university of groningen

faculty of science and engineering

Finding significant vulnerabilities in complex web applications using Fuzzing

Rowan van Beckhoven

university of groningen

faculty of science and engineering

University of Groningen

Finding significant vulnerabilities in complex web applications using Fuzzing

Master's Thesis

To fulfil the requirements for the degree of
Master of Science in Computing Science (Software Engineering & Distributed Systems)
at the University of Groningen under the supervision of
dr. Fatih Turkmen (Computer Science, University of Groningen),
dr. A. Rastogi (Computer Science, University of Groningen) and
Gerben Broenink (TNO)

Rowan van Beckhoven (s2536730)

June 20, 2022

1 Abstract

Web applications are a large part of the internet infrastructure nowadays. By exposing these applications to the public, a vulnerability within these applications could cause harmful situations. To prevent vulnerabilities from being introduced in applications, tools like SAST & DAST tools exist, which provide reasonable insight into some of the vulnerabilities of an application. However, these tools lack at finding unknown patterns of vulnerabilities. Fuzzing is a testing approach that has proven its usefulness in discovering unknown software vulnerabilities. While the area of fuzzing binary applications is common in academic research, fuzzing web applications is still uncommon. In this research, methods of existing works are combined to create a fuzzer that is capable of finding vulnerabilities in web applications. This fuzzer, GRFUZZ, uses a stateful approach in combination with grey-box fuzzing methods. Additionally, an optimization is introduced which uses feedback of an application to guide GRFUZZ towards making better choices in future iterations. In a case study, GRFUZZ is compared to RESTler, which shows that GRFUZZ outperforms RESTler by finding more bugs, and higher coverage, while also being able to detect more significant types of vulnerabilities.

Contents

Acronyms

API Application Programming Interface.

CVE Common Vulnerabilities & Exposures.
CWE Common Weakness Enumeration.

DAST Dynamic Application Security Testing.

HTTP HyperText Transfer Protocol.

IAST Interactive Application Security Testing.
IDOR Insecure Direct Object Reference.

JaCoCo Java Code Coverage.
JNI Java Native Interface.
JSON JavaScript Object Notation.
JVM Java Virtual Machine.

NVD National Vulnerability Database.

ORM Object-Relational Mapping.

PUT Program Under Test.

REST REpresentational State Transfer.
RFC Request For Comments.

SAST Static Application Security Testing.
SDLC Software Development Life-Cycle.
SMT Satisfiability Modulo Theories.
SQL Simple Query Language.
SQLi Simple Query Language (SQL) injection.
SSL Secure Socket Layer.
SSRF Server-Side Request Forgery.

TCP Transmission Control Protocol.
TLS Transport Layer Security.

URI Uniform Resource Identifier.

XSS Cross-Site Scripting.

2 Introduction

Since Web 2.0, the web has evolved where traditional services have been replaced by digital, web services; from the automotive industry to healthcare, to public transport. Anything that can be provided as an online service, will be provided as an online service. While this is great for the general user experience, this also comes with grave danger. The ecosystem of online services is still rather young, where the infrastructure of the web leaves all the room for engineers to come up with their own systems. These systems are then exposed to the internet and therefore impose a major threat if any security flaw is overlooked. Since 2003, the OWASP Foundation [1] has been working on improving the security of software. By raising awareness and providing other tools, they target engineers and other involved parties to help them secure the web. A key resource is the OWASP Top 10[2], which contains the most significant, impactful and commonly seen vulnerabilities in web applications.

In the general Software Development Life-Cycle (SDLC), security is most often evaluated during the testing phase. Because of the shift-left approach in Continuous Integration/Continuous Delivery (CI/CD), software testing is done earlier in the SDLC and more often, thus automated testing is highly preferred here. In the area of DevOps, the term *DevSecOps*[3] is tossed to instruct a collaborative mindset into the general software engineer, or software engineering team. Security testing forms a major bottleneck here, since automatic security testing is still extremely limited. Ranging from static code analysis, to scanning public libraries for known Common Vulnerabilities & Exposures (CVE)s [4] [5], these methods provide limited insights in the quality of security of an application. To overcome, or compensate these limitations, thorough security testing is often done less regularly, but more intensive. One form of such thorough security testing is penetration-testing (pen-testing), which uses a combination of automated tooling, e.g. Burp Suite or OWASP ZAP and manual labour.

Besides these techniques, another advanced security testing technique is used; fuzzing, which has been around since 1989, introduced by Prof. Barton Miller [6]. Fuzzing is most popular in the software testing process for binary applications and software libraries. However, in the area of web application security testing, this technique is fairly unknown. Although some academic research has been performed in this area [7],[8], publicly available security testing tools provide no to extremely limited fuzzing capabilities [9], [10]. Reasons for this could be that there is a different mindset in these two areas, or that these areas are inherently different, which impacts the usefulness of existing methods when applied to web applications. However, from an abstract level, both types of applications can be seen as black-boxes that process inputs into outputs. Such applications provide entry-points or an Application Programming Interface (API) which can be seen as the trust-boundary of these applications. Whenever some input crosses this boundary, the application should be very considerate of this input. Such input could potentially reach vulnerable parts of the system, which is an obvious security hazard. What is considered to be vulnerable, depends on the environmental constraints of a system, such as which language it is written in, which operating system it uses, and many more constraints. These constraints identify where to look for vulnerabilities and how to trigger them. By going more in-depth, the two areas of applications start to diverge.

Considering the most significant vulnerabilities, provided by OWASP, we would like to be able to find these vulnerabilities in web applications through fuzzing. Specifically in web applications that expose a REpresentational State Transfer (REST) API [11]. With this goal in mind, this research aims to address the following questions:

- *Can significant vulnerabilities of a REST API be exposed by fuzzing?*

- *Which traditional fuzzing techniques can be applied in the context of testing web applications?*

Following the first question, a sub-question arises regarding the complexity imposed by a typical REST architecture:

- *When taking into account the complexity of web applications, is it possible to find more, significant vulnerabilities?*

In this research, the tool GRFUZZ (Grey-box REST Fuzzer) is developed to present answers to these questions. By combining an existing fuzzer with other fuzzing techniques from existing research fields, a more effective fuzzer is created. Using a case study on an existing, real-world application, we show that this fuzzer outperforms comparable fuzzers in multiple aspects.

2.1 Structure of this thesis

Following the introduction to this topic, the background and related work to this research area are described in Chapter 3 and Chapter 4. In Chapter 5, the approach is described which elaborates on the methods used and how they interact with each other. Next up, in Chapter 6, a high-level architecture is presented, accompanied by more concrete implementation details. To validate this implementation, a small testing setup is described in Chapter 7 after which the implementation is subdued to a case study in Chapter 8. Results of the case study and implementation are discussed in Chapter 9, after which ideas for future work are presented in Chapter 10. Finally, the work is concluded in Chapter 11.

3 Background

3.1 Fuzzing

Fuzzing is a term used in software testing, which describes a testing process in which a Program Under Test (PUT) is subdued to a large input space. Such an input space is generated by the fuzzer, specifically with the goal to create an input space that surpasses the input space of inputs that are expected by the PUT. By creating these pseudo-random inputs, the fuzzer attempts to trigger unintended behaviour in the PUT. Provided with the information of *what* is sent to the PUT, unintended behaviour can be explained, which allows a tester or developer to fix this unintended behaviour. This technique can be applied to find security-related bugs or other classes of bugs that are defined by constraints on correctness.

3.1.1 Black-box

Fuzzers are generally classified as *black-box* when they do not make use of any of the internals of the PUT [12]. The only observations are made on the input and output behaviour of the PUT. Within black-box fuzzers, different techniques can be used to generate inputs for the PUT. For example, inputs can be generated from scratch or can be evolved using a mutation-based approach. In the latter, seeds are scheduled for mutation and these seeds evolve. Generation-based mutations often rely on some definition of a grammar to be able to generate values that are accepted by the PUT. Examples of these generation-based fuzzers are Peach [13], FunFuzz [14] and CATs [15].

3.1.2 White-box

White-box fuzzers on the other hand, have full access to the internals of the PUT. Introduced by Godefroid [16], this technique uses heavyweight analysis of the PUT to achieve better performance. Symbolic execution is a prime example of such an analysis technique which uses information about branching statements inside the PUT to generate inputs that match the conditionals within these statements. By doing so, an input space can be generated that achieves full program coverage. Other well-known methods within white-box fuzzing are *Concolic fuzzing* and *Dynamic taint analysis*.

Concolic fuzzing
Concolic fuzzing makes use of execution data on which conditionals are encountered during a function call. A concolic fuzzer collects information about the input parameters that are used to evaluate conditions that lead to different branches in the code. Using this feedback, a fuzzer can mutate the input to trigger other paths that can be led by this conditional. This technique is similar to symbolic execution, with the exception that it collects symbols during program execution [17]. In order to pass branching conditions, a common approach is to use a constraint solver, such as an Satisfiability Modulo Theories (SMT) solver, e.g. Z3 [18]. These solvers are designed to create inputs that solve first-order logic conditionals, enhanced by being able to reason about primitives such as integers and strings. For example, given a small piece of code:

```
void main(char[] args)
{
       if (args[0] == 'b')
              if (args[1] == 'a')
                     if (args[2] == 'd')
```

```
                    if (args[3] == '!')
                        error();
}
```

when the input equals *bad!*, an error is triggered. When given the input *good*, the first conditional is not satisfied. This allows an SMT solver to generate a value that solves this conditional. In this case, the conditional *args[0] == 'b'* is trivial and only solvable where *args[0]* is *b*, but more complex conditionals can also be solved by such solvers. This results in a new input generated in the next iteration, which contains the value *b*. By passing the new input *bood*, the next conditional is reached, and so on.

Dynamic taint analysis

Dynamic taint analysis, or dynamic taint propagation is a technique in which the user-supplied input is associated with a taint. The taint value is used to mark a value as trusted or untrusted. This taint is then propagated through the program during runtime which allows for certain constructs for checking whether executing behaviour using this value is safe or not. The part where behaviour is executed is also referred to as a *sink*. For example, when a web application receives user input that is used to create a database query, the application needs to ensure that this does not trigger a SQL injection (SQLi). To do so, a process called sanitization can be used, which ensures that the value can be safely used. This is a conceptual process which is extremely use-case specific. Nevertheless, this is generalized to a concept which has user input that is considered unsafe, a sanitizer that makes this data safe and a sink that executes behaviour using this data.
When fuzzing for vulnerabilities, many origins of vulnerabilities are caused by unsafe handling of user-defined input [2]. In order to use this technique, two questions need to be answered:

- *Which parts of an application should be considered a sink?*
- *When can input data be considered safe, or what is the definition of a sanitizer?*

Taking the example of a SQLi, a sink would be the Object-Relational Mapping (ORM) or database driver that executes a query. The challenge here is to define a general construct that validates this input data to ensure safety. This challenge is also reflected in the findings of [19]. Their findings contained false positives due to the inability to accurately determine what a sanitizer should be. Input validation that leads to mitigation of certain vulnerabilities can be highly specific to the application logic and using this would be impractical. What is left is a construct that is able to verify whether user-supplied input is able to reach a sink in the application. This still is useful information which can be used to identify the potentially vulnerable parts of the application.
Besides these aforementioned challenges, another defect of dynamic taint analysis is that it is only able to trace values that are directly supplied by a user. However, not all unsafe data directly originates from user input. However, consider the case where a user is able to supply a value which is harmless when stored in a database, but when this database entry is used in another piece of code, this value exhibits dangerous behaviour. For example, when a URL is persisted in the database in a first request, without being used to execute any new request, this is still harmless. However, when a second request queries this data and retrieves this URL to execute another request, the application should be certain that the URL can be trusted.
In web applications, generally, the domain consists of many coupled entities which are connected to implement complex domain logic. In order to find these scenarios, we need to reconsider some of the definitions of dynamic taint analysis. User input can also be indirectly used, which would mean

that taints need to be propagated beyond code at runtime. When a stored value is retrieved from a database that can be controlled by a user, this value should again be considered as untrusted. However, applying this concept could be impractical. Another approach would be to discard the knowledge of whether a value is tainted or untainted, thus generalizing from indirect and direct input. By finding a sink in the application, specific input can be crafted which could validate that the user input reaches this sink. This can be taken one step further, where the user input is crafted in such a way that it contains values that could trigger a vulnerability in that sink. The downside of this approach however is that it is unknown whether a value, used by the sink, is tainted. This highly reduces the effectiveness of dynamic taint analysis, with the benefit of being generic.

