headless cms | bloomreach developers - bloomreach cms, [Online]. Available: https://developers.bloomreach.com/products/cms.

[89] XAMPP. (Mar. 1, 2020). Xampp installers and downloads for apache friends, [Online]. Available: https://www.apachefriends.org/.

[90] MariaDB Foundation. (Aug. 9, 2020). Mariadb foundation - mariadb.org, [Online]. Available: https://mariadb.org/.

[91] A. Suroyo. (Mar. 1, 2020). Wp api swaggerui, [Online]. Available: https://wordpress.org/plugins/wp-api-swaggerui/.

[92] Spring. (Mar. 22, 2020). Spring boot, [Online]. Available: https://spring.io/projects/spring-boot.

[93] Maven. (Mar. 6, 2020). Maven | welcome to apache maven, [Online]. Available: http://maven.apache.org/.

[94] Springfox. (Sep. 20, 2020). Springfox by springfox, [Online]. Available: https://springfox.github.io/springfox/.

[96] OWASP. (Jun. 26, 2020). Code pulse | real-time code coverage, [Online]. Available: http://code-pulse.com.

[97] PHPUnit. (Jun. 26, 2020). Phpunit - the php testing framework, [Online]. Available: `https://phpunit.de/`.

[99] Hibernate. (May 9, 2020). Ibernate. everything data. - hibernate, [Online]. Available: `http://hibernate.org/`.

[102] Webpack. (Mar. 23, 2020). Webpack, [Online]. Available: `https://webpack.js.org/`.

[103] Vue.js. (Jun. 14, 2019). Vue.js - the progressive javascript framework, [Online]. Available: `https://vuejs.org/`.

[104] Bootstrap team. (Mar. 23, 2020). Bootstrap · the most popular html, css, and js library in the world., [Online]. Available: `https://getbootstrap.com/`.

[105] Vue.js. (Mar. 23, 2020). Bootstrapvue, [Online]. Available: `https://bootstrap-vue.js.org/`.

[106] Vue.js. (Mar. 23, 2020). What is vuex? | vuex, [Online]. Available: `https://vuex.vuejs.org/`.

[107] Axios. (Mar. 23, 2020). Github - axios/axios: Promise based http client for the browser and node.js, [Online]. Available: `https://github.com/axios/axios`.

[108] DBeaver Community. (Apr. 1, 2020). Dbeaver community | free universal database tool, [Online]. Available: `https://dbeaver.io/`.

[109] Swagger. (Mar. 1, 2020). Swagger parser, [Online]. Available: `https://mvnrepository.com/artifact/io.swagger/swagger-parser`.

[110] Apache Software Foundation. (Aug. 10, 2020). Apache httpcomponents - httpcomponents httpclient overview, [Online]. Available: `https://hc.apache.org/httpcomponents-client-5.0.x/index.html`.

[112] Sara Rosso. (Aug. 13, 2020). Wordpress security white paper, [Online]. Available: `https://raw.githubusercontent.com/WordPress/Security-White-Paper/master/WordPressSecurityWhitePaper.pdf`.

Credits

Front cover has been designed using resources from Freepik.com.

Appendices

Appendix A: Results literature review

Table 11: Results systematic literature review found in ACM.

Reference	Title	Authors	Year of publication	Keywords	Research questions	Web service type(s)	Vulnerability type(s)	Research method	Short description
[63]	Automated and Effective Testing of Web Services for XML Injection Attacks	Sadeeq Jan, Cu D. Nguyen, Lionel C. Briand	2016	XML injection, security testing, constraint solving	RQ1 [Effectiveness]: Are the tools able to generate malicious messages (tests) bypassing the first layer of defence (the XML gateway) and thus reaching the targeted web services? RQ2 [Cost]: What is the cost of using the tools in terms of generation and execution time?	SOAP	XML injection	case study - case series	A tool called SOLMI is developed to manipulate XML messages by mutating operators. A constraint solver is used to generate valid though malicious XML messages, which are used as test cases.
[113]	An Advisor for Web Services Security Policies	Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, Greg O'Shea	2005	web services, XML security, WS-Security, policy-driven security	-	SOAP	XML rewriting	experiment	XML rewriting attacks on web services exist for a while now. Several formal analyses were developed to find these vulnerabilities. Usually there is a difference between the formal model and the implementation. This study offers a trade off between formal correctness and usability for detecting XML rewriting attacks on web services.
[114]	Verifying Policy-Based Security for Web Services	Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon	2004	web services, pi calculus, XML security	-	SOAP	XML rewriting	experiment	The tool described in this paper compiles a link specification to WS-Security policy configuration files. Also an analyser is developed to verify (before execution) whether the security goals are achieved. The analyser creates a formal model of a collection SOAP processors. With this tool the formal model can be verified for vulnerability to XML rewriting attacks.
[75]	Misuse Pattern: Spoofing Web Services	Jaime Muñoz-Arteaga, Eduardo B. Fernandez, Héctor Caudel-García	2011	authentication, misuse patterns, security, spoofing attacks, web services	-	SOAP	spoofing (principal)	case study - case series	A misuse pattern: spoofing web services, is presented. How the attack is performed and how to stop it is described in this study.

continued ...

Reference	Title	Authors	Year of publication	Keywords	Research questions	Web service type(s)	Vulnerability type(s)	Research method	Short description
[58]	In the Wild: a Large Scale Study of Web Services Vulnerabilities	Sushama Karumanchi, Anna Cinzia Squicciarini	2014	-	-	SOAP	confidentiality and integrity vulnerability, error on interface, invalid parser, invalid XML, logging vulnerability, credential exposure, session replay, SQL injection, XPath injection	case study - case series	A new taxonomy to classify vulnerabilities in web services is presented. Furthermore, 2000 WSDLs are statically checked according to this new taxonomy.
[59]	Experimental analysis of attacks against web services and countermeasures	Abdallah Ghourabi, Tarek Abbes, Adel Bouhoula	2010	web services attacks, web services security, command injection, session hijacking, DoS attack	-	SOAP	code execution, denial of service, parameter tampering, session hijacking, SQL injection, XML injection	experiment	Various vulnerabilities and attacks against web services are discussed. Three types of attacks are performed and emulated on web services. The impact of these attacks are observed. Also, countermeasures are proposed to prevent and mitigate those attacks.
[115]	Automated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach	Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, Nadia Alshahwan	2014	mutation testing, SQL injection, test generation	RQ1: Are standard attacks and mutated inputs (generated by 4SQLi) likely to reveal exploitable SQLi vulnerabilities? RQ2: With and without the presence of the WAF, which input generation technique performs better? -	SOAP	SQL injection	experiment	A black-box automated testing approach targeting SQL injection vulnerabilities is proposed in this study, named 4SQLi. Starting with legal test cases, this approach applied a set of mutation operators that are developed to increase the chance of exploiting SQL injection vulnerabilities.

Table 12: Results systematic literature review found in IEEE.

Reference	Title	Authors	Year of publication	Keywords	Research questions	Web service type(s)	Vulnerability type(s)	Research method	Short description
[60]	Detecting SQL Injection Vulnerabilities in Web Services	Nuno Antunes, Marco Vieira	2009	-	-	SOAP	SQL injection	experiment	A new method for the detection of SQL injection vulnerabilities in web services is presented. Malicious tests, also named attacks, are executed to exploit SQL injection vulnerabilities. Rules are defined to eliminate false positives. The experiment was performed on 262 public and 4 private web services.
[116]	Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services	Nuno Antunes, Marco Vieira	2009	security, vulnerabilities, SQL injection, penetration testing, static code analysis, web Services	-	SOAP	SQL injection	experiment	An experimental study on the comparison of several web vulnerability detection tools implementing either penetration-testing or static code analysis was conducted. Commercial and open source tools were compared to discover SQL Injection vulnerabilities in a collection of vulnerable services. Results for penetration testing tools and static code analysis tools were analysed separately and then compared to gain better understanding of the strengths and weaknesses of each approach.
[117]	Combinatorial Mutation Approach to Web Service Vulnerability Testing based on SOAP Message Mutations	Qing Li, Jinfu Chen, Yongzhao Zhan, Chengying Mao, Huanhuan Wang	2012	web service testing, SOAP message mutation, combinatorial testing, mutation operator, vulnerability testing	-	SOAP	XML injection	experiment	Many of the running state behaviours are contained due to the distributed features of web services. Therefore, new testing techniques are needed to test web services. This study presented a method to inject a collection of mutation operators that can be combined with combinatorial strategies. So, multiple mutants are injected together instead of one mutant at a time.
[118]	Assessing and Comparing Vulnerability Detection Tools for Web Services: Benchmarking Approach and Examples	Nuno Antunes, Marco Vieira	2014	benchmarking, vulnerability detection, penetration testing, static analysis, runtime anomaly detection	-	SOAP	SQL injection	experiment	A method was proposed that defined benchmarks for vulnerability detection tools in web services. Two benchmarks targeting tools able to detect SQL injection vulnerabilities have been developed. The first benchmark is based on a predefined workload, while the second allows a user defined workload. Several commercial and open-source tools have been benchmarked.
[67]	An Approach Dedicated for Web Service Security Testing	Sébastien Salva, Patrice Laurençot, Issam Rabhi	2010	web Services, security rules, test purposes, test generation	-	stateful	broken authentication, authorisation issues, unavailability, SQL injection, XML injection	experiment	A security testing method is proposed in this study dedicated to stateful web services. There are security rules defined with the Nomad language and are translated to so called test purposes. Then test cases are generated from these test purposes to verify whether the security rules are satisfied despite the use of malicious requests. Many security issues were detected on real web services with the developed solution.

continued ...

Reference	Title	Authors	Year of publication	Keywords	Research questions	Web service type(s)	Vulnerability type(s)	Research method	Short description
[119]	A Learning-Based Approach to Secure Web Services from SQL/XPath Injection Attacks	Nuno Laranjeiro, Marco Vieira, Henrique Madeira	2010	web services, security, SQL/XPath injection, vulnerabilities, code instrumentation	-	SOAP	SQL injection, XPath injection	experiment	In this study an approach was presented to improve the security of web services. Pattern learning was applied in a training period analysing real requests. This information was used later to block potentially malicious requests. To handle unknown cases during incomplete training another filter was used. This approach was effective in protecting various open source and private web services.
[5]	Using Web Security Scanners to Detect Vulnerabilities in Web Services	Marco Vieira, Nuno Antunes, Henrique Madeira	2009	-	RQ1: What is the coverage of the vulnerability scanners tested when used in a web services environment? RQ2: What is the false-positive rate of the vulnerability scanners when used in a web services environment? RQ3: What are the most common types of vulnerabilities in web services environments?	SOAP	buffer overflow, code execution, credential exposure, server path disclosure, SQL injection, XPath injection	experiment	Four commercial vulnerability scanners are compared in this study. There were 300 web services analysed. Many vulnerabilities were detected, confirming many web services were deployed without sufficient testing. Selecting a vulnerability scanner is a precise task. First, different types of vulnerabilities were detected by various scanners. Second, many notifications are false positives. Finally, the source code coverage of these scanners is rather low. A prominent observation is that SQL injection vulnerabilities are most commonly detected, because 84% of all types of vulnerabilities are typed as SQL injection.
[66]	Trustworthy Web Services Based on Testing	Weider D. Yu, Passarawarin Supthaweesuk, Dhanya Aravind	2005	-	-	SOAP	inadequate or missing access validation, forcing error, services traversal, script injection, buffer overflows, cross-site scripting, credential exposure, information exposure, spoofing, SQL injection	case study - case series	A procedure was presented on assessing web services security with the mindset of a malicious actor. The testing method is created by analysing common vulnerabilities in web applications and transferring these to the web service domain. Three important conclusions were drawn: no user input should be trusted, no security mechanism is to be trusted, and a successful test will only apply to the configuration that was used.
[64]	SOA and Web Services: New Technologies, New Standards - New Attacks	Meiko Jensen, Nils Gruschka, Ralph Herkenhöner, Norbert Luttenberger	2007	-	-	SOAP	WSDL scanning, denial of service (oversize payload, coercive parsing, oversized cryptography), spoofing (SOAPAction, metadata), XML injection	experiment	Various attack types are presented to discover different types of vulnerabilities in web services. Some of the vulnerabilities are caused by implementation failures, while most are introduced by protocol flaws. Also, countermeasures are presented to mitigate these vulnerability types.
[65]	Security Testing Methodology for Evaluation of Web Services Robustness - Case: XML Injection	Marcelo Invert Palma Salas, Paulo Lício de Geus, Eliane Martins	2015	web services, XML injection, fault injection, WSSecurity, UsernameToken	-	SOAP	XML injection	experiment	In this study 10 web services were tested on XML injection vulnerabilities (e.g. Cross-site scripting and XPath injection). WSInject is used as proxy between the web service and soapUI. A script is used for WSInject to mutate the XML messages. Also eight rules are defined to detect if XML injection was successful.

continued ...