3.1.3 Grey-box

Combining both white- and black-box techniques, grey-box fuzzing is capable of retrieving some information about the PUT. A common approach within grey-box fuzzing is *coverage-based* grey-box fuzzing or CGF. In CGF, code coverage is retrieved during the execution of a seed, which represents the interestingness of the execution. By combining this with an evolutionary algorithm, mutations of seeds retrieve a fitness score, which allows the fuzzer to make decisions about how to schedule these. Code coverage is retrieved by using lightweight code instrumentation. Besides coverage, instrumentation can also be used to retrieve other runtime information. For example, VUzzer [20] additionally uses control-flow analysis in order to reason about where and how to mutate inputs effectively. This helps in finding more bugs, more efficiently than a basic coverage-guided fuzzer, such as AFL [21]. LibFuzzer [22] uses similar techniques as VUzzer, where comparison (CMP) instructions are used to guide mutations based on the arguments of these comparison instructions.

While black-box methods often suffer from receiving limited feedback about the execution on the PUT, white-box methods often impose high overhead due to heavyweight program analysis or constraint solvers. Grey-box fuzzing is a middle ground which is capable of reasoning about the application to a certain degree, without having high overhead.

3.2 REST

REST[11] is an architectural style that is specifically purposed for the transfer of state. A web application that conforms to the principles of REST is also referred to as *RESTful*. In web applications, the concept of *state* is used to define how a service manages state: an application can either be *stateful* or *stateless*. The definition of REST states that a RESTful web application should be stateless.

> *each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.*

A stateful web application is a web application that does rely on a stored context on the server. An example of this is an application that stores client authentication data in a session context on the server. Whenever a client authenticates to the server, some state is stored which is used in following requests to determine whether this client is properly authenticated. If this context does not exist on the server, it will not consider the client to be authenticated. Stateful web applications for that reason are more difficult to scale horizontally.

In a RESTful architecture, state needs to be managed by the stateless web application. This does not mean that making use of a RESTful web application does not affect any state. Most often, a web application makes use of some database implementation to store this state. In the rest of this thesis, state refers to the resources created using the API of the web application.

3.2.1 Resource operations

To create or modify state, REST guidelines define a set of HTTP methods. These methods can be used in combination with a resource path and an optional resource identifier. These requests will be interpreted differently based on how this URL is constructed. For example:

GET /books/id
GET /books

Note that although a lot of definitions in REST are widely agreed upon, the definitions are not strict; No Request For Comments (RFC) has been written, thus the implementation of the architectural style is open for interpretation. The methods that a RESTful architecture typically uses are (but not limited to):

- GET
- POST
- PUT
- PATCH
- DELETE

GET
The GET method is strictly used to read existing data. This method can either be used with a resource identifier, retrieving a single resource or without, retrieving a set of resources.

POST
The POST method is used to create a new resource. This method does not require a resource identifier. When accepted by a server, the server should respond with at least some information about where to find this newly created resource. This can either be encoded in the *Location* header, or encoded in the response body.

PUT
The PUT method is used to modify an existing resource. This method does require a resource identifier. Some implementations also create a new resource whenever no resource exists with the provided identifier.

PATCH
The PATCH method is used to partially modify an existing resource. This method does require a resource identifier. This method is not always implemented since all its functionality can be imposed by the PUT method. In some cases, it is practical to only send the subset of changed parameters. For example, when dealing with large entities.

DELETE
The DELETE method is used to delete an existing resource. This method does require a resource identifier.

3.2.2 Resource dependencies

In typical domain models, entities are modelled using a relationship model. Entities are made up of sub-entities and can belong to a parent entity, these typically are *has-a* relationships. These relations, or dependencies, are also reflected in how state is represented in REST. For example, the request

GET /authors/123/books

could be interpreted to retrieve all the books that belong to the specific author *123*. In other words, an Author entity has a relation to books and vice-versa. Depending on the implementation, this also requires the client to specify this relation when creating the resource. In this case, the assumption is that an author can have many books, but a book only can have one author. Two options (not mutually exclusive) remain here:

1. Supply a reference to the Author when creating a Book
2. Supply all references to the Books when creating an Author

Either option will require a reference to another entity. This reference can only be obtained by creating this entity beforehand, thus imposing a dependency on another request.

Explicit dependencies
Typically, relationship dependencies are explicitly defined in a request: the reference to the dependent resource is sent along with the request. Other types of dependencies that are sent along with the request as either a parameter, header or value in the HyperText Transfer Protocol (HTTP) message body, can be considered *explicit* dependencies. In other words, when the output(s) from another request are required inputs for another request, there is an explicit dependency. For example, in the OAuth2 protocol, when a client authenticates, it receives some form of an authentication token, which can be used to send to a server in order to authorize a request. A common approach here is to attach the token to the *Authorization* header, specifying a *Bearer* token as a value.

Implicit dependencies
Implicit dependencies on the other hand are not defined as inputs or outputs from the API level. These dependencies are derived from some state transition system. Consider the case of a stateful web application that handles authentication;

1. The service first should set up a session with the client (stored on the server)
2. The client then authenticates to this service
3. The context in this session contains some authentication of this client. For example, some flag that informs about the user being authenticated or not
4. In proceeding requests, this context will be used to determine the authenticated state of this client

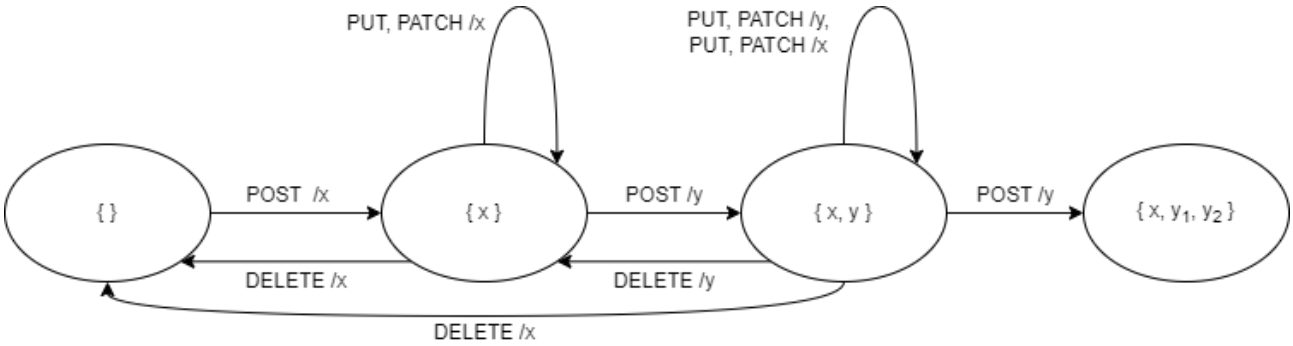


Figure 1: REST State diagram with dependencies

Without the first three steps, a request to the server will never know whether a client has been authenticated. However, in a stateful application, a client does not need to provide this authentication to the server after these first three steps, since this is stored as state on the server. This state can implicitly be used by the server in proceeding requests. Such a case is considered to be an *implicit* dependency on the requests that lead to this state. A general definition of an implicit dependency is a dependency which is not provided by the API client, but rather by the server.

The main difference between implicit and explicit dependencies is that implicit dependencies are fully determined by the server, whereas explicit dependencies are provided by the client.

3.2.3 State transitions

State transitions can be considered at different levels. Domain entities can have certain domain logic that defines a set of transition rules. Based on these rules, the state of a domain entity can only transition to another state, if it conforms to these rules. For example, a checking account only allows for withdrawals when the account balance is positive. These domain level state transitions are highly application-specific.
A higher-level approach in state transitions, in the context of REST, is to consider resources as a single entity, with one state. This state can either be created, updated or deleted. Each resource can have in- and outbound dependencies on other resources, which results in the following state transition diagram in Figure 1. Note that resource type y can only be created after the creation of resource type x. The internal state of a resource is abstracted here, so modifying a resource does not affect the global state. Dependencies can be both implicit and explicit. Finally, the transition from $\{x, y\}$ to $\{\}$ does not necessarily have to be implemented; depending on the implementation, a web application could automatically delete child resources when a parent resource is deleted. Using either database logic, using CASCADE DELETE operations, or application-level logic, such a transition can be implemented.

3.3 Vulnerabilities

The goal of Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST) and Interactive Application Security Testing (IAST) tools is to find bugs and vulnerabilities before the application is deployed to production. These classes of tools have intrinsic differences, which make them more or less viable for detecting certain classes of vulnerabilities.

SAST
Static Application Security Testing examines the source code to detect vulnerabilities before runtime. This can be done at the early stages of the application. Since this has access to the source code, it is able to determine which exact parts of the code are vulnerable. However, this technique cannot be used to detect vulnerabilities that only occur during runtime, such as authentication and authorization vulnerabilities.

DAST
Dynamic Application Security Testing allows finding vulnerabilities during runtime of the application. This is often done in a black-box setting and therefore does not require any access to the source code. However, it is also more difficult to trace which part of the code is responsible for a found vulnerability.

IAST
Thirdly, Interactive Application Security Testing mostly relies on code instrumentation to find vulnerabilities. This allows for determining the exact location of the source of a vulnerability. The only deficits are that it has runtime overhead because of the instrumentation and not many commercial implementations exist.

In short, static and dynamic code analysis techniques have their advantages and disadvantages. One way to determine which classes of vulnerabilities benefit most from using specific techniques, is to look at the OWASP top 10 vulnerabilities. These vulnerabilities are classified in the following paragraphs as to whether static or dynamic analysis is required, and if dynamic analysis is required, whether it requires IAST to provide more insights than a black-box DAST setting. Note that the assumption here is that the capabilities of an IAST method is a superset of the capabilities of a DAST method. These evaluations are made by combining existing overviews [23] with OWASP Top 10 descriptions and experience in the field.

1. **Broken Access Control**
   Access control violations can only be evaluated during runtime, which requires at least a dynamic approach. A black-box approach can detect violations, such as an Insecure Direct Object Reference (IDOR), or path-traversal vulnerabilities. This makes this category suitable for DAST tools.