Reference	Title	Authors	Year of publication	Keywords	Research questions	Web service type(s)	Vulnerability type(s)	Research method	Short description
[120]	Evaluating and Improving Penetration Testing in Web Services	Nuno Antunes, Marco Vieira	2012	web-services, security, vulnerability detection, attack signatures, penetration testing, interface monitoring	-	SOAP	SQL injection	experiment	A method using attack signatures and monitoring interfaces to analyse and improve attack and penetration testing tools was proposed and tested in this study. This enabled the detection of injection vulnerabilities in web services. A prototype was developed to detect SQL injection vulnerabilities. This prototype acts as a proxy and intercepts all requests to the web service, while injecting the original message. Furthermore, the communication with the database is monitored to validate if the attack was successful. Over 20 web services were tested by the prototype and compared with other tools.
[121]	Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services	Nuno Antunes, Nuno Laranjeiro, Marco Vieira, Henrique Madeira	2009	-	-	SOAP	SQL injection, XPath injection	experiment	A new approach is proposed in this study for the detection of SQL and XPath injection vulnerabilities in web services. An attack load generator generates malicious XPath and SQL commands. These are presented to the web service. This approach was implemented in a prototype (CIVS-WS). Automatic detection of these vulnerabilities was tested on 9 web services.

Table 13: Results systematic literature review found in ScienceDirect.

Reference	Title	Authors	Year of publication	Keywords	Research questions	Web service type(s)	Vulnerability type(s)	Research method	Short description
[72]	DDoS defense system for web services in a cloud environment	Thomas Vissers, Thamarai Selvi Somasundaramb, Luc Pieters, Kannan ovindarajan, Peter Hellinckx	2014	denial-of-service, cloud computing, mitigation, web services, SOAP, XML, HTTP	-	SOAP	(distributed) denial of service (HTTP flooding, oversized XML, coercive parsing, oversized encryption, WS-addressing spoofing)	experiment	Vulnerabilities present in the application-layer of web services are a great risk for the availability of these services. Different tests are presented and demonstrated the unavailability of web services due to denial of service attacks. Single machines and even single requests can cause web services to fail, even without a distributed attack. A defence system is proposed to mitigate these attacks.
[74]	Web Services Attacks and Security- A Systematic Literature Review	Varsha R Mouli, KP Jevitha	2016	web services, systematic literature review, attacks, security	RQ1: How much research has happened on web service security since 2005? RQ2: What are the web service security issues that are addressed in the research papers? RQ3: What are the techniques proposed to solve web service security issues? RQ4: Which are the web service security areas focused in the research papers?	SOAP, XML-RPC, JSON-RPC	denial of Service, SQL injection, XML injection, XPath injection, spoofing	literature review	In this study 36 papers on web services attacks were systematically analysed. Denial of service attacks are mentioned the most, followed by XML injection attacks. The focus is mainly on detecting attacks in these papers.
[70]	Security Testing Methodology for Vulnerabilities Detection of XSS in Web Services and WS-Security	M.I.P. Salas, E. Martins	2014	web services, cross-site scripting, XSS attack, penetration testing, fault injection, WS-Security, WSS, Security Token, soapUI, WSInject	-	SOAP	cross-site scripting	experiment	A method for determining the robustness of web services for fault injection using WSInject is presented in this paper. The tool enabled emulation and generation of cross-site scripting (XSS) attacks.
[62]	On cloud security attacks: A taxonomy and intrusion detection and prevention as a service	Salman Iqbal, Miss Laiha Mat Kiah, Babak Dhaghighi, Muzammil Hussain, Suleman Khan, Muhammad Khurram Khan, Kim-Kwang Raymond Choo	2016	cloud computing, taxonomy, security attacks, intrusion detection	-	SOAP, REST, RPC protocols	broken authentication, cross-site scripting, denial of service, SQL injection, XML signature wrapping	literature review	A literature study is executed to present different vulnerabilities on different layers in the cloud computing architecture. Software as a service (Saas), Platform as a service (Paas), and Infrastructure as a service (Iaas) are investigated. Each layer of the cloud computing architecture has its own vulnerability types. Further research will focus on a defence mechanism that works on each layer of the cloud computing architecture.

Table 14: REST web service related vulnerabilities found in the NVD database grouped per OWASP (2017) Top 10 item, CWE, and year.

Categorised vulnerability types (OWASP/CWE)	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2009 - 2019
A1 Injection	**0**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**4**	**5**	**13**
CWE-77 Improper Sanitization of Special Elements used in a Command (’Command Injection’)	0	0	0	0	0	0	0	0	0	1	3	4
CWE-89 Improper Sanitization of Special Elements used in an SQL Command (’SQL Injection’)	0	0	0	1	0	1	1	1	0	3	2	9
A2 Broken authentication	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**10**	**6**	**4**	**22**
CWE-287 Improper Authentication	0	0	0	0	1	0	0	0	8	4	3	16
CWE-384 Session Fixation	0	0	0	0	0	0	1	0	1	1	0	3
CWE-522 Insufficiently Protected Credentials	0	0	0	0	0	0	0	0	0	1	1	2
CWE-613 Insufficient Session Expiration	0	0	0	0	0	0	0	0	1	0	0	1
A3 Sensitive data exposure	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**4**	**0**	**5**
CWE-311 Missing Encryption of Sensitive Data	0	0	0	0	0	0	0	0	1	1	0	2
CWE-312 Cleartext Storage of Sensitive Information	0	0	0	0	0	0	0	0	0	2	0	2
CWE-319 Cleartext Transmission of Sensitive Information	0	0	0	0	0	0	0	0	0	1	0	1
A4 XML external entity reference	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**0**	**2**	**4**	**1**	**9**
CWE-611 Information Leak Through XML External Entity File Disclosure	0	0	0	0	0	0	2	0	2	4	1	9
A5 Broken access control	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**5**	**4**	**5**	**7**	**22**
CWE-22 Improper Limitation of a Pathname to a Restricted Directory (’Path Traversal’)	0	0	0	0	0	0	0	0	4	4	1	9
CWE-284 Access Control (Authorization) Issues	0	0	0	0	0	0	1	4	0	0	2	7
CWE-285 Improper Access Control (Authorization)	0	0	0	0	0	0	0	1	0	1	4	6
A7 Cross-Site Scripting (XSS)	**0**	**0**	**0**	**1**	**1**	**4**	**2**	**2**	**6**	**2**	**4**	**22**
CWE-79 Failure to Preserve Web Page Structure (’Cross-site Scripting’)	0	0	0	1	1	4	2	2	6	2	4	22
A8 Insecure deserialization	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**3**
CWE-502 Deserialization of Untrusted Data	0	0	0	0	0	0	0	1	1	1	0	3
Uncategorised	**1**	**0**	**2**	**7**	**9**	**10**	**20**	**25**	**32**	**25**	**34**	**165**
CWE-119 Failure to Constrain Operations within the Bounds of a Memory Buffer	0	0	0	0	0	0	0	0	2	0	0	2
CWE-17 Deprecated	0	0	0	0	0	0	1	0	0	0	0	1
CWE-190 Integer Overflow or Wraparound	0	0	0	0	0	0	0	0	0	0	1	1
CWE-20 Improper Input Validation	0	0	1	0	3	0	0	2	5	3	8	22
CWE-200 Information Exposure	0	0	0	0	1	2	8	8	5	5	3	32
CWE-254 7PK - Security Features	0	0	0	0	0	0	1	1	0	0	0	2
CWE-255 Credentials Management	0	0	0	0	0	0	0	0	0	0	4	4
CWE-264 Permissions, Privileges, and Access Controls	0	0	0	2	2	2	3	8	0	0	3	20
CWE-269 Improper Privilege Management	0	0	0	0	0	0	0	0	1	0	0	1

continued ...

Categorised vulnerability types (OWASP/CWE)	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2009 - 2019
CWE-275 Permission Issues	0	0	0	0	0	0	0	0	0	0	2	2
CWE-276 Incorrect Default Permissions	0	0	0	0	0	0	0	0	0	0	1	1
CWE-310 Cryptographic Issues	0	0	0	0	0	1	1	0	0	1	0	3
CWE-347 Improper Verification of Cryptographic Signature	0	0	0	0	0	0	0	0	0	0	1	1
CWE-352 Cross-Site Request Forgery (CSRF)	0	0	0	1	1	1	2	2	4	3	3	17
CWE-362 Race Condition	0	0	0	0	0	0	0	0	1	0	0	1
CWE-399 Resource Management Errors	0	0	0	0	0	0	3	0	0	0	0	3
CWE-400 Uncontrolled Resource Consumption (’Resource Exhaustion’)	0	0	0	0	0	0	0	0	0	0	2	2
CWE-404 Improper Resource Shutdown or Release	0	0	0	0	0	0	0	0	1	0	0	1
CWE-434 Unrestricted Upload of File with Dangerous Type	0	0	0	0	0	0	0	0	0	0	1	1
CWE-444 Inconsistent Interpretation of HTTP Requests (’HTTP Request Smuggling’)	0	0	0	0	0	0	0	0	0	3	0	3
CWE-532 Information Leak Through Log Files	0	0	0	0	0	0	0	0	0	1	0	1
CWE-59 Improper Link Resolution Before File Access (’Link Following’)	0	0	0	0	0	1	0	0	0	0	0	1
CWE-732 Incorrect Permission Assignment for Critical Resource	0	0	0	0	0	0	0	0	1	1	0	2
CWE-74 Failure to Sanitize Data into a Different Plane (’Injection’)	0	0	0	0	0	0	0	2	1	0	1	4
CWE-749 Exposed Dangerous Method or Function	0	0	0	0	0	0	0	1	0	0	0	1
CWE-770 Allocation of Resources Without Limits or Throttling	0	0	0	0	0	0	0	0	0	1	0	1
CWE-798 Uncontrolled Memory Allocation	0	0	0	0	0	0	0	0	1	0	1	2
CWE-829 Inclusion of Functionality from Untrusted Control Sphere	0	0	0	0	0	0	0	0	0	1	0	1
CWE-862 Missing Authorization	0	0	0	0	0	0	0	0	2	0	0	2
CWE-863 Incorrect Authorization	0	0	0	0	0	0	0	0	1	2	1	4
CWE-918 Server-Side Request Forgery (SSRF)	0	0	0	0	0	0	0	0	1	1	1	3
CWE-94 Failure to Control Generation of Code (’Code Injection’)	0	0	0	0	0	0	0	0	1	0	1	2
NVD-CWE-Other Other	0	0	0	0	0	2	2	0	0	0	0	4
NVD-CWE-noinfo Insufficient Information	1	0	1	4	2	1	0	2	5	3	0	19
Total number of REST web service related vulnerabilities	1	0	2	9	11	15	27	34	56	49	55	259
Total number of vulnerabilities	4968	5086	4628	5559	6182	8545	8296	10191	15883	15908	10007	95253
Percentage REST vulnerabilities	0.02%	0.00%	0.04%	0.16%	0.18%	0.18%	0.33%	0.33%	0.35%	0.31%	0.55%	0.27%

Appendix B: Architecture diagram

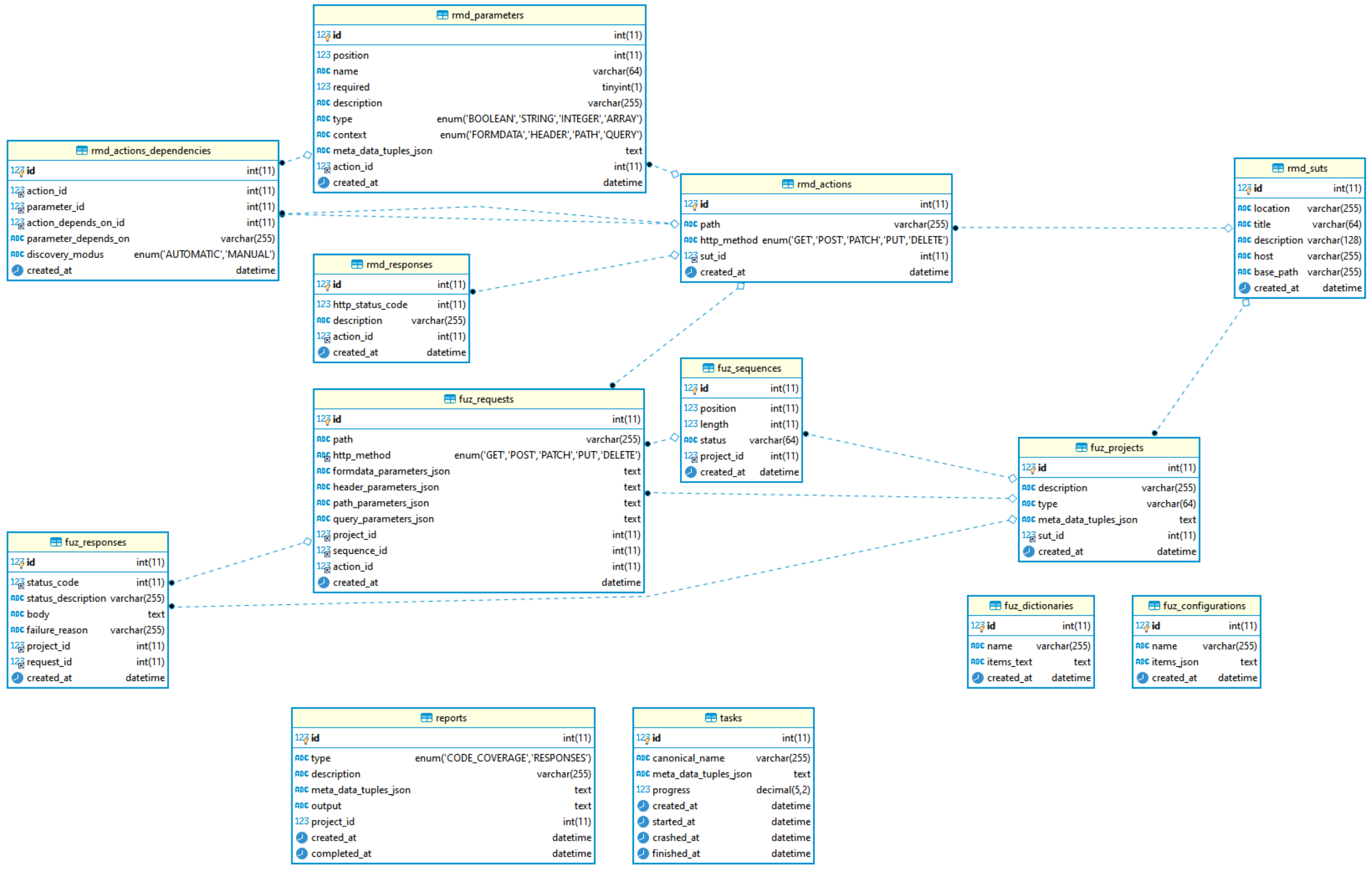
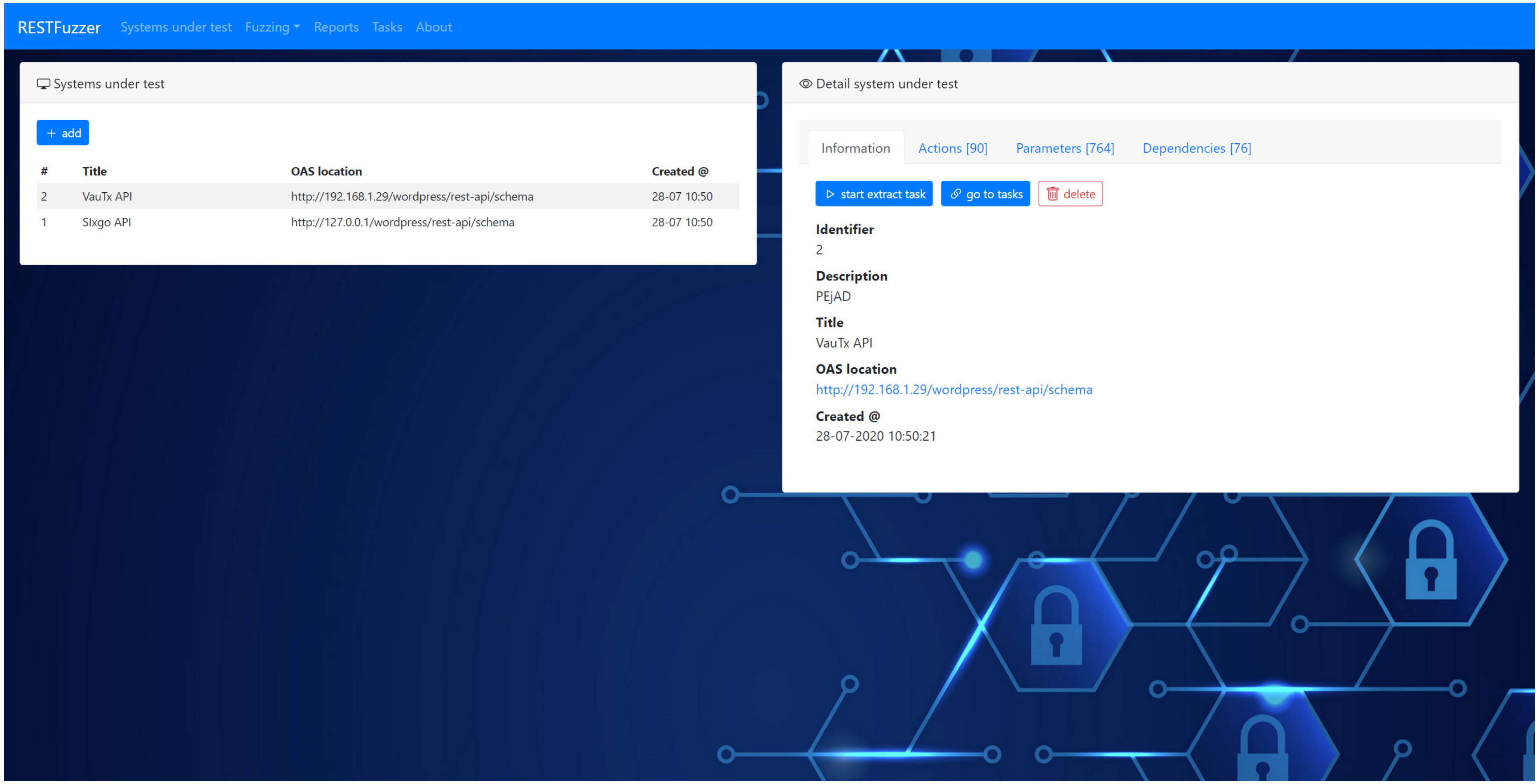


Figure 20: The entity relationship diagram of the database, visualising the tables, types, fieldnames, and relations between entities.

Appendix C: Screenshots RESTFuzzer



#	Title	OAS location	Created @
2	VauTx API	http://192.168.1.29/wordpress/rest-api/schema	28-07 10:50
1	SIxgo API	http://127.0.0.1/wordpress/rest-api/schema	28-07 10:50

Identifier
2

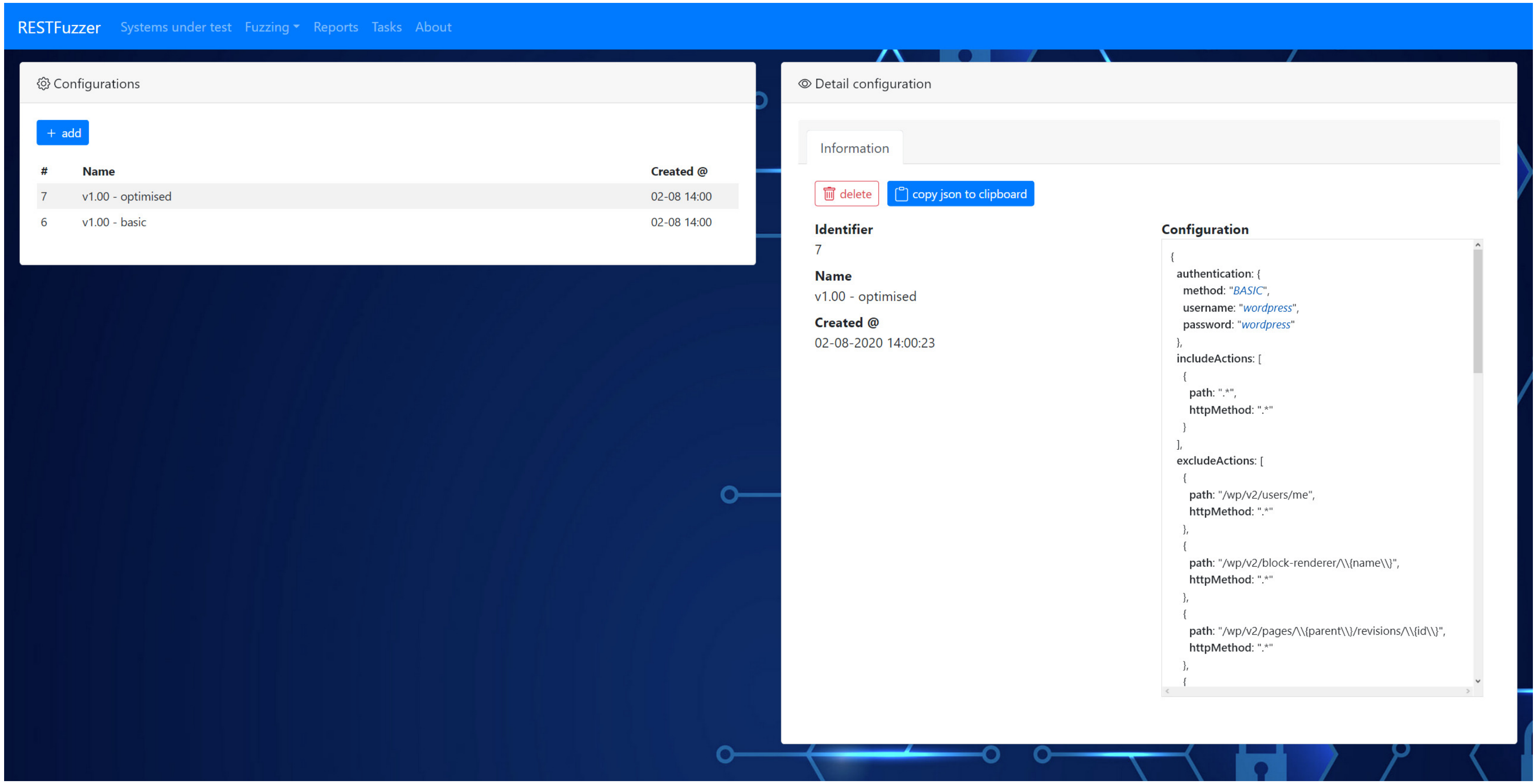
Description
PEjAD

Title
VauTx API

OAS location
http://192.168.1.29/wordpress/rest-api/schema

Created @
28-07-2020 10:50:21

Figure 21: Screenshot of the Systems Under Test overview page.