2. **Cryptographic Failures**
   Cryptographic failures can be detected by both static and dynamic methods. Some wrongly used cryptographic methods can be detected by source-code analysis, such as the use of outdated cipher-suites, the embedding of secure keys in source control, or the use of weak or deprecated cryptography libraries. However, runtime analysis is required for the detection of invalid or missing cryptography methods such as the enforcement of encryption in Secure Socket Layer (SSL)/Transport Layer Security (TLS) or the valid use of server certificates. This makes this category suitable for both SAST and DAST.

3. **Injection**
   Injection is mostly only detectable by dynamic methods. These vulnerabilities occur when the input of a user passes a trust boundary, such as the server API and this boundary does not

properly validate this input. When the underlying logic is susceptible to injection attacks, this may or may not expose any additional information. Although most injection detection methods rely on black-box methods, it is untrue that all injection attacks are visible in the response of a server. Relying on a server providing this information is in most cases a vulnerability itself [24]. In this situation, IAST tools could provide more insight than a black-box DAST method.

4. **Insecure Design**
   This is a broad category that includes many Common Weakness Enumeration (CWE)s. Most of these CWE are related to design choices that affect the application at runtime, such as using validation frameworks for input validation, making them most suitable for DAST and IAST tools. However, some of these CWEs are highly domain-specific and require either manual code inspection or the capability of defined custom rule-sets for detection patterns.

5. **Security Misconfiguration**
   This category has many overlaps with Insecure Design, however, it is scoped down to improper configuration. Configuration is mostly determined before runtime which allows the use of SAST tools to scan for common patterns that point at the presence of a vulnerability.

6. **Vulnerable and Outdated Components**
   Most software is built from components or packages, of which many of these dependencies are 3rd-party. These components themselves can be exploitable, which makes the user of these components vulnerable as well. This knowledge is mostly retrieved from sources such as the National Vulnerability Database (NVD). By comparing the used versions with the known vulnerable versions, static analysis can determine whether a piece of software contains a vulnerable dependency.

7. **Identification and Authentication Failures**
   Identification and Authentication Failures revolve around vulnerabilities that occur when authentication is weakly implemented such as the lack of Multi-factor authentication, permitting brute-force attacks or allowing the use of weak passwords. These methods can only be detected at runtime and are therefore most suitable for DAST methods.

8. **Software and Data Integrity Failures**
   Similar to using vulnerable and outdated components, data integrity failures are mostly vulnerabilities that are caused by the lack of establishing trust with a 3rd party. For example, using a 3rd party repository, without validating the source using signatures or hashes. Or deserialization of untrusted data, which could expose the system to remote code execution. Some of these vulnerabilities can be detected using static code-analysis, whereas others require a dynamic approach.

9. **Security Logging and Monitoring Failures**
   This class of vulnerabilities is strictly detectable by IAST methods. These are caused by insufficient forms of monitoring of security-related events, such as user authentication. Although some of these are detectable by IAST methods, this category is mostly detectable by manual audits. Proper security protocols such as the storage of log files can not be validated by tools that inspect the application. Mostly because logging and monitoring are often delegated to a 3rd-party tool.

10. **Server-Side Request Forgery (SSRF)**
    This vulnerability is exploitable when a server does not properly sanitize user input. Specifically

when this user input contains Uniform Resource Identifier (URI)s that are used by the server to retrieve additional data. When this URI is not properly sanitized, the user can control the request executed by the server, which potentially exposes or even exploits the internal (private) network of the system. Although this exploit is most interesting when it exposes information, some exploits do not provide any additional insights, but rather abuse the internal system. The latter is only detectable using IAST methods.

Analysing the OWASP top 10 with regard to Application Security Testing methods, results in Table 1:

Table 1: OWASP Top 10 vulnerabilities, with potential detection methods

	SAST	DAST	IAST
Broken Access Control		x	
Cryptographic Failures	x	x	
Injection		x	x
Insecure Design		x	x
Security Misconfiguration	x		
Vulnerable and Outdated Components	x		
Identification and Authentication Failures		x	
Software and Data Integrity Failures	x	x	
Security Logging and Monitoring Failures			x
Server-side Request Forgery (SSRF)		x	x

3.4 Fuzzing for vulnerabilities

Using the knowledge of what kind of tools are most applicable for which classes of vulnerabilities, it can be seen that dynamic security testing tools are most widely applicable for detecting vulnerabilities. On an abstract level, a fuzzer can be classified as a dynamic security testing tool. Following that, a grey-box fuzzer is similar to an IAST tool, since it also has the capabilities of using code instrumentation to collect runtime information about the application. Although not many classes of vulnerabilities rely on having this runtime information to detect these classes, in some cases this additional information is helpful. Since there is a trade-off in using grey-box fuzzing methods, compared to black-box, it should be carefully evaluated whether using a grey-box method provides any benefits in achieving a certain goal.

Within this research, only SSRF is focused on to provide insight in how grey-box methods could be used to find such types of vulnerabilities.

4 Related work

Atlidakis et al. [8] is the first paper, to our knowledge, on fuzzing for web applications to take into consideration the stateful properties of a REST API. In their approach, a black-box fuzzer is used which is initialized by parsing an OpenAPI specification. By parsing this specification, RESTler is able to infer dependencies among different requests which are used to construct a test case. These test cases are represented by a sequence of requests in which all dependencies are fulfilled by previous requests in the sequence. They apply their implementation to numerous cloud services, which expose a REST API to the public. The use of REST is not limited to cloud services, but it is more predominant in that area of web services. Based on the design principles of REST, it is easy to see that certain endpoints of an API can only be called by using parameters that have been dynamically created during a previous request. This can be within a single resource endpoint, where for example the deletion of a resource depends on creating this resource beforehand or spanning multiple resources, where resources are related to each other.
In the paper of RESTler, it becomes clear that fuzzing a simple Blog posts service already benefits from using a stateful approach. By creating test cases that span two or more requests, the coverage during execution increases. The length of this sequence is directly related to the complexity of the web application; the more nested dependencies a request has, the longer this sequence needs to be, to be able to fulfil all dependencies.

RESTler uses the term *stateful* to define its approach in sending request sequences to a web service. They explicitly define sequences of requests that are executed in order. This however does not mean that these requests can only be executed on stateful web services. This simply implies that the requests in these sequences are dependent on the previous requests. This dependency can either be explicitly defined or implicit. RESTler however only considers sequences of requests that have explicit dependencies. A request can be a producer of data, or a request can be a consumer of data (or both). A dependency is only considered when a consumer relies on a producer. From these dependencies, sequences can be generated. The downside of this approach is that no request sequences that rely on implicit dependencies will be generated. Finding bugs or vulnerabilities within sequences that have implicit dependencies is not a capability of RESTler.

Another pre-optimization strategy of RESTler is to prefer shorter request sequences. Request sequences start by creating a resource, followed up by doing some other interesting operations and finally deleting the resource. This number of operations is configurable, but longer sequences that cover the same paths as shorter sequences are discarded. This could potentially hide certain states that rely on longer sequences. Also, their generated test cases cover all endpoints and generate multiple request sequences for these. This also implies that there is a degree of redundancy. This is reduced by imposing different search strategies that select a subset of test cases based on different heuristics.
After executing a request or sequence of requests, RESTler classifies the output of the PUT by looking at the HTTP status code of the response. Specifically, when a server responds with a 500 HTTP status code, RESTler classifies this execution as a bug. Using this classification, RESTler is not able to make well-informed classifications. Everything is either a bug or not a bug. Minor improvements to this classification process have been proposed in [25], where four different rules are defined. By checking these rules after the execution of a test case, RESTler is able to determine whether certain rules have been violated, thus enabling a more detailed classification of bugs. This is a promising improvement to their initial setup. However, this is still limited by the fact that their approach is black-box. Rules that impose constraints or properties on the internals of an application can not be

defined.

The limitations of a black-box fuzzer are surpassed by Gauthier et al [7]. Their fuzzer *BackREST* combines black-box techniques with lightweight white-box techniques, such as taint-analysis and coverage feedback.

BackREST aims at applications that expose an API via a frontend (JavaScript) framework. It imposes techniques to infer a state machine by crawling a frontend. During this initial phase, endpoints of the application are found and triggered by this crawler. This executes requests, which are recorded by a proxy, which allows reconstructing an API specification which contains which data is sent to which endpoints. This is then converted to an OpenAPI spec format, which is parsed to generate test-case specifications. For this, the framework *Sulley* [26] is used, which is a black-box API fuzzing library. BackREST extends this framework by including libraries for coverage and taint feedback instrumentation. Using this infrastructure, BackREST is able to find SQLi , Cross-Site Scripting (XSS) and command injection vulnerabilities. A clear difference between RESTler and BackREST however is their comprehension of state. In RESTler, state is explicitly modelled by the API specification, whereas in BackREST, little to no attention is paid to a stateful approach. In their specific setup, the applications under test did not require a stateful approach, but basic manual inspection was sufficient to properly deal with this matter.

4.1 Lessons learned from existing solutions

Looking at the papers of RESTler and BackREST, we see a similar approach in their architecture. Both approaches make use of a code generator that generates an executable script or grammar, based on the OpenAPI specification of an application. This grammar contains a set of instructions that can be interpreted by the fuzzer in order to create and execute a request, or request sequence. The key detail here is that neither approaches use a seed corpus, but rather use an executable set of instructions. These instructions can only be interpreted by their own fuzzer and therefore limit interoperability between multiple fuzzers or other tools like ZAP. BackREST makes use of a fuzzing framework called *Sulley*, whereas RESTler makes use of their definition of a test-case specification.
Since RESTler is a stateful black-box fuzzer and BackREST is a stateless grey-box fuzzer, there are obvious differences in *what* is executed and *how* the behaviour of the application is interpreted. Both implementations however use a pruning method where, based on the behaviour of the application, the initial set of instructions is pruned. This reduces the number of requests that need to be executed in order to achieve the same goal. For example, RESTler prunes request sequences that are contained in other request sequences.
Another main difference is the goals of both tools; RESTler aims at finding bugs, which are defined by a specific response status, whereas BackREST aims at finding specific classes of vulnerabilities. To find these vulnerabilities, it makes use of a taint-analysis framework, which is a method that cannot be applied in the architecture of RESTler, since taint-analysis requires feedback from within the application.
Finally, both implementations interpret the responses and other feedback during execution to generate some report of the fuzzing execution.

4.1.1 Relevant components

From these existing architectures, some principles or components are general enough to be placed in another architecture.

- Specification/seed generation based on OpenAPI specification
- (Dynamic) Taint analysis to find vulnerabilities
- JavaScript Object Notation (JSON) mutation operators

Although RESTler is a pioneer at fuzzing stateful web services, other approaches to maintaining and dealing with state exist [27]. Since their approach relies heavily on executing requests in predefined sequences, this limits the possibility of having randomness in these sequences. This also requires some form of encoding these sequences in the initial seed or specification that is fed to the fuzzer.