#	Name	Created @
7	v1.00 - optimised	02-08 14:00
6	v1.00 - basic	02-08 14:00

Identifier
7

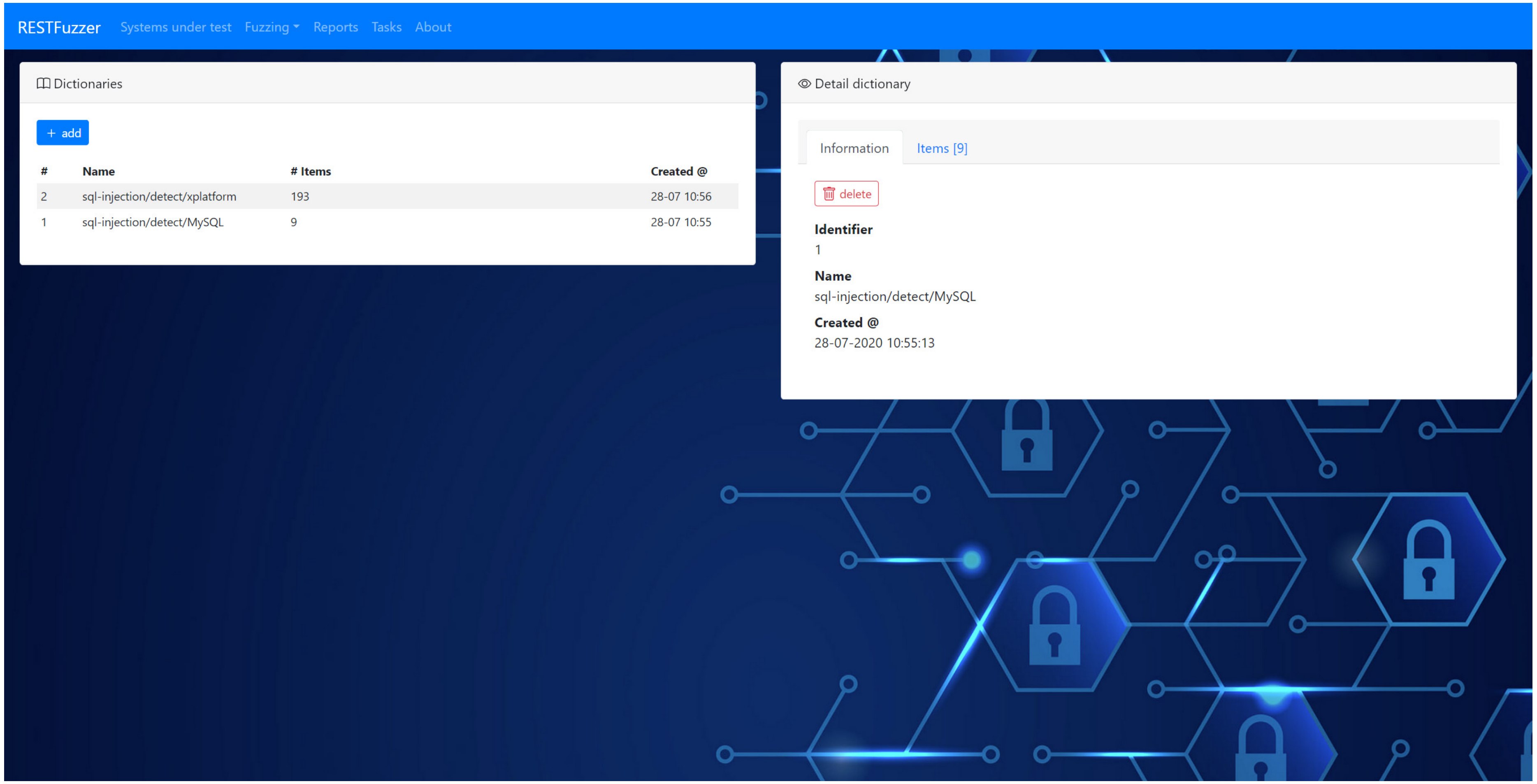
Name
v1.00 - optimised

Created @
02-08-2020 14:00:23

Configuration

```
{
  authentication: {
    method: "BASIC",
    username: "wordpress",
    password: "wordpress"
  },
  includeActions: [
    {
      path: ".*",
      httpMethod: ".*"
    }
  ],
  excludeActions: [
    {
      path: "/wp/v2/users/me",
      httpMethod: ".*"
    },
    {
      path: "/wp/v2/block-renderer/\\{name\\}",
      httpMethod: ".*"
    },
    {
      path: "/wp/v2/pages/\\{parent\\}/revisions/\\{id\\}",
      httpMethod: ".*"
    },
    {
```

Figure 22: Screenshot of the configurations overview page.



Dictionaries

+ add

#	Name	# Items	Created @
2	sql-injection/detect/xplatform	193	28-07 10:56
1	sql-injection/detect/MySQL	9	28-07 10:55

Detail dictionary

Information | Items [9]

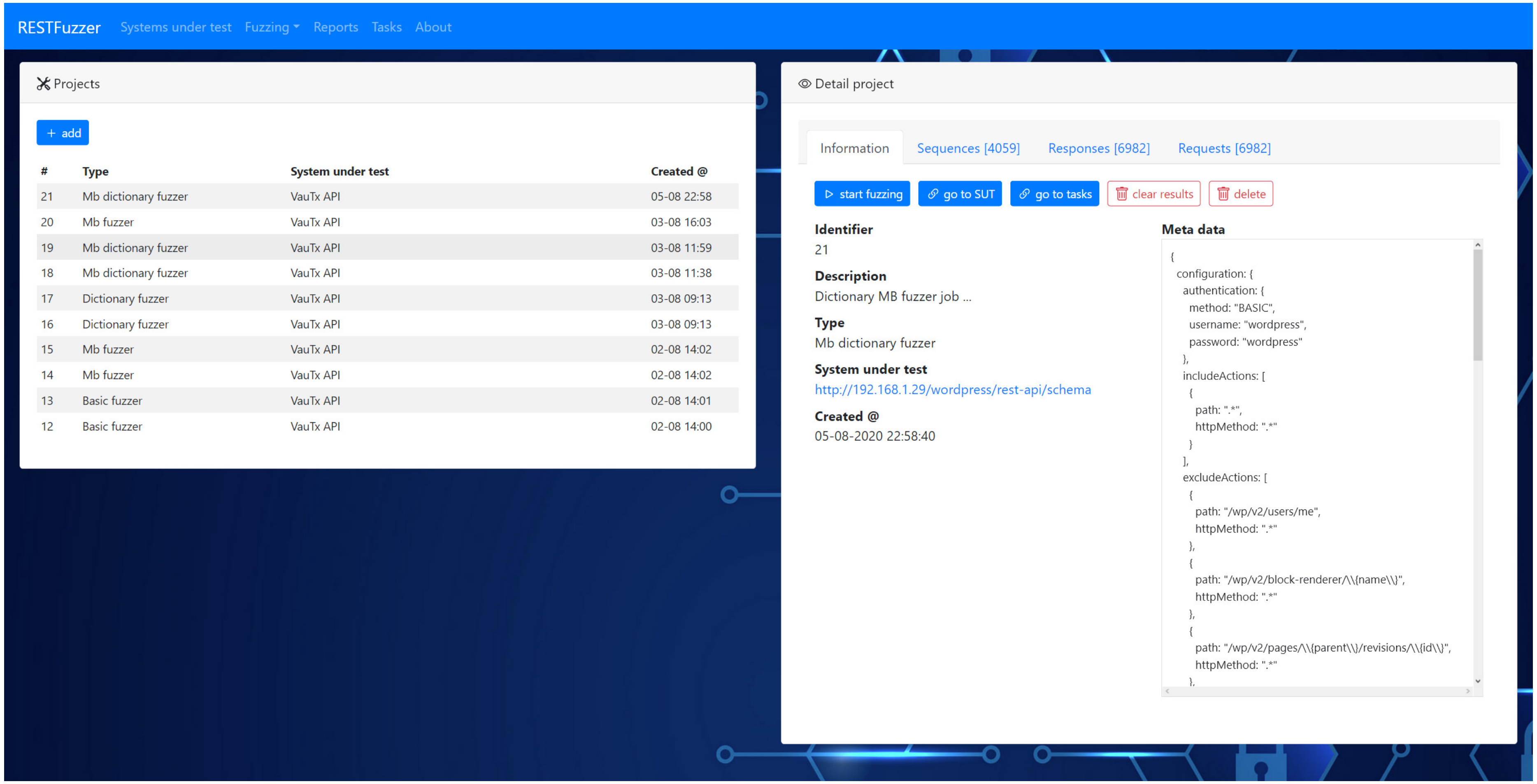
delete

Identifier
1

Name
sql-injection/detect/MySQL

Created @
28-07-2020 10:55:13

Figure 23: Screenshot of the dictionaries overview page.



#	Type	System under test	Created @
21	Mb dictionary fuzzer	VauTx API	05-08 22:58
20	Mb fuzzer	VauTx API	03-08 16:03
19	Mb dictionary fuzzer	VauTx API	03-08 11:59
18	Mb dictionary fuzzer	VauTx API	03-08 11:38
17	Dictionary fuzzer	VauTx API	03-08 09:13
16	Dictionary fuzzer	VauTx API	03-08 09:13
15	Mb fuzzer	VauTx API	02-08 14:02
14	Mb fuzzer	VauTx API	02-08 14:02
13	Basic fuzzer	VauTx API	02-08 14:01
12	Basic fuzzer	VauTx API	02-08 14:00

Identifier
21

Description
Dictionary MB fuzzer job ...

Type
Mb dictionary fuzzer

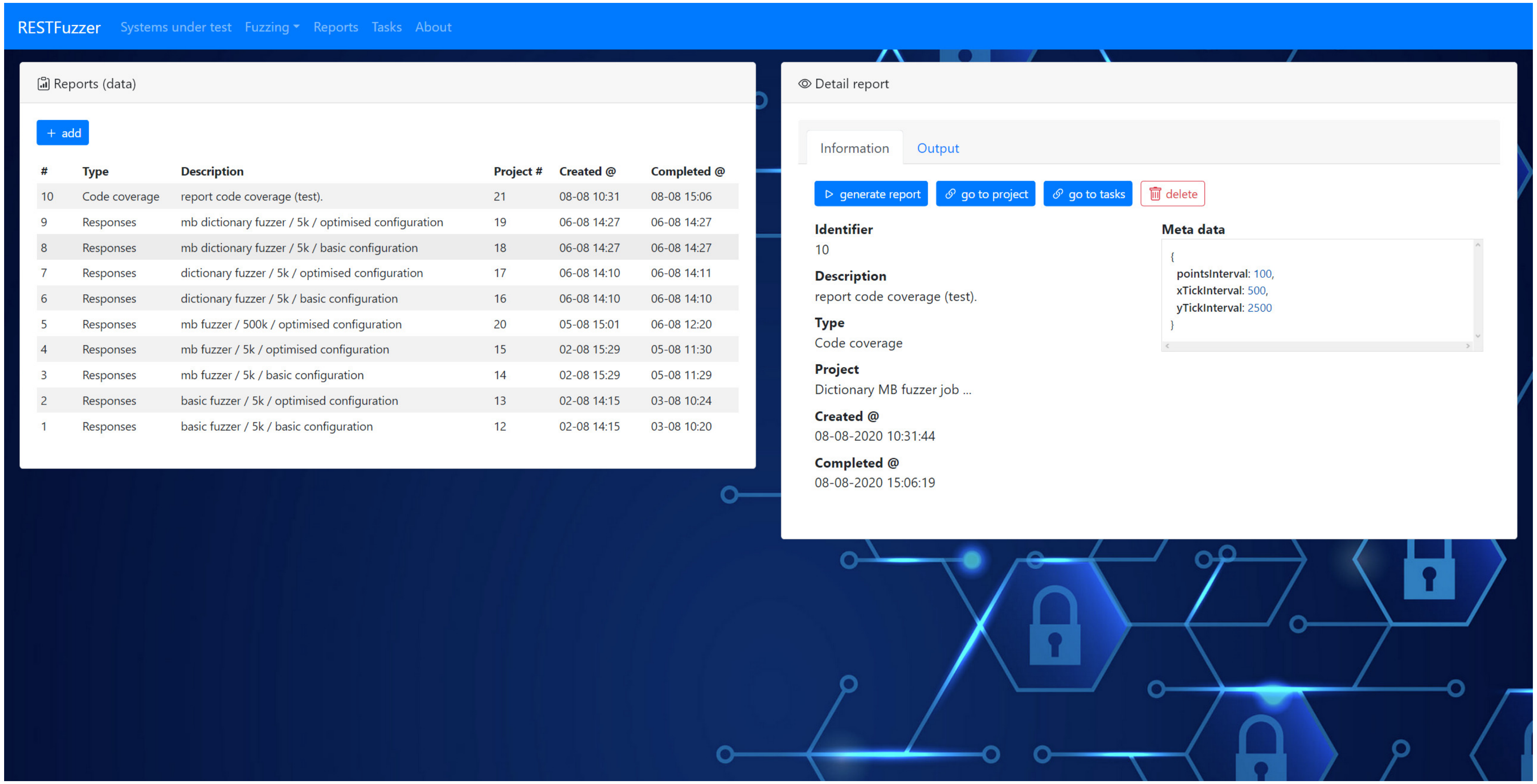
System under test
http://192.168.1.29/wordpress/rest-api/schema

Created @
05-08-2020 22:58:40

Meta data

```
{
  configuration: {
    authentication: {
      method: "BASIC",
      username: "wordpress",
      password: "wordpress"
    },
    includeActions: [
      {
        path: ".*",
        httpMethod: ".*"
      }
    ],
    excludeActions: [
      {
        path: "/wp/v2/users/me",
        httpMethod: ".*"
      },
      {
        path: "/wp/v2/block-renderer/\\{name\\}",
        httpMethod: ".*"
      },
      {
        path: "/wp/v2/pages/\\{parent\\}/revisions/\\{id\\}",
        httpMethod: ".*"
      },
```

Figure 24: Screenshot of the projects overview page.



Reports (data)

#	Type	Description	Project #	Created @	Completed @
10	Code coverage	report code coverage (test).	21	08-08 10:31	08-08 15:06
9	Responses	mb dictionary fuzzer / 5k / optimised configuration	19	06-08 14:27	06-08 14:27
8	Responses	mb dictionary fuzzer / 5k / basic configuration	18	06-08 14:27	06-08 14:27
7	Responses	dictionary fuzzer / 5k / optimised configuration	17	06-08 14:10	06-08 14:11
6	Responses	dictionary fuzzer / 5k / basic configuration	16	06-08 14:10	06-08 14:10
5	Responses	mb fuzzer / 500k / optimised configuration	20	05-08 15:01	06-08 12:20
4	Responses	mb fuzzer / 5k / optimised configuration	15	02-08 15:29	05-08 11:30
3	Responses	mb fuzzer / 5k / basic configuration	14	02-08 15:29	05-08 11:29
2	Responses	basic fuzzer / 5k / optimised configuration	13	02-08 14:15	03-08 10:24
1	Responses	basic fuzzer / 5k / basic configuration	12	02-08 14:15	03-08 10:20

Detail report

Identifier
10

Description
report code coverage (test).

Type
Code coverage

Project
Dictionary MB fuzzer job ...

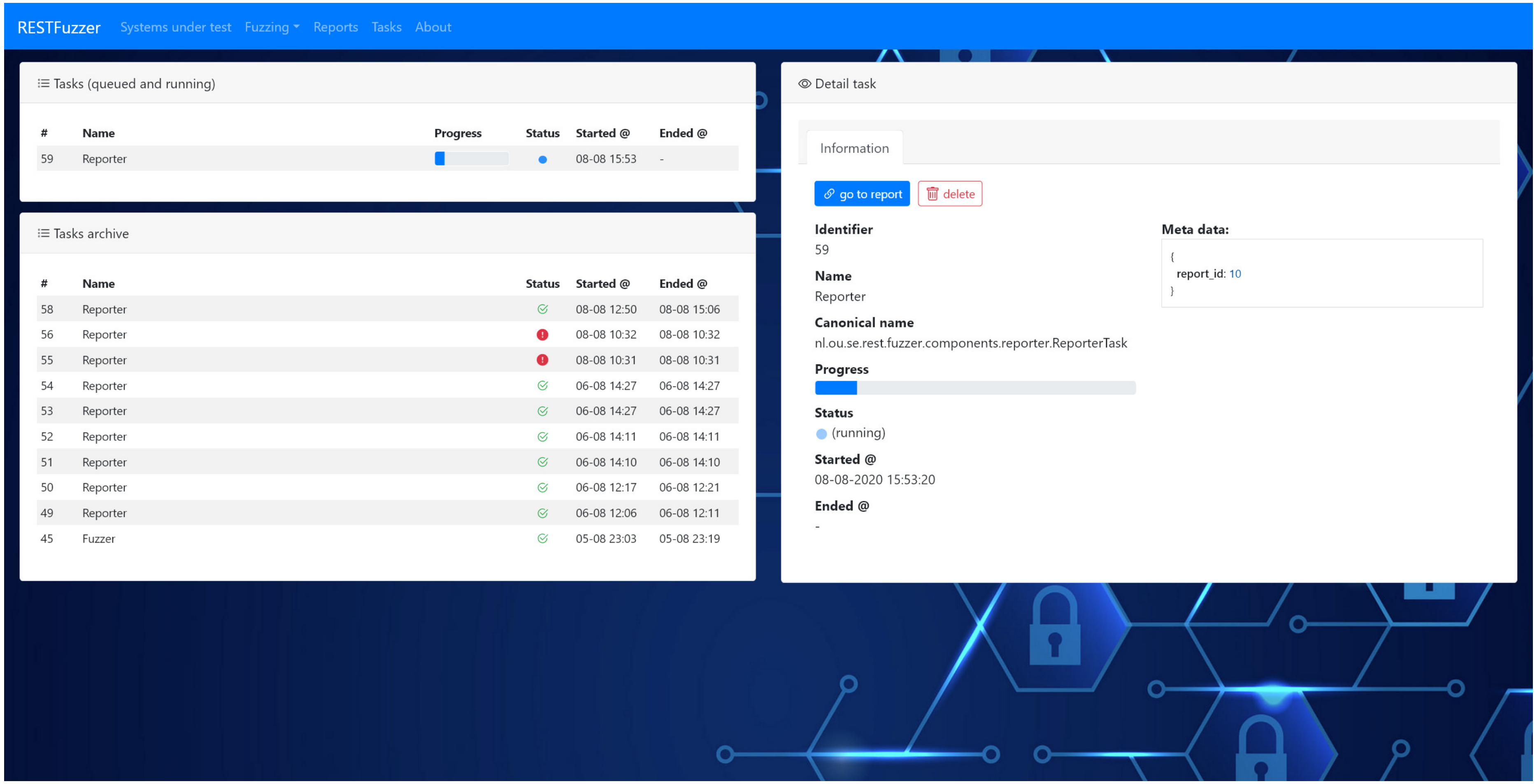
Created @
08-08-2020 10:31:44

Completed @
08-08-2020 15:06:19

Meta data

```
{
  pointsInterval: 100,
  xTickInterval: 500,
  yTickInterval: 2500
}
```

Figure 25: Screenshot of the reports overview page.



Tasks (queued and running)

#	Name	Progress	Status	Started @	Ended @
59	Reporter		●	08-08 15:53	-

Tasks archive

#	Name	Status	Started @	Ended @
58	Reporter	✓	08-08 12:50	08-08 15:06
56	Reporter	!	08-08 10:32	08-08 10:32
55	Reporter	!	08-08 10:31	08-08 10:31
54	Reporter	✓	06-08 14:27	06-08 14:27
53	Reporter	✓	06-08 14:27	06-08 14:27
52	Reporter	✓	06-08 14:11	06-08 14:11
51	Reporter	✓	06-08 14:10	06-08 14:10
50	Reporter	✓	06-08 12:17	06-08 12:21
49	Reporter	✓	06-08 12:06	06-08 12:11
45	Fuzzer	✓	05-08 23:03	05-08 23:19

Detail task

Identifier
59

Name
Reporter

Canonical name
nl.ou.se.rest.fuzzer.components.reporter.ReporterTask

Progress

Status
● (running)

Started @
08-08-2020 15:53:20

Ended @
-

Meta data:

```
{
  report_id: 10
}
```

Figure 26: Screenshot of the tasks overview page.

Appendix D: Information related to the experiments

Table 15: Specifications development environment.

Specification	Value
Operating System	Windows 10
CPU	Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz
RAM	16,0 GB
RESTFuzzer	v1.0.1

Table 16: Specifications test setup.