4.2 Smart fuzzing

To increase the overall effectiveness of the fuzzer, a typical grey-box fuzzer uses a seed scheduling policy. Such a policy evaluates how much coverage a certain seed reaches and how much time is spent during execution. In [28] a grey-box fuzzer is enhanced by a *smart* grey-box fuzzer which uses a grammar definition to generate valid virtual structures. The downside of this technique is that it is rather slow. By only using this technique, the trade-off between effectiveness and efficiency is not well-balanced; too much time is spent on generating valid structures, which results in similar efficiency of the fuzzer compared to a general grey-box fuzzer. However, both methods have their advantages. *Smart* fuzzing is incorporated into the fuzzing architecture using a technique called *Deferred Parsing*. The rationale behind this technique is that mutating and executing a seed is fast, but using their smart grey-box fuzzer to construct a virtual structure, takes much longer. The probability of constructing this virtual structure $prob_{virtual}(s)$, takes a seed s as a parameter and is computed as

$$prob_{virtual}(s) = min\left(\frac{t}{\varepsilon}, 1\right)$$

where t is the time since the last execution that led to an increase in coverage and ε is a configurable threshold. When t increases, the probability of creating this virtual structure increases. The notion of computing a probability which increases when the time since the last discovery increases, is widely applicable. A similar notion is made in [7], where their approach also uses a threshold to determine whether or not to proceed with the current actions, based on the information of prior results. Since their research is aimed at fuzzing web applications, their approach uses a metric that uses the time since the last new path discovery, for a given endpoint of the application.

5 Approach

In this research, existing methods are combined to achieve a goal where vulnerabilities can be found in web applications by fuzzing. Specifically, vulnerabilities in RESTful web applications. For this, the existing methods are described and how they are applied in the context of this research. Where the strength of RESTler lies in being able to fuzz APIs that impose dependencies between endpoints, BackREST is capable of finding vulnerabilities using grey-box fuzzing methods.

5.1 Grey-box fuzzing

The first promise of grey-box fuzzing is that it helps in achieving a higher code coverage. It also opens up other techniques where code instrumentation can be used to improve the fuzzer. Besides an existing architecture of BackREST, no grey-box fuzzers, that are aimed at web applications, exist to our knowledge. For this reason, a hybrid architecture is created where an existing grey-box fuzzer is extended such that it is capable of sending inputs to a server-based application. This separates responsibilities for scheduling and mutation from the execution of these generated inputs.
In the basis, this grey-box fuzzer uses coverage feedback to guide the fuzzer to find seeds that explore deeper paths within the PUT. This does not consider the stateful behaviour of a RESTful web application. Without considering state, the grey-box fuzzer will only be able to execute inputs on endpoints that do not have any dependencies. By combining stateful methods with grey-box fuzzing, both deeper paths within the code can be reached, as well as endpoints that have dependencies. This results in a much higher overall coverage that can be achieved by the fuzzer.

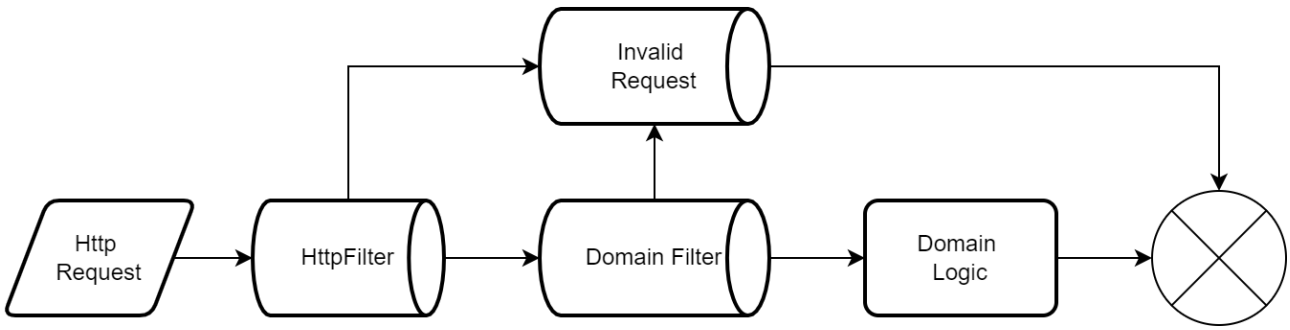


Figure 2: Generic flow of an HTTP request

Within coverage-guided grey-box fuzzing, multiple techniques can be used to increase the effectiveness of the fuzzer. A key decision here is to focus on generating valid inputs, that exhibit unique behaviour on the application. By only generating valid inputs, the seeds that are executed will not end up being rejected by the first level of input validation. A generic representation of a flow that happens during an HTTP request can be seen in Figure 2. While it certainly is interesting to trigger code that ends up in the invalid request channel, these parts are generally provided by libraries or frameworks and are therefore less prone to security flaws. Any invalid HTTP request will end up being rejected by this first filter, and will not trigger any potentially interesting parts of the code. Secondly, there are domain-specific validations. These validations put constraints on the input model, for example by restricting which headers can be used, or what the JSON structure of the HTTP Body should contain. These constraints can be specified by some form of API specification or other forms of documentation. In this research, a mutation strategy is used which aims at generating valid HTTP structures.

This does not consider any domain-specific grammar, but it reduces the number of requests that are rejected because of malformed structures. This is done by parsing the bytes of the seed and validating its correctness. Using this structure, the separate parts of the HTTP structure can be mutated such that the result of this mutation is yet again a valid HTTP structure. In this strategy, the HTTP request parameters, headers and body are treated separately. The body is assumed to be a JSON structure and is mutated as such; only the values of the existing structure are mutated while keeping the keys intact. I.e. in the following JSON structure

```
{
  "name": "Roald Dahl",
  "best_seller": "Charlie and the Chocolate Factory"
}
```

only the values *Roald Dahl* and *Charlie and the Chocolate Factory* are mutated. This is a simple strategy that could be enhanced by techniques, such as those found in [29] This however requires knowledge about the domain-specific grammar in order to prevent generating a lot of invalid structures. By only modifying the existing JSON structure, the mutation logic does not need to be aware of this domain-specific grammar.
The complexity in REST services lies in the domain model and domain-specific logic. A domain model generally can be described in terms of relations between objects or entities. A textbook example of this is the domain of a bookstore that has *Book* and *Author* entities. In this example, a Book is written by an Author. This relationship imposes a complexity where two entities are coupled by reference. This conceptual complexity increases when the size of the domain increases. This complexity is reflected in the API of these services, where these relations between entities have to be supplied by the client, to the server.

5.2 Stateful approach

The approach taken in RESTler uses predefined sequences of operations. Instead of defining test cases as predefined sets of operations, a request can also be considered to be a single operation on a resource, with inbound dependencies. For example, in Figure 1, the state $\{x, y\}$ can only be reached by having a previous state $\{x\}$ and before that, the state $\{\}$. Instead of defining an entire sequence to go from the initial state to $\{x, y\}$, in this research, an approach is used that defines a requirement of being in state $\{x\}$ before making the transition to $\{x, y\}$. This abstracts away how state $\{x\}$ is reached. When the current state of the application is in state $\{x\}$, this transition can be made. Because of the random nature of a fuzzer, this transition behaviour is not deterministic and any sequence of transitions can happen before this transition to $\{x, y\}$ is made. Of course, there is a downside to this approach: transition sequences are no longer explicitly defined and deriving which sequence of operations leads to a certain state is more difficult. However, by considering explicit dependencies, a relation between two resources can be recorded. I.e. when a resource X is created using (partial) information of the state of Y, then X has an explicit dependency on Y. This implies that operations on resource Y also could affect the state of resource X. These operations on resources are recorded during execution, after which this information is used to trace back which operations have occurred on related resources in order to achieve a certain, interesting state.
Because of the more random nature of this approach, requests that have in- or outbound implicit dependencies to an interesting resource could also result in causing the interesting state of that resource. However, since this dependency is implicitly fulfilled, this information cannot be traced back in the same way as explicit dependencies. The only general knowledge that is available at this point is

which requests occurred between the first request of the sequence and the last. This of course would potentially result in an extremely long sequence of requests.
To include information from implicit requests and limiting this to contain only relevant requests, a dynamic approach is taken which does not consider any domain/application-specific logic. The only assumption that is made here is that state is persisted in a SQL database. During a request, multiple queries are executed on such a database, which can be recorded to generate a collection of database entries that are used to fulfill the request. From this information, implicit relations to other endpoints can be determined.
This does come with at least one caveat, where instead of just a database, the backend of an application makes use of a database plus a cache in between. Instead of querying the database, first, the cache would be queried, which could lead to no query being executed on the actual database. This is just one example that explains the complexity of making such assumptions. Since this is a method that is highly dependent on infrastructural constraints, this is added as an optional feature to the fuzzer.

5.3 Improvements

In order to improve the effectiveness of the fuzzer, two white-box methods are integrated into the fuzzer architecture. Also, smart fuzzing, summarized in Section 4.2 is applied which incorporates RESTful architectural properties of a system into seed scheduling.

5.3.1 Conditional tracing

Using bytecode instrumentation, tracing is implemented in the fuzzer architecture. When conditionals are encountered in the code, instrumentation is inserted that records this conditional, which is used to create values that pass these conditions. These values are created when the instrumentation is encountered during execution and sent as feedback to the fuzzer. In following mutations, the fuzzer mutates the previous value to the new value by pseudo-random mutations. Instead of using an SMT solver, a simple construction is used that evaluates the left and right operands. For example when the condition

```
if (input[1] == 6)
```

is encountered, both the value of *input[1]* and the value *6* are sent to the fuzzer. In a following iteration, this input value might be mutated to match this encountered value *6*.

The same instrumentation is inserted for all comparison operators, which would mean that comparisons such as $<$ or $>$ would never be satisfied directly. However, since other mutation operators can be stacked by the fuzzer, these values can be slightly altered, which could satisfy the condition. Alternatively, a more complex constraint solver could be used, which results in solving these constraints more effectively, at the cost of efficiency.
This implementation of concolic fuzzing enables the fuzzer to explore branches in the code more effectively, at the cost of efficiency. When branch conditionals can be easily explored by random mutation operators, this trade-off might have a large impact on efficiency, while not impacting effectiveness as much. However, when more complex conditionals are encountered, random mutations have limited capabilities for exploring these branches. This should only be used when the fuzzer is able to send a high throughput of data to the PUT.

5.3.2 Dynamic taint analysis

Dynamic taint analysis is used in this research to find vulnerabilities within the web application. Within this research, this is scoped down to focus on SSRF vulnerabilities. Generally, this technique uses a method that marks variables with a taint; user input is marked as tainted, whereas other data is considered safe. When tainted data flows through a sink, this sink is exploitable by the user.
In this research, a method is used that does not mark variables as tainted or safe, but rather focuses on defining sinks and vulnerable behaviour. The result of this is that all data that flows through a sink is considered potentially harmful. This is a trade-off that is made because of architectural decisions; by using a 3rd-party binary fuzzer, the capabilities of extending the architecture with other 3rd-party libraries for dynamic taint analysis are limited. Because of this decision, a large number of false positives is created, where safe data is considered harmful. However, it also discards any edge case where unsafe data is considered safe.

To find vulnerabilities, the following steps are applied:

1. *Define the sink* For example, an ORM that executes a query. The sink would be the function that executes this behaviour, which enables the use of input parameters in some logic.