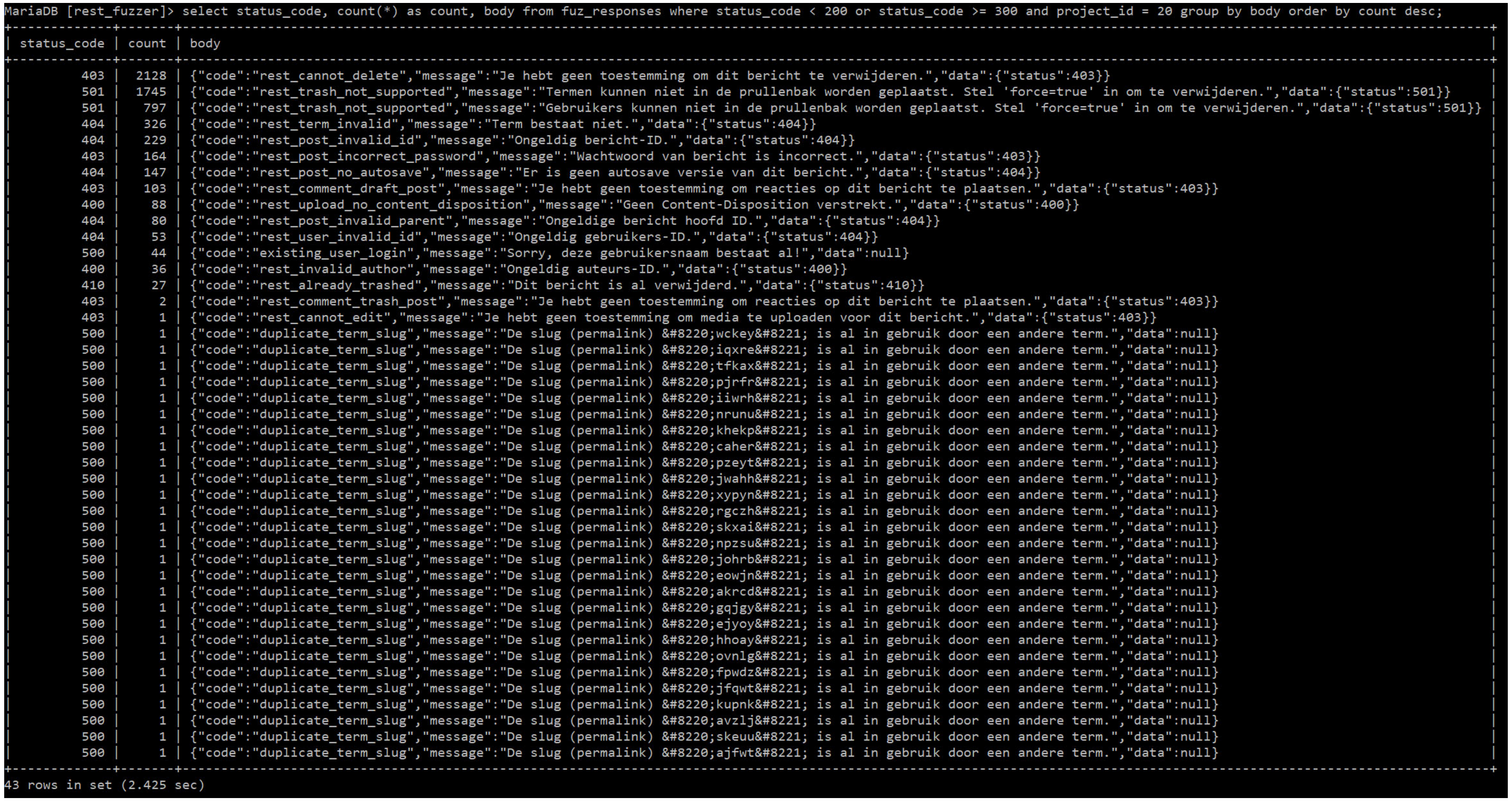
Specification	Value
Operating System	Ubuntu 20.04.1 LTS
CPU	Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz
RAM	4,0 GB
WordPress	5.4.2
SutSqlI	v1.00

```
{
  "authentication": {
    "method": "BASIC",
    "username": "wordpress",
    "password": "wordpress"
  },
  "includeActions": [
    {
      "path": ".*",
      "httpMethod": ".*"
    }
  ],
  "excludeActions": [
    {
      "path": "/wp/v2/users/me",
      "httpMethod": ".*"
    },
    {
      "path": "/wp/v2/users/\\{id\\}",
      "httpMethod": "PATCH|POST|PUT"
    }
  ],
  "excludeParameters": [],
  "defaults": []
}
```

Listing 4: Basic configuration used in experiments on REST web services WordPress.

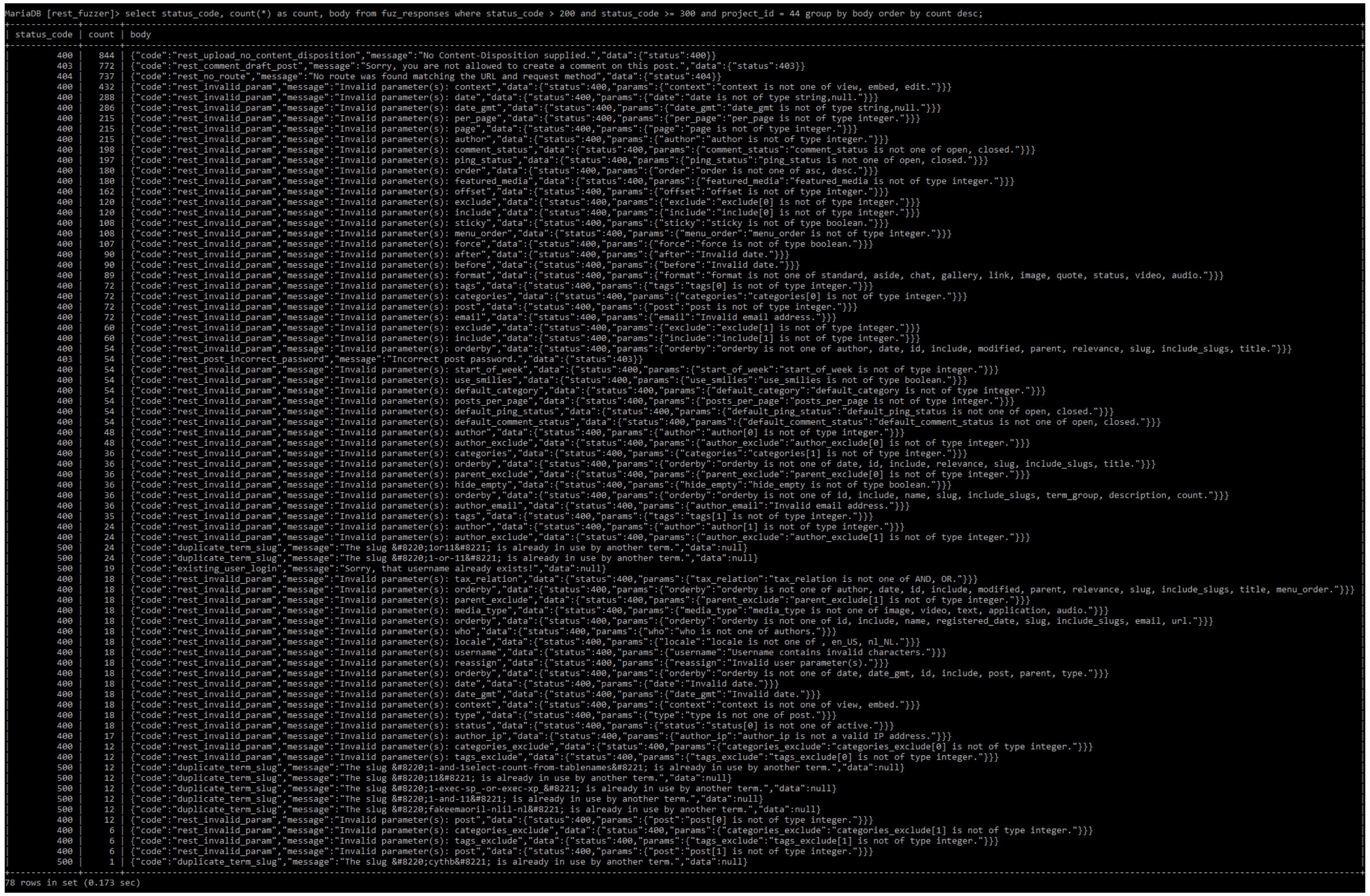
```json
{
  "authentication": {
    "method": "BASIC",
    "username": "wordpress",
    "password": "wordpress"
  },
  "includeActions": [
    {
      "path": ".*", "httpMethod": ".*"
    }
  ],
  "excludeActions": [
    {
      "path": "/wp/v2/users/me", "httpMethod": ".*"
    },
    {
      "path": "/wp/v2/block-renderer/\\{name\\}", "httpMethod": ".*"
    },
    {
      "path": "/wp/v2/pages/\\{parent\\}/revisions/\\{id\\}", "httpMethod": ".*"
    },
    {
      "path": "/wp/v2/posts/\\{parent\\}/revisions/\\{id\\}", "httpMethod": ".*"
    },
    {
      "path": "/wp/v2/users/\\{id\\}", "httpMethod": "PATCH|POST|PUT"
    },
    {
      "path": "/wp/v2/taxonomies.*", "httpMethod": "GET"
    },
    {
      "path": "/wp/v2/types/\\{type\\}", "httpMethod": "GET"
    },
    {
      "path": "/wp/v2/statuses/\\{status\\}", "httpMethod": "GET"
    }
  ],
  "excludeParameters": [
    {
      "action": {
        "path": ".*", "httpMethod": ".*"
      },
      "parameter": {
        "name": "template|meta|subtype|status|username|roles|parent",
        "required": "false"
      }
    }
  ],
  "defaults": [
    {
      "action": {
        "path": ".*", "httpMethod": ".*"
      },
      "parameter": {
        "name": "sticky",
        "required": ".*"
      },
      "default": "false"
    },
    {
      "action": {
        "path": "/wp/v2/themes", "httpMethod": "GET"
      },
      "parameter": {
        "name": "status",
        "required": ".*"
      },
      "default": "[active]"
    }
  ]
}
```

Listing 5: Optimised configuration used in experiments on REST web services WordPress.



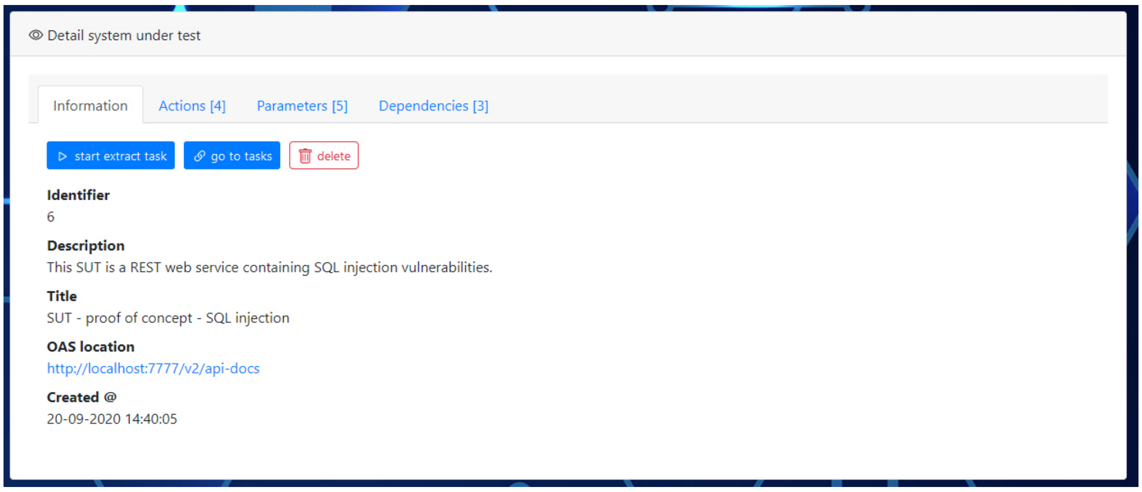
```
MariaDB [rest_fuzzer]> select status_code, count(*) as count, body from fuz_responses where status_code < 200 or status_code >= 300 and project_id = 20 group by body order by count desc;
+-------------+-------+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| status_code | count | body
+-------------+-------+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------+
|         403 |  2128 | {"code":"rest_cannot_delete","message":"Je hebt geen toestemming om dit bericht te verwijderen.","data":{"status":403}}
|         501 |  1745 | {"code":"rest_trash_not_supported","message":"Termen kunnen niet in de prullenbak worden geplaatst. Stel 'force=true' in om te verwijderen.","data":{"status":501}}
|         501 |   797 | {"code":"rest_trash_not_supported","message":"Gebruikers kunnen niet in de prullenbak worden geplaatst. Stel 'force=true' in om te verwijderen.","data":{"status":501}} |
|         404 |   326 | {"code":"rest_term_invalid","message":"Term bestaat niet.","data":{"status":404}}
|         404 |   229 | {"code":"rest_post_invalid_id","message":"Ongeldig bericht-ID.","data":{"status":404}}
|         403 |   164 | {"code":"rest_post_incorrect_password","message":"Wachtwoord van bericht is incorrect.","data":{"status":403}}
|         404 |   147 | {"code":"rest_post_no_autosave","message":"Er is geen autosave versie van dit bericht.","data":{"status":404}}
|         403 |   103 | {"code":"rest_comment_draft_post","message":"Je hebt geen toestemming om reacties op dit bericht te plaatsen.","data":{"status":403}}
|         400 |    88 | {"code":"rest_upload_no_content_disposition","message":"Geen Content-Disposition verstrekt.","data":{"status":400}}
|         404 |    80 | {"code":"rest_post_invalid_parent","message":"Ongeldige bericht hoofd ID.","data":{"status":404}}
|         404 |    53 | {"code":"rest_user_invalid_id","message":"Ongeldig gebruikers-ID.","data":{"status":404}}
|         500 |    44 | {"code":"existing_user_login","message":"Sorry, deze gebruikersnaam bestaat al!","data":null}
|         400 |    36 | {"code":"rest_invalid_author","message":"Ongeldig auteurs-ID.","data":{"status":400}}
|         410 |    27 | {"code":"rest_already_trashed","message":"Dit bericht is al verwijderd.","data":{"status":410}}
|         403 |     2 | {"code":"rest_comment_trash_post","message":"Je hebt geen toestemming om reacties op dit bericht te plaatsen.","data":{"status":403}}
|         403 |     1 | {"code":"rest_cannot_edit","message":"Je hebt geen toestemming om media te uploaden voor dit bericht.","data":{"status":403}}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;wckey&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;iqxre&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;tfkax&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;pjrfr&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;iiwrh&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;nrunu&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;khekp&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;caher&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;pzeyt&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;jwahh&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;xypyn&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;rgczh&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;skxai&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;npzsu&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;johrb&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;eowjn&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;akrcd&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;gqjgy&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;ejyoy&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;hhoay&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;ovnlg&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;fpwdz&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;jfqwt&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;kupnk&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;avzlj&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;skeuu&#8221; is al in gebruik door een andere term.","data":null}
|         500 |     1 | {"code":"duplicate_term_slug","message":"De slug (permalink) &#8220;ajfwt&#8221; is al in gebruik door een andere term.","data":null}
+-------------+-------+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------+
43 rows in set (2.425 sec)
```