2. *Define which input parameters exhibit vulnerable behaviour.* For example, input that is directly inserted into a SQL query.

3. *Define when a vulnerability is triggered.* For example, when a SQL string contains the injected payload when being executed, this clearly is not sanitized by the application and will reach the database.

4. *Create input that contains malicious payloads.* This requires some knowledge about which parameters can be used to inject these payloads.

By using code instrumentation, these structures are inserted in the bytecode. When such a structure is encountered during execution, the fuzzer can take two actions: either the vulnerability is triggered, or the sink is triggered with a non-malicious payload. When the vulnerability is triggered, this information is used by the fuzzer to include in a report. When the payload is non-malicious, feedback is sent to the fuzzer that helps in creating payloads that exploit the vulnerable behaviour of this sink.

5.3.3 Smart fuzzing

In the hybrid architecture of this research, a similar situation as in [28] occurs: mutating seeds is quick, but executing these seeds is slow. This means that every execution of a seed that is unlikely to produce an interesting result should be avoided as much as possible. To do so, a similar concept is used, which computes a probability using a seed. This probability represents the chance of executing a specific seed. When an execution produces no interesting results, the chances of executing similar seeds will be reduced by using the algorithm in Algorithm 1. By leveraging a probability of execution, seeds that produce interesting results are prioritized. This probability is evaluated after mutating the seed.

5.4 High-level design

These aforementioned methods are combined into a single design which is shown in Figure 3. This is a high-level overview that displays how the different methods relate to each other and how the hybrid

Algorithm 1: Stateful grey-box fuzzing loop

Input : corpus, fuzzer, iterations, PUT
Output: R
Data: $S \leftarrow \{\}$, $P \leftarrow \{\}$
while $iterations > 0$ **do**
    $s \leftarrow chooseNext(corpus, fuzzer)$
    $s' \leftarrow mutate(s, fuzzer)$
    if s' *is valid* **then**
        $d \leftarrow getDynamicVariables(s')$
        $trace \leftarrow \{\}$
        $resolved \leftarrow \{\}$
        for $v \in d$ **do**
            $key \leftarrow getResourceKey(v)$
            $resource \leftarrow S[key]$
            $trace \leftarrow trace \cup resource$
            $resolved \leftarrow resolved \cup resolve(resource, v)$
        end
        $s'' \leftarrow resolveDynamicVariables(resolved, s')$
        if $P[s'']$ *is null* **then**
            $P[s''] \leftarrow 1$
        end
        $p \leftarrow P[s'']$
        if $p \geq random(0,1)$ **then**
            $r \leftarrow executeHttpRequest(s'')$
            $i \leftarrow getInstrumentationFeedback(PUT)$
            $P[s''] \leftarrow updateProbability(r, i)$
            for $resource \in trace$ **do**
                $resource.trace \leftarrow resource.trace \cup r$
            end
            if $300 > r.httpstatus \geq 200$ **then**
                $key \leftarrow getResourceKey(r)$
                $resource \leftarrow parseResponse(r)$
                $resource.trace \leftarrow trace$
                $S[key] \leftarrow updateState(S[key], resource)$
            end
            $fuzzer \leftarrow updateFuzzer(i)$
            if $isInteresting(r, i)$ **then**
                $R \leftarrow R \cup generateReport(s', s'', trace, r, i)$
            end
        end
    end
    $iterations \leftarrow iterations - 1$
end

design operates. This design is described in terms of an abstract algorithm in Algorithm 1.

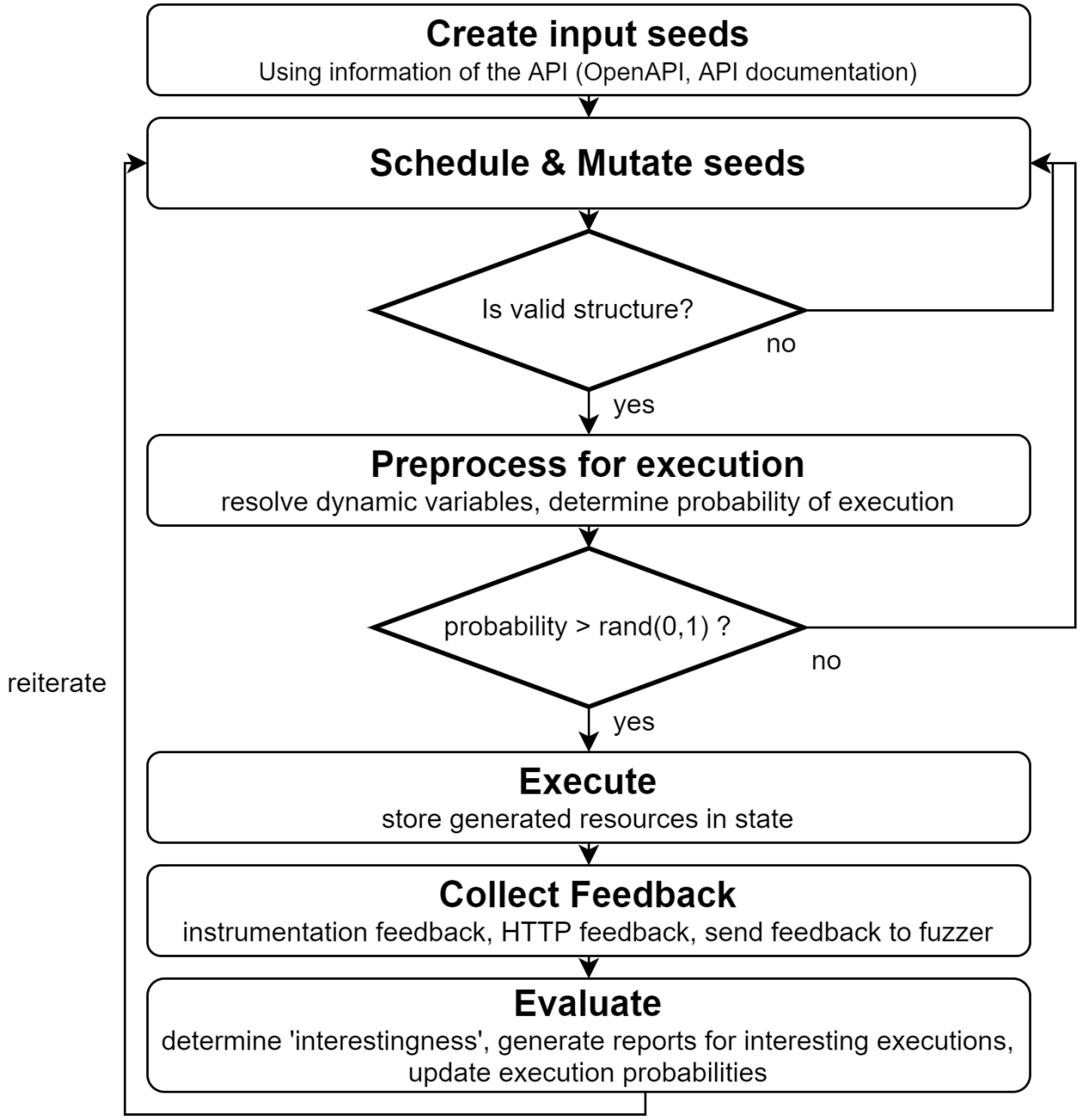


Figure 3: Design of a stateful, grey-box fuzzer

6 Implementation

In this chapter, the implementations of methods described in Chapter 5 are described. Starting with a generic architecture description, followed by a more detailed, concrete implementation after which the processes within this architecture are elaborated on.

6.1 Requirements

In the following section, functional and non-functional requirements are described which have been used as a starting point for developing the implementation of GRFUZZ.

6.1.1 Functional requirements

In order to define a high-level architecture, first, the functional requirements need to be known. Based on these functional requirements, the choice can be made to either select an existing tool/framework or to completely start from scratch. The following requirements are essential for a stateful grey-box api fuzzer.

Statefulness
Whenever there is a dependency between requests, the fuzzer should be able to resolve these dependencies dynamically. Consider the case where a resource is created by a *POST* request, which returns the id of the created resource. In a *GET* request to retrieve this resource, this id is required in the search parameters of the request:

GET /resource?id=123

Here, the *id* parameter is a dynamic variable. Dynamic variables can be passed as parameters in the request line (as search parameters), in the headers and in the request body. All three options should be considered.

HTTP Requests
A fuzzer should be able to send the mutated seed to the PUT over HTTP. HTTP operates over Transmission Control Protocol (TCP), so the minimum requirement of a fuzzer is that it should be able to open a TCP connection to a server and send its input over this connection. Ideally, it is able to understand the HTTP protocol and produce inputs that conform to this protocol. SSL/TLS should be considered, but for a proof of concept, this is irrelevant, since the goal is not to fuzz the HTTP protocol.

Grammar-based mutation
RESTful web services expose APIs that generally use JSON as a serialization format for data; Data that is structured to conform to a specific schema that defines which fields can be encoded in this JSON structure. Such a schema can either be implicit or explicitly defined using a form of API documentation, like OpenAPI/Swagger. The main point of such a definition is that it is known beforehand what the JSON structure should look like for a specific request for a specific resource endpoint. Based on this knowledge, a fuzzer should perform mutations that only generate valid inputs. Optionally, it could also deliberately generate invalid inputs to find bugs in the input validation of an application.

Feedback (Grey-box)
The key point of a grey-box fuzzer is that it is able to receive feedback from the PUT during execution. Using this feedback, a fuzzer is able to guide its process towards achieving a specific goal. The goal of fuzzing web services is to find bugs and vulnerabilities. By aiming toward a high coverage, more parts of the application will be explored, thus increasing the chances of triggering code paths that exhibit vulnerable behaviour. Feedback that can be considered:

- Coverage
- Taint analysis
- HTTP feedback (response processing)

Further extensions
Webservices often have several security measures in place. These measures could also affect the way a fuzzer has to send data to the application. For example, authentication and authorization functionality can be added to determine whether the client is able to perform a specific request. Since there are a plethora of authentication protocols and the goal is not to fuzz these authentication protocols, it is irrelevant to the research question. However, in order to be usable, a fuzzer should be able to deal with this.

Assuming that an application has an OpenAPI specification, this specification could be used to generate parts of the dynamic part of the fuzzer. For example, when using a seed corpus, initial seeds can be generated based on this specification. Other components, like grammar-based mutators, could also be aware of this specification and use this to generate inputs.

6.1.2 Non-functional requirements

Besides being functional, a fuzzer has non-functional requirements that put constraints on some design decisions.

Effectiveness
Traditional binary fuzzers mostly aim at high performance by generating a lot of unique inputs efficiently. Here, the bottleneck is most often found in the fuzzer itself. However, in web services, the execution time for each request ranges from a few milliseconds to a few seconds. Even though it is heavily influenced by the infrastructure and complexity of the application, it generally holds for most web services. This shifts the bottleneck towards the application itself; being able to generate a lot of inputs does not affect the number of executions per second as much. For this reason, it is more important to focus on creating interesting, effective inputs.

Efficiency
Being able to execute more requests per second goes hand in hand with executing effective inputs. When the application itself becomes a bottleneck, a sensible approach would be to increase the resources of the application. Which might affect the performance of a single request, but mostly increases the performance when dealing with multiple requests. A fuzzer could potentially deal with multiple parallel executions at the same time. This creates a more complex situation with concurrency issues, but should theoretically increase the efficiency of the fuzzer.

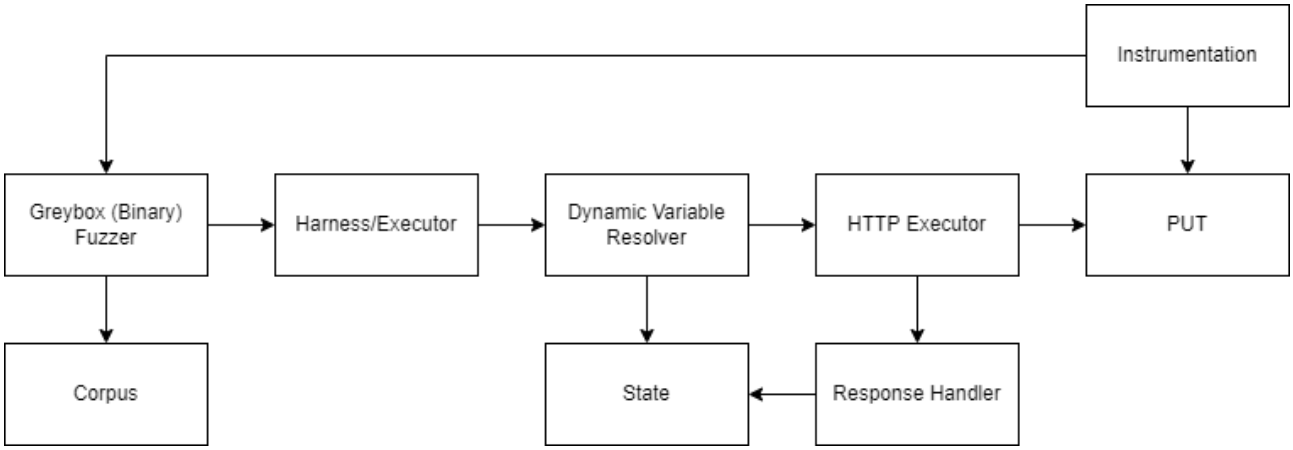


Figure 4: Logical view

6.2 Architecture

In Figure 4, the logical view of the architecture of GRFUZZ can be seen. The arrows represent a dependency between components, where the direction corresponds to outgoing dependencies. This is a high-level view in which no specific implementations are described yet, leaving room for replacing certain components with different implementations, if the interfaces between the components allow this.

Corpus
The corpus contains a set of seeds which are either found during execution or generated before the first execution. The format of a seed should be as general as possible, such that multiple fuzzers or tools can interpret this format.

Fuzzer
Any grey-box fuzzer that supports grammar-based mutations and has some interface to deal with execution feedback, can be used here. Note that it should be possible to intercept the final mutated input before it is sent to the application. Since most fuzzers require the user to define a fuzzing harness, this is a common design principle in a fuzzer.

Harness/Executor
The second part of the fuzzer is an extension of binary fuzzing. This part acts as an intermediary between the binary fuzzer and the web service under test and is therefore responsible for dealing with all challenges that are specific to web services.

Dynamic Variable Resolver
Whenever an input contains a dynamic variable, this dynamic variable needs to be replaced. This component acts as a pipe and filters component which processes the input and outputs the processed input in the same format. This component is responsible for creating and executing the query to resolve these dynamic variables.

State
The State is a data store that contains all currently available objects for dynamic querying. It should expose an interface to add a new object and retrieve an object or resolve a specific query. This com-

ponent can either be an in-memory storage, or a more resilient database implementation.

HTTP Executor
The HTTP Executor is responsible for setting up a TCP connection to the PUT and sending the input data over this socket. The response of the application is also received by this component, but not processed.

Response Handler
The Response Handler handles the responses from the execution. This involves creating a new object in the State on object creation and dealing with responses that should be contained in the final report.

Instrumentation
The PUT should be instrumentable. Instrumentation should be extensible, but have in-built for code coverage at least. How instrumentation is added to the code depends on the target application. For example, instrumentation can be added by recompiling the application, but also by dynamically altering the bytecode.

6.3 Jazzer implementation

The aforementioned architecture is implemented by using Jazzer [30] as a binary fuzzer. This extends the architecture with implementation-specific components, which can be seen in Figure 5. Jazzer provides an interface to LibFuzzer and a Java Agent. This makes it extensible and compatible with Java applications. LibFuzzer and the Java Agent are two components that are coupled through shared memory and an API. This API calls C++ code via Java Native Interface (JNI), which tightly couples these two components. In a web-service setup, these components should ideally be decoupled, such that the fuzzer does not need to be deployed on the PUT.
LibFuzzer exposes an interface *LLVMCustomMutator* [31], which can be used to define a custom mutation strategy. This is used for grammar-based mutations. In this research, this mutator is limited to only producing valid JSON message mutations, but this could be extended by deriving a more detailed grammar definition from an OpenAPI specification, such as seen in the methods of RESTler.

6.3.1 Java Agent

Jazzer makes use of the Java Agent framework, which allows to dynamically instrument bytecode. The Java agent can be attached to the PUT on startup. Jazzer inserts instrumentation for coverage, as well as dynamic taint instrumentation and instrumentation for branching instructions. This feedback is sent back to the LibFuzzer interface using messaging.

6.3.2 Messaging

A key implementation detail is that all communication between the fuzzer and the PUT is implemented by a TCP messaging system. This is a low-level interface, which enables high throughput. This communication can be distinguished into two categories: asynchronous and synchronous communication. Asynchronous communication is used for all feedback of branching instructions, whereas synchronous communication is used for coverage and taint-analysis feedback.
Messages are only interchanged between the Executor and the Java Agent. These messages are built using the following generic format:

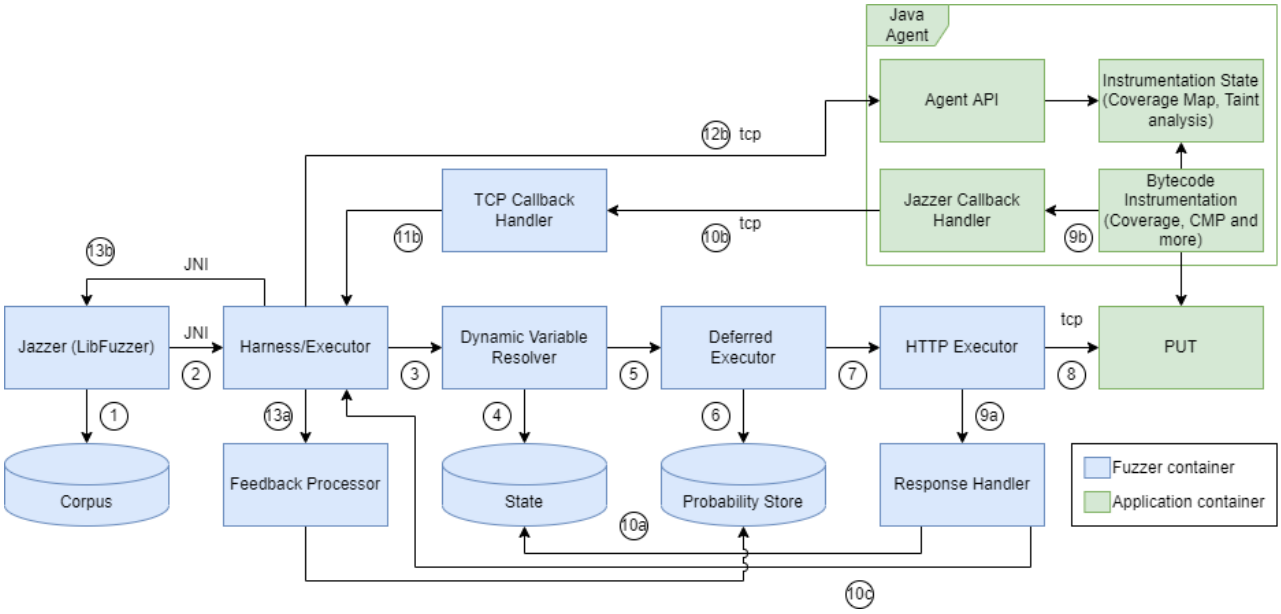


Figure 5: Hybrid architecture implementation (process view)

```
size: <SIZE>
type: <TYPE>
<DATA>
```

These headers allow for the receiving end to handle different types of messages. Data is encoded using Base64 encoding.
To allow the Jazzer Java Agent to communicate using these messages, the Agent is extended with these TCP functionalities.

Asynchronous communication
The Java Agent opens a ServerSocket, that dispatches all incoming socket connections to a Callback Handler. During the startup of the fuzzer, this socket connection with the Java Agent is initiated by the *Executor*. Within the Java Agent, a message queue is used to forward messages from bytecode instrumentation to the Executor. This message queue also allows for the Java Agent to throttle the outgoing messages; since it is application-specific how much feedback is received during execution, this could potentially result in a number of messages that is larger than the Agent can handle within a certain time frame. This is implemented by keeping an expiration time for all messages. When the queue overflows with messages, older messages are removed. By doing this, some messages may get lost, with the benefit of not having to block until all messages are sent.

Synchronous communication
Similar to asynchronous communication, a ServerSocket is opened on the Java Agent and the socket connection is initiated by the Executor. This communication is solely used by the Executor to request specific data from instrumentation. This data is collected during execution and stored in an in-memory object on the Java Agent. This type of communication is used when the Executor needs to wait for specific information in order to determine the interestingness of an execution. By doing this in a blocking manner, it is ensured that no data is received that is not created during this specific execution.

6.3.3 Conditional tracing in Jazzer

Jazzer provides an extensive set of conditional instrumentation operators, which mostly focus on primitive conditional checks and string comparisons. This covers a large set of conditional operators since many operators are constructs made of primitive operators. For example, when an integer comparison is encountered on the bytecode level, the instrumentation is inserted which records both integer values and sends these values back to the fuzzer. These values can be used in following mutation cycles where existing value are mutated to the other value seen by the fuzzer. Now, this instrumentation is inserted for all comparison operators, which would mean that comparisons such as $<$ or $>$ would never be satisfied. However, due to other mutation operators which can be stacked, these values can be slightly altered, which could satisfy the condition. Alternatively, a more complex constraint solver could be used, which results in solving these constraints more effectively, at the cost of efficiency.
More complex operators are seen in pattern matching, which uses Regular Expressions to evaluate whether a string value matches this pattern. While not supported by Jazzer out of the box, the framework is extensible, which allows for custom implementation of such instrumentation constructs. A simple Regex Solver is used here, which dynamically creates values that correspond to a regular expression (see Appendix 7). When this expression evaluates to false, the Regex solver will generate a value that matches this regular expression.