Figure 27: Query with results from experiment with long running task with the model-based fuzzer.



```
MariaDB [rest_fuzzer]> select status_code, count(*) as count, body from fuz_responses where status_code > 200 and status_code >= 300 and project_id = 44 group by body order by count desc;
+-------------+-------+------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| status_code | count | body                                                                                                                                                             |
+-------------+-------+------------------------------------------------------------------------------------------------------------------------------------------------------------------+
|         400 |   844 | {"code":"rest_upload_no_content_disposition","message":"No Content-Disposition supplied.","data":{"status":400}}
|         403 |   772 | {"code":"rest_comment_draft_post","message":"Sorry, you are not allowed to create a comment on this post.","data":{"status":403}}
|         404 |   737 | {"code":"rest_no_route","message":"No route was found matching the URL and request method","data":{"status":404}}
|         400 |   432 | {"code":"rest_invalid_param","message":"Invalid parameter(s): context","data":{"status":400,"params":{"context":"context is not one of view, embed, edit."}}}
|         400 |   288 | {"code":"rest_invalid_param","message":"Invalid parameter(s): date","data":{"status":400,"params":{"date":"date is not of type string,null."}}}
|         400 |   286 | {"code":"rest_invalid_param","message":"Invalid parameter(s): date_gmt","data":{"status":400,"params":{"date_gmt":"date_gmt is not of type string,null."}}}
|         400 |   215 | {"code":"rest_invalid_param","message":"Invalid parameter(s): per_page","data":{"status":400,"params":{"per_page":"per_page is not of type integer."}}}
|         400 |   215 | {"code":"rest_invalid_param","message":"Invalid parameter(s): page","data":{"status":400,"params":{"page":"page is not of type integer."}}}
|         400 |   215 | {"code":"rest_invalid_param","message":"Invalid parameter(s): author","data":{"status":400,"params":{"author":"author is not of type integer."}}}
|         400 |   198 | {"code":"rest_invalid_param","message":"Invalid parameter(s): comment_status","data":{"status":400,"params":{"comment_status":"comment_status is not one of open, closed."}}}
|         400 |   197 | {"code":"rest_invalid_param","message":"Invalid parameter(s): ping_status","data":{"status":400,"params":{"ping_status":"ping_status is not one of open, closed."}}}
|         400 |   180 | {"code":"rest_invalid_param","message":"Invalid parameter(s): order","data":{"status":400,"params":{"order":"order is not one of asc, desc."}}}
|         400 |   180 | {"code":"rest_invalid_param","message":"Invalid parameter(s): featured_media","data":{"status":400,"params":{"featured_media":"featured_media is not of type integer."}}}
|         400 |   162 | {"code":"rest_invalid_param","message":"Invalid parameter(s): offset","data":{"status":400,"params":{"offset":"offset is not of type integer."}}}
|         400 |   120 | {"code":"rest_invalid_param","message":"Invalid parameter(s): exclude","data":{"status":400,"params":{"exclude":"exclude[0] is not of type integer."}}}
|         400 |   120 | {"code":"rest_invalid_param","message":"Invalid parameter(s): include","data":{"status":400,"params":{"include":"include[0] is not of type integer."}}}
|         400 |   108 | {"code":"rest_invalid_param","message":"Invalid parameter(s): sticky","data":{"status":400,"params":{"sticky":"sticky is not of type boolean."}}}
|         400 |   108 | {"code":"rest_invalid_param","message":"Invalid parameter(s): menu_order","data":{"status":400,"params":{"menu_order":"menu_order is not of type integer."}}}
|         400 |   107 | {"code":"rest_invalid_param","message":"Invalid parameter(s): force","data":{"status":400,"params":{"force":"force is not of type boolean."}}}
|         400 |    90 | {"code":"rest_invalid_param","message":"Invalid parameter(s): after","data":{"status":400,"params":{"after":"Invalid date."}}}
|         400 |    90 | {"code":"rest_invalid_param","message":"Invalid parameter(s): before","data":{"status":400,"params":{"before":"Invalid date."}}}
|         400 |    89 | {"code":"rest_invalid_param","message":"Invalid parameter(s): format","data":{"status":400,"params":{"format":"format is not one of standard, aside, chat, gallery, link, image, quote, status, video, audio."}}}
|         400 |    72 | {"code":"rest_invalid_param","message":"Invalid parameter(s): tags","data":{"status":400,"params":{"tags":"tags[0] is not of type integer."}}}
|         400 |    72 | {"code":"rest_invalid_param","message":"Invalid parameter(s): categories","data":{"status":400,"params":{"categories":"categories[0] is not of type integer."}}}
|         400 |    72 | {"code":"rest_invalid_param","message":"Invalid parameter(s): post","data":{"status":400,"params":{"post":"post is not of type integer."}}}
|         400 |    72 | {"code":"rest_invalid_param","message":"Invalid parameter(s): email","data":{"status":400,"params":{"email":"Invalid email address."}}}
|         400 |    60 | {"code":"rest_invalid_param","message":"Invalid parameter(s): exclude","data":{"status":400,"params":{"exclude":"exclude[1] is not of type integer."}}}
|         400 |    60 | {"code":"rest_invalid_param","message":"Invalid parameter(s): include","data":{"status":400,"params":{"include":"include[1] is not of type integer."}}}
|         400 |    54 | {"code":"rest_invalid_param","message":"Invalid parameter(s): orderby","data":{"status":400,"params":{"orderby":"orderby is not one of author, date, id, include, modified, parent, relevance, slug, include_slugs, title."}}}
|         403 |    54 | {"code":"rest_post_incorrect_password","message":"Incorrect post password.","data":{"status":403}}
|         400 |    54 | {"code":"rest_invalid_param","message":"Invalid parameter(s): start_of_week","data":{"status":400,"params":{"start_of_week":"start_of_week is not of type integer."}}}
|         400 |    54 | {"code":"rest_invalid_param","message":"Invalid parameter(s): use_smilies","data":{"status":400,"params":{"use_smilies":"use_smilies is not of type boolean."}}}
|         400 |    54 | {"code":"rest_invalid_param","message":"Invalid parameter(s): default_category","data":{"status":400,"params":{"default_category":"default_category is not of type integer."}}}
|         400 |    54 | {"code":"rest_invalid_param","message":"Invalid parameter(s): posts_per_page","data":{"status":400,"params":{"posts_per_page":"posts_per_page is not of type integer."}}}
|         400 |    54 | {"code":"rest_invalid_param","message":"Invalid parameter(s): default_ping_status","data":{"status":400,"params":{"default_ping_status":"default_ping_status is not one of open, closed."}}}
|         400 |    54 | {"code":"rest_invalid_param","message":"Invalid parameter(s): default_comment_status","data":{"status":400,"params":{"default_comment_status":"default_comment_status is not one of open, closed."}}}
|         400 |    48 | {"code":"rest_invalid_param","message":"Invalid parameter(s): author","data":{"status":400,"params":{"author":"author[0] is not of type integer."}}}
|         400 |    48 | {"code":"rest_invalid_param","message":"Invalid parameter(s): author_exclude","data":{"status":400,"params":{"author_exclude":"author_exclude[0] is not of type integer."}}}
|         400 |    36 | {"code":"rest_invalid_param","message":"Invalid parameter(s): categories","data":{"status":400,"params":{"categories":"categories[1] is not of type integer."}}}
|         400 |    36 | {"code":"rest_invalid_param","message":"Invalid parameter(s): orderby","data":{"status":400,"params":{"orderby":"orderby is not one of date, id, include, relevance, slug, include_slugs, title."}}}
|         400 |    36 | {"code":"rest_invalid_param","message":"Invalid parameter(s): parent_exclude","data":{"status":400,"params":{"parent_exclude":"parent_exclude[0] is not of type integer."}}}
|         400 |    36 | {"code":"rest_invalid_param","message":"Invalid parameter(s): hide_empty","data":{"status":400,"params":{"hide_empty":"hide_empty is not of type boolean."}}}
|         400 |    36 | {"code":"rest_invalid_param","message":"Invalid parameter(s): orderby","data":{"status":400,"params":{"orderby":"orderby is not one of id, include, name, slug, include_slugs, term_group, description, count."}}}
|         400 |    36 | {"code":"rest_invalid_param","message":"Invalid parameter(s): author_email","data":{"status":400,"params":{"author_email":"Invalid email address."}}}
|         400 |    35 | {"code":"rest_invalid_param","message":"Invalid parameter(s): tags","data":{"status":400,"params":{"tags":"tags[1] is not of type integer."}}}
|         400 |    24 | {"code":"rest_invalid_param","message":"Invalid parameter(s): author","data":{"status":400,"params":{"author":"author[1] is not of type integer."}}}
|         400 |    24 | {"code":"rest_invalid_param","message":"Invalid parameter(s): author_exclude","data":{"status":400,"params":{"author_exclude":"author_exclude[1] is not of type integer."}}}
|         500 |    24 | {"code":"duplicate_term_slug","message":"The slug &#8220;1or11&#8221; is already in use by another term.","data":null}
|         500 |    24 | {"code":"duplicate_term_slug","message":"The slug &#8220;1-or-11&#8221; is already in use by another term.","data":null}
|         500 |    19 | {"code":"existing_user_login","message":"Sorry, that username already exists!","data":null}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): tax_relation","data":{"status":400,"params":{"tax_relation":"tax_relation is not one of AND, OR."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): orderby","data":{"status":400,"params":{"orderby":"orderby is not one of author, date, id, include, modified, parent, relevance, slug, include_slugs, title, menu_order."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): parent_exclude","data":{"status":400,"params":{"parent_exclude":"parent_exclude[1] is not of type integer."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): media_type","data":{"status":400,"params":{"media_type":"media_type is not one of image, video, text, application, audio."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): orderby","data":{"status":400,"params":{"orderby":"orderby is not one of id, include, name, registered_date, slug, include_slugs, email, url."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): who","data":{"status":400,"params":{"who":"who is not one of authors."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): locale","data":{"status":400,"params":{"locale":"locale is not one of , en_US, nl_NL."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): username","data":{"status":400,"params":{"username":"Username contains invalid characters."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): reassign","data":{"status":400,"params":{"reassign":"Invalid user parameter(s)."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): orderby","data":{"status":400,"params":{"orderby":"orderby is not one of date, date_gmt, id, include, post, parent, type."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): date","data":{"status":400,"params":{"date":"Invalid date."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): date_gmt","data":{"status":400,"params":{"date_gmt":"Invalid date."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): context","data":{"status":400,"params":{"context":"context is not one of view, embed."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): type","data":{"status":400,"params":{"type":"type is not one of post."}}}
|         400 |    18 | {"code":"rest_invalid_param","message":"Invalid parameter(s): status","data":{"status":400,"params":{"status":"status[0] is not one of active."}}}
|         400 |    17 | {"code":"rest_invalid_param","message":"Invalid parameter(s): author_ip","data":{"status":400,"params":{"author_ip":"author_ip is not a valid IP address."}}}
|         400 |    12 | {"code":"rest_invalid_param","message":"Invalid parameter(s): categories_exclude","data":{"status":400,"params":{"categories_exclude":"categories_exclude[0] is not of type integer."}}}
|         400 |    12 | {"code":"rest_invalid_param","message":"Invalid parameter(s): tags_exclude","data":{"status":400,"params":{"tags_exclude":"tags_exclude[0] is not of type integer."}}}
|         500 |    12 | {"code":"duplicate_term_slug","message":"The slug &#8220;1-and-1select-count-from-tablenames&#8221; is already in use by another term.","data":null}
|         500 |    12 | {"code":"duplicate_term_slug","message":"The slug &#8220;11&#8221; is already in use by another term.","data":null}
|         500 |    12 | {"code":"duplicate_term_slug","message":"The slug &#8220;1-exec-sp_-or-exec-xp_&#8221; is already in use by another term.","data":null}
|         500 |    12 | {"code":"duplicate_term_slug","message":"The slug &#8220;1-and-11&#8221; is already in use by another term.","data":null}
|         500 |    12 | {"code":"duplicate_term_slug","message":"The slug &#8220;fakeemaoril-nlil-nl&#8221; is already in use by another term.","data":null}
|         400 |    12 | {"code":"rest_invalid_param","message":"Invalid parameter(s): post","data":{"status":400,"params":{"post":"post[0] is not of type integer."}}}
|         400 |     6 | {"code":"rest_invalid_param","message":"Invalid parameter(s): categories_exclude","data":{"status":400,"params":{"categories_exclude":"categories_exclude[1] is not of type integer."}}}
|         400 |     6 | {"code":"rest_invalid_param","message":"Invalid parameter(s): tags_exclude","data":{"status":400,"params":{"tags_exclude":"tags_exclude[1] is not of type integer."}}}
|         400 |     6 | {"code":"rest_invalid_param","message":"Invalid parameter(s): post","data":{"status":400,"params":{"post":"post[1] is not of type integer."}}}
|         500 |     1 | {"code":"duplicate_term_slug","message":"The slug &#8220;cythb&#8221; is already in use by another term.","data":null}
+-------------+-------+------------------------------------------------------------------------------------------------------------------------------------------------------------------+
78 rows in set (0.173 sec)
```