6.3.4 Dynamic taint analysis in Jazzer

Jazzer uses an implementation of this technique which is based on dictionary mutations. Step 4 in the listed steps in Section 5.3.2 is where Jazzer uses the interface of LibFuzzer for tracing branch conditions. By storing which values are compared to other values, LibFuzzer is able to mutate a specific value to a given value. When a value reaches the entry point of this sink, LibFuzzer is able to mutate this specific value to a malicious payload. Note that this value is not guaranteed to originate from the user-supplied input, thus there are no guarantees that this value will be mutated to this malicious payload. Another minor detail is that the implementation of LibFuzzer stores a dictionary of found comparison instructions, of which a random entry is selected to replace a value in a mutation operator. This random entry does not necessarily contain a pair that is used in the input that is being mutated. In the implementation of GRFUZZ, LibFuzzer is slightly modified such that it occasionally tries to find a pair of which the left-hand side is found in the input that is being mutated. This increases the effectiveness slightly.

In the implementation of GRFUZZ, a sanitizer for SSRF vulnerabilities is created. Following the steps:

1. *Define the sink.* The target sink of the sanitizer is the Apache HttpClient [32], targeting the *doExecute* method.

2. *Define which input parameters exhibit vulnerable behaviour.* The *doExecute* receives three parameters, of which the HttpHost *target* is an exploitable parameter. This is directly used to set up a socket connection to a certain host.

3. *Define when a vulnerability is triggered.* To ensure that the SSRF vulnerability is triggered, a mock server implementation is hosted on a certain address. When this server is called, it

responds with a valid HTTP message, containing a specific header. This custom header signals that a response came from a server that should not have been called during execution. This means that the request has triggered an SSRF vulnerability.

4. *Create input that contains malicious payloads.* There are many payloads for SSRF vulnerabilities. These payloads should be constructed such that these payloads contain an address that is identical to the address on which the mock server implementation is hosted. Note that SSRF vulnerabilities can be both vulnerable to localhost bypasses or external request forgeries. Localhost bypass attacks are considered more vulnerable since they can expose private network infrastructures.

Now, whenever the flow of an application reaches this sink, the HttpHost parameter is retrieved and is sent together with an SSRF payload to LibFuzzer. If this original parameter originated from user-defined input, LibFuzzer will eventually replace this parameter with the SSRF payload, thus triggering the vulnerability.

6.3.5 Smart fuzzing implementation

In the implementation of GRFUZZ, a low-level grey-box fuzzer is used, which only allows us to apply the smart fuzzing strategy on the intermediate level at the *Executor* (see Section 6.1). Since both LibFuzzer and the Executor are affecting which seeds are executed, both implementations must have the same goal in mind. LibFuzzer uses an *entropic* scheduling technique, which uses coverage and execution time in order to compute the cost/benefit ratio. This results in LibFuzzer prioritizing seeds that have a high chance of discovering new coverage, at a low cost. The smart fuzzing strategy has a similar goal, except that it is capable of taking more metrics than coverage into account in order to compute whether a seed is likely to execute interesting behaviour.

To apply this strategy in GRFUZZ, we first need to determine which information can be used to determine this probability. Because of design decisions, the execution layer does not have the information about the original seed, but only the mutated seed. Since this is likely to be different for every mutation, it is not right to use this as a function parameter. What we do know, is that in REST web services, resources are located using their appropriate path. This path can be extracted from the mutated seed, to use as a function parameter in combination with the HTTP method. For example, using the seed, shown in Listing 5

Listing 1: Seed 1

```
POST /author HTTP/1.1
Connection: close
Content-Length: 107
Host: localhost
Content-Type: application/json

{"name":"author_name","authorType":"POET","externalPropertySource":"http
   ↪ ://127.0.0.1:8090/author_property"}
```

the key *POST /author* is extracted.
Since multiple seeds exhibit behaviour on the same path, this computed probability will be used for multiple seeds. This seems negligible, however, this could lead to a defective seed lowering the overall probability of other seeds of the same path, which could have a positive effect. Another key

detail is that the probability of execution should never be zero, thus making the execution of certain paths impossible. To overcome this, every non-execution will increase the probability slightly.
During execution, the following results affect the probability of future executions ((+) for increase, (−) for decrease):

- (+) The execution has resulted in the discovery of new paths (increased coverage)
- (+) The execution has triggered a vulnerability within a sink. Increase the probability.
- (−) The execution has not led to any interesting behaviour. Also, increase the probabilities of all other paths.
- (+) The execution was skipped based on this probability. Slightly increase the probability.

6.3.6 State

In order to resolve dependencies of requests at runtime, an approach is taken that stores information from previous requests in a state repository which is accessible at runtime. The state repository contains all required information about the created resources and stores an event log of actions that are performed on these resources. This allows recreating a log of all operations that are performed on specific resources.

Dependencies between requests are encoded through first-level dependencies. For example, when a request *A* has a dependency on *B* and request *B* a dependency on *C*, there will be no information about *C* in request *A*. This creates an assumption that the dependency between *B* and *C* should be fulfilled before request *A* is executed. This can be strictly enforced, which will ensure that all requests are only executed when all dependencies are fulfilled, however, some scenarios are only triggered when some of these dependencies are either invalid or can not be fulfilled for other reasons. Therefore, it is not enforced that all dependencies are resolvable before executing a request.

To encode these dependencies in a seed, an encoding format is chosen which embeds a path to a resource and an optional path within that resource. The most common example of a dependency between requests is where an identifier or URI of another related resource is required in a request. For example, when creating a book, the client is also required to provide the identifier (*id*) of the author:

$$POST\ /author/ < id > /book$$

Encoding this results in

$$POST\ /author/\$\${authorId}\$\$/book$$

where *authorId* resolves to the resource identifier of a resource on the resource path */author*. Typically, these paths are constructed hierarchically, but exceptions may occur. For this reason, the path that *authorId* resolves to can be overridden.

To resolve a URI that contains multiple dependencies, the dependencies are resolved from right to left. For example, a request to retrieve a specific book could look like this:

$$GET\ \$\$/author/\{authorId\}/book/\{bookId\}\$\$$$

Looking at Figure 6, it can be seen that *Author_2* does not have any books. By resolving from right to left, first, a book resource is resolved after which the *authorId* is resolved by looking at the relations

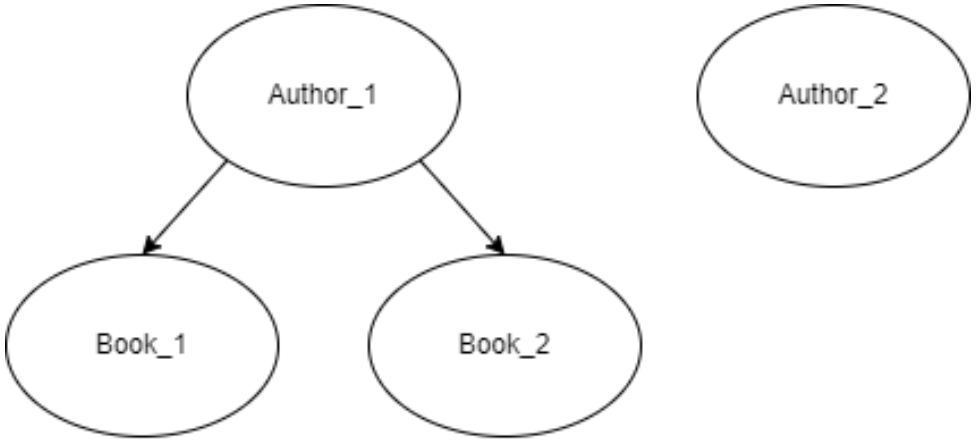


Figure 6: Resource hierarchy

of this book resource. Using this method prevents selecting resources that do not contain the full relational structure to resolve all variables. Of course, when no resources exist of this type, not all variables can be resolved. Optionally, this could be a point to prioritize specific requests in order to fulfil this dependency. I.e. when the request

$$GET\ \$\$/author/\{authorId\}/book/\{bookId\}\$\$$$

cannot resolve the book dependency, prioritize a request that produces a book.

To maintain the state, a key-value map is used which maps a resource path to a set of resources. This reflects how REST defines resource locations. When a request is performed, several things can happen on a server which affect the state:

- A resource is created, through a POST request
- A resource is updated, through a PUT request
- A resource is deleted, using a DELETE request
- An action is performed on a resource, through any type of request

When a resource is created, a server has to inform the client about how to access this resource. Although this is open for implementation, a common principle in REST is to either return the created resource, which contains a resource identifier or by returning a URI in the *Location* header. This information is stored within a State map within the *Executor*. Whenever an operation occurs that affects the state of a specific resource, this operation is linked to this resource. Also, whenever an operation happens on a dependent resource, this operation is linked to this resource. Using Figure 6 as an example, any operation that uses a dynamic variable of *Author_1* will be attached to the trace of *Author_1*. This trace is also attached to its child resources.

1. POST /author (creates Author_1)
2. POST /author (creates Author_2)
3. POST /author/Author_1/book (creates Book_1)

4. POST /author/Author_1/book (creates Book_2)

5. PUT /author/Author_1/book/Book_1 (updates Book_1)

These operations will result in the following traces:

- Author 1: $\{1,3,4\}$
- Author 2: $\{2\}$
- Book 1: $\{1,3,5\}$
- Book 2: $\{1,4\}$

6.3.7 Classification of results

In order to collect information about the fuzzing campaign that is useful, first, we need to define which information is available. Each execution collects the following information:

Coverage
Each execution contains coverage feedback about which lines have been triggered. This is not unique per execution, but rather a sum of all executions. The difference between executions displays which coverage is new per execution.

Bytecode instructions
Each execution traverses a path within the application, resulting in a set of comparison instructions that have been encountered. This also includes instructions that are programmatically generated in Sanitizers.

Sinks
Whenever a sink is triggered and the condition for a vulnerability is fulfilled, feedback is sent back which contains information about which sink has been triggered at which point in the code. This is done by attaching a stack trace.

HTTP Response
When the server responds, it sends back an HTTP response, containing at least a status code which informs about if the request was successful [33]. Additionally, this could contain information about resources being created, or other response body data that could indicate some interesting behaviour.

Executions are classified as either interesting or non-interesting executions. An execution is interesting when either the coverage has increased, a sink has triggered a vulnerability or an unexpected HTTP status code, between 500 and 600, has been received. Interesting executions are also persisted to an output file in which the reason or reasons for classifying as interesting is given, alongside with information on how the interesting state is reached, using the traces provided by the stateful implementation. Finally, in the case of vulnerabilities, a stacktrace is attached which provides the user with helpful debugging information.

6.4 Process view

In Figure 5, the process of the main fuzzing loop can be seen. It is a linear process, which branches at the execution on the PUT (denoted by *a* and *b*). This is the implementation of the abstract pseudo-code in Algorithm 1

Step 1: Retrieve a seed from the corpus
Based on a seed-selection policy, retrieve a seed from the corpus. This is fully determined by LibFuzzer, which optimizes the execution by selecting seeds that execute quickly. Besides execution time, it considers the number of new features that a seed explores in order to assign a weight to this seed. The more features a seed explores, the higher the chance that it will be selected for mutation.

Step 2: Mutation
LibFuzzer takes this seed and mutates this according to its internal mutation operators. This also includes the custom mutator. In this custom mutator, the seed is parsed to an HTTP struct. Note that invalid HTTP structures are discarded by the fuzzer and will not be mutated further. After parsing, the separate parts of the HTTP message are mutated. If the body contains a JSON payload, this is also separately mutated by parsing the entries of this JSON structure. This mutated seed is then sent to the harness of LibFuzzer.