Figure 28: Query with results from experiment with long running task with the model-based dictionary fuzzer.



Identifier
6

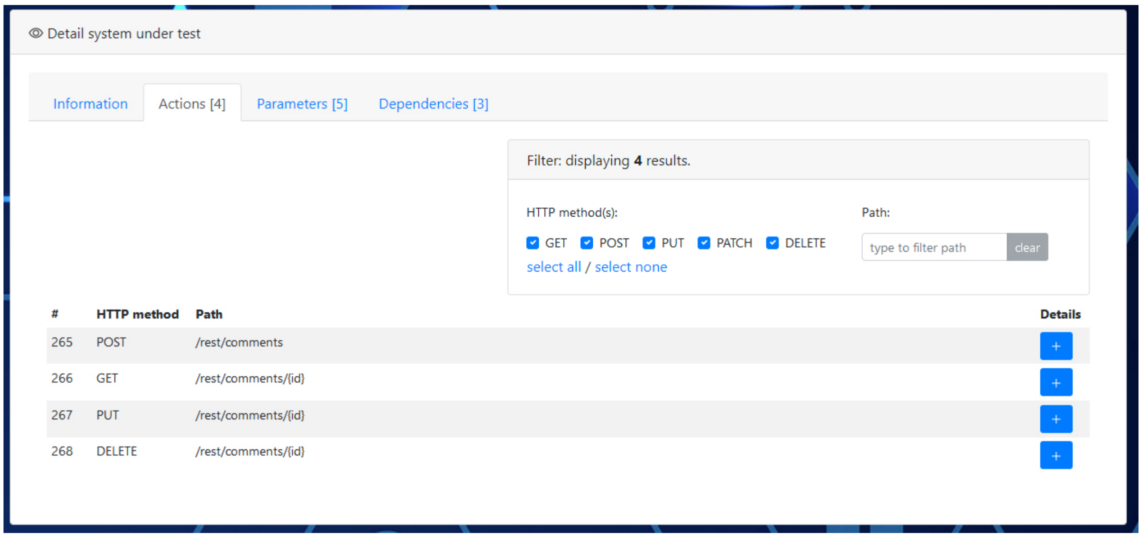
Description
This SUT is a REST web service containing SQL injection vulnerabilities.

Title
SUT - proof of concept - SQL injection

OAS location
http://localhost:7777/v2/api-docs

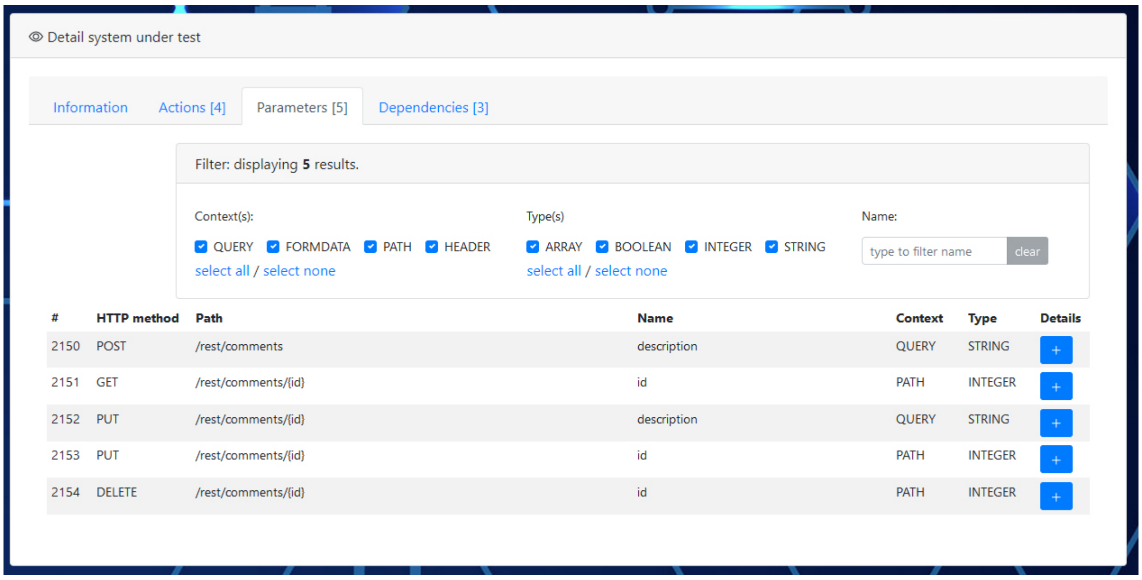
Created @
20-09-2020 14:40:05

Figure 29: Overview of general information extracted from OpenAPI specification from system under test SutSqlI.



#	HTTP method	Path	Details
265	POST	/rest/comments	+
266	GET	/rest/comments/{id}	+
267	PUT	/rest/comments/{id}	+
268	DELETE	/rest/comments/{id}	+

Figure 30: Overview of actions extracted from OpenAPI specification from system under test SutSqlI.



Filter: displaying **5** results.

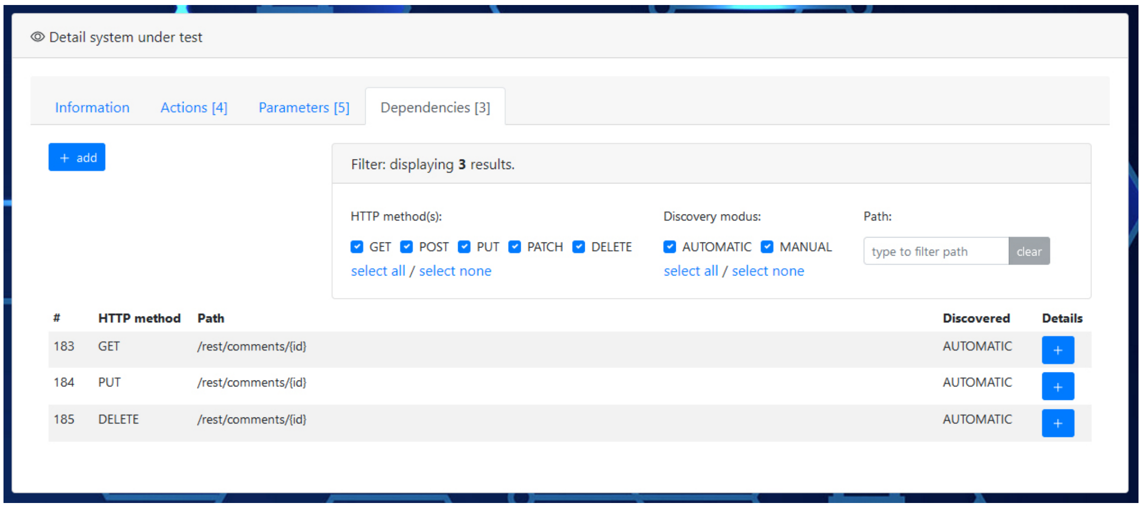
Context(s): ✓ QUERY ✓ FORMDATA ✓ PATH ✓ HEADER
select all / select none

Type(s): ✓ ARRAY ✓ BOOLEAN ✓ INTEGER ✓ STRING
select all / select none

Name: type to filter name clear

#	HTTP method	Path	Name	Context	Type	Details
2150	POST	/rest/comments	description	QUERY	STRING	+
2151	GET	/rest/comments/{id}	id	PATH	INTEGER	+
2152	PUT	/rest/comments/{id}	description	QUERY	STRING	+
2153	PUT	/rest/comments/{id}	id	PATH	INTEGER	+
2154	DELETE	/rest/comments/{id}	id	PATH	INTEGER	+

Figure 31: Overview of the parameters extracted from OpenAPI specification from system under test SutSqlI.



Filter: displaying **3** results.

HTTP method(s): ✓ GET ✓ POST ✓ PUT ✓ PATCH ✓ DELETE
select all / select none

Discovery modus: ✓ AUTOMATIC ✓ MANUAL
select all / select none

Path: type to filter path clear

#	HTTP method	Path	Discovered	Details
183	GET	/rest/comments/{id}	AUTOMATIC	+
184	PUT	/rest/comments/{id}	AUTOMATIC	+
185	DELETE	/rest/comments/{id}	AUTOMATIC	+

Figure 32: Overview of the dependencies extracted from OpenAPI specification from system under test SutSqlI.