Step 3: Execution
The Executor takes this mutated seed and validates its structure. When it contains a valid HTTP structure, it is passed along to pre-processors, including the Dynamic Variable Resolver. After preprocessing, this input is passed to the HTTP Executor (Step 7).

Step 4 & 5: Preprocessing
The Dynamic Variable Resolver replaces all dynamic variables by resolving them through the State repository.

Step 6 & 7: Deferred execution
The Deferred Executor partially parses the mutated seed and retrieves, (or computes if absent) the probability of execution based on the path of this seed.

Step 8: HTTP execution
Finally, the input is sent over a socket to the PUT. This connection sets up a new connection per request. The HTTP response is then passed along to the Response Handler.

Step 9a, 10a & 10c: Response handling
The response of the PUT is handled. This contains information about the request and the actual response. Based on this knowledge, a representation of a created object can be stored in the State object. This is done by parsing the request line of the HTTP request to determine the resource type. I.e. *POST /books* results in a key *books*. The response is also sent back to the Executor in order to process it for other purposes (Step 13a).

Step 9b: Instrumentation feedback
Instrumented code calls the Jazzer Callback Handler with the corresponding parameters.

Step 10b: Callback handling
The Callback handler processes the callbacks and serializes this callback as a message. This message is then written to the TCP socket. This messaging is processed asynchronously.

Step 11b, 12b & 13b: TCP Callback handling
The TCP Callback Handler receives this message and dispatches it to the correct handler. These specific handlers then call the LibFuzzer API.

Step 13a: Feedback processing
The Executor aggregates all feedback received from steps 10c, 11b and 12b which is sent to the Feedback Processor. This determines whether an execution is interesting, based on this information. An interesting execution is then persisted into an output file. With the same information, the Probability Store is updated, based on the level of interestingness.

7 Validation

During the design and implementation of the fuzzer, a simple testing application is used. This application represents a generic design of a RESTful web application. Using a textbook example of a service that keeps track of *Authors* and *Books*, this service has endpoints that require information from previous requests (see Figure 7).

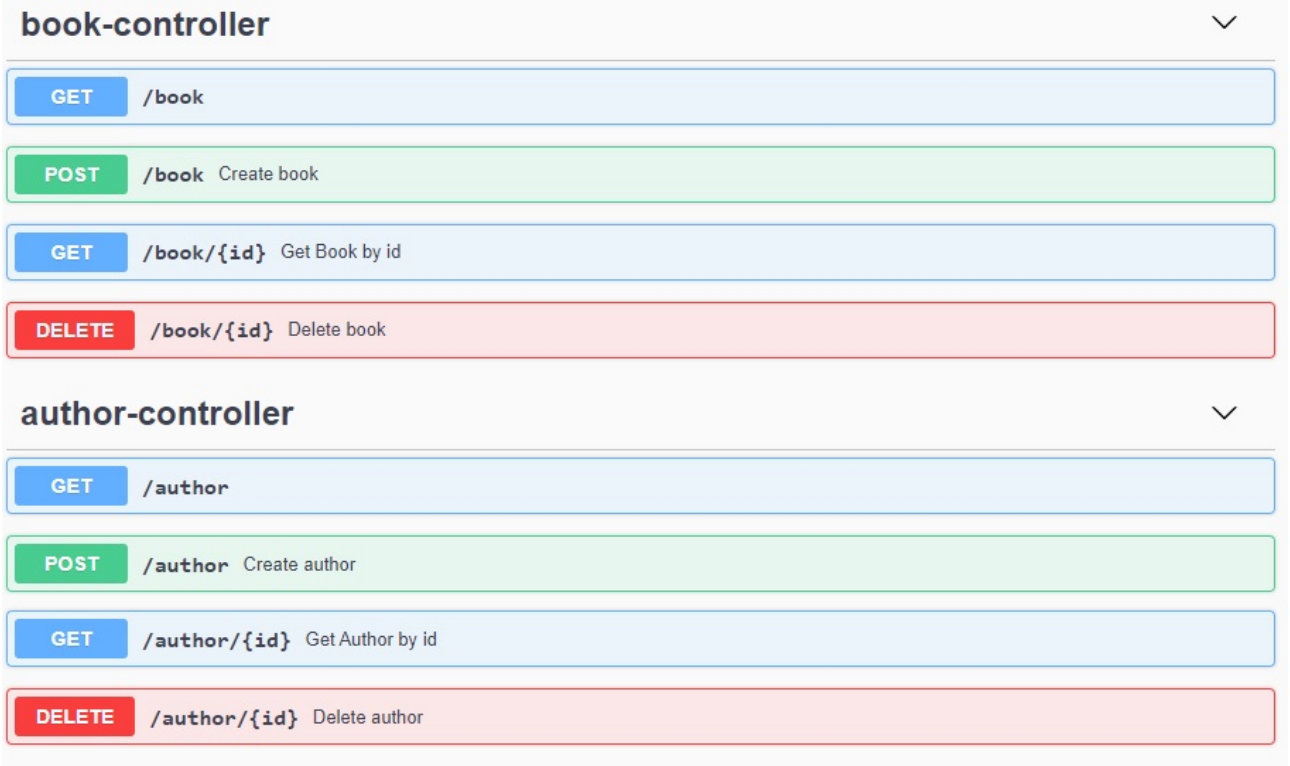


Figure 7: Swagger OpenAPI UI of an Author & Book service

To validate previous assumptions about concolic fuzzing and dynamic taint analysis, this service is designed such that it contains specific branching statements that can only be explored by concolic fuzzing and such that it contains sinks that are exploitable by dynamic taint analysis. These sinks are inserted at different parts of the application, such that different levels of complexity (in terms of cyclomatic complexity) trigger these paths. This results in a classification with three degrees of complexity:

1. Level 0 (Surface level); No information from the input is used in order to traverse branches within the code. All code of this endpoint will be executed, regardless of the data sent to this endpoint (apart from model validity).

2. Level 1; Some information of the input is used directly in order to traverse one or more branches within the code. Not all code paths of the endpoint will be executed on every input.

3. Level 2; Some information on prior requests is required to traverse one or more branches within the code. Not all code paths of the endpoint will be executed on every input.

For example, in the domain logic of the AuthorService, two Level 1 statements are inserted (Listing 2).

Listing 2: Create Author domain logic

```
    fun createAuthor(dto: AuthorCreateOrUpdateDTO): AuthorResponseDTO
    {
        var authorProperty: String? = null

        if (dto.authorType == AuthorType.NOVELIST)
        {
            authorProperty = vulnerableService.getAuthorExternalProperty(dto)
        }

        if (dto.authorType == AuthorType.POET && dto.name == "Hackerman")
        {
            authorProperty = vulnerableService.getAuthorExternalProperty(dto)
        }

        val entity = Author(UUID.randomUUID(), dto.name, emptyList(),
            ↪ authorProperty)
        return repository.save(entity).let(AuthorResponseDTO.Companion::
            ↪ fromEntity)
    }
```

This displays how the properties *authorType* and *name* are used to trigger vulnerable code within a sink. In order to trigger these branches, a fuzzer needs to be capable of mutating the input to match these values. Of course, this is a trivial example, but it is not uncommon for web applications to exhibit different behaviour on different user input. The distinction made here is that some values are bounded by strict sets of enumeration values, whereas other values are bounded by more loose constraints, such as string-based logic. These strict constraints are well-defined constraints which can easily be reflected in a grammar specification. For example, enumeration constraints can be defined in a grammar specification like the following:

```
"protocol": {
    "type": "string",
    "enum": [
        "openid-connect",
        "saml"
    ]
}
```

However, this cannot be done for loose constraints. This makes fuzzers that rely on a grammar (Open API specification), less able to find these constraints. With the use of concolic testing, also these loose constraints should be satisfiable by the fuzzer.

Secondly, Level 2 statements are inserted in the following manner (Listing 3):

Listing 3: Create Book domain logic

```
    fun createBook(dto: BookCreateOrUpdateDTO): BookResponseDTO
    {
        val author: Author = authorRepository.findById(dto.authorId).orElse(
            ↪ null) ?: throw NotFoundException()
```

```
        var isbn: String? = null

        if (dto.bookType == BookType.PAPERBACK)
        {
            isbn = vulnerableService.getISBN(dto)
        }

        if (dto.bookType == BookType.HARDCOVER && author.name == "Hackerman++")
        {
            isbn = vulnerableService.getISBN(dto)
        }

        val entity = Book(UUID.randomUUID(), dto.name, isbn, author)
        return repository.save(entity).let(BookResponseDTO.Companion::
            ↪ fromEntity)
    }
```

For the second branching statement, a property from the *Author* resource is used. This resource is created in a prior request, meaning that at the moment that this *Author* resource is created, no knowledge is available about the requirement of this *name* property. This makes these statements inherently complex since this prior request has to fulfil requirements that are determined in a later request.

Within these branching statements, an external request is executed by using a parameter that originates from the user input. This URI is parsed by the application before executing the request. A deliberately wrongly implemented sanitization is added to this domain logic; following the prevention guidelines on SSRF vulnerabilities [34]: using a blacklist approach leaves a lot of room to bypass such limitations. A whitelist approach prevents much more abuse. For that reason, a blacklist approach is taken, which filters out any request targeted at the host *localhost* (see Listing 4). Since there are many ways to encode a host that resolves to the localhost, this can easily be bypassed. For example, using *127.0.0.1* on a Windows machine resolves to the same network address as localhost. A dummy application is deployed that runs on the address localhost:8091. Any request sent to this server will result in a response that contains a header which provides information about a triggered SSRF vulnerability. This information is then used by the fuzzer to trigger a scenario of an execution with interesting results.

Listing 4: URI sanitization

```
private fun sanitizeUrl(url: String): URI
    {
        return runCatching { UriComponentsBuilder.fromUriString(url).build() }
            .getOrNull()
            ?.takeIf { it.host != "localhost" }
            ?.toUri()
            ?: throw UnauthorizedOperationException()
    }
```

These branching statements are displayed in the Table 2, where a ground truth is established by the total number of sinks that require a certain level of complexity. The fuzzer is then run against this ap-

plication for 15 minutes, during which coverage information is collected. This coverage information displays which parts of the code are executed during runtime. For each sink, the time till the first time reached is recorded. For complexity levels with multiple sinks, this number represents the average of all occurrences.

Table 2: Different levels of complexity, explored by GRFUZZ

level	amount	time (s)
0	2	12
1	1	45
2	1	136

Within these 15 minutes, all sinks that have been found have all been exploited by a mutated URI within the JSON body of the request. This displays that using the grey-box approach for concolic fuzzing helps in passing branching conditions, within a small application. In larger applications, of course, the number of branching conditions will increase substantially, potentially decreasing the efficiency of this approach. Also, note that because this dummy application is extremely small, it is also extremely fast. Up to 50 requests per second could be handled, which places the metrics in perspective; for the level 0 complexity, roughly 600 requests are required to trigger the sink.

7.1 Example run

To demonstrate how different methods and processes help in order to increase the efficiency and effectiveness of the fuzzer, an example run is displayed in terms of steps that are taken during execution. This example reflects the processes that are described in Section 6.4.
Initially, the fuzzer starts without any information regarding state or comparison instructions. Seeds are defined by the tester. For this example, two seeds (See Listing 5 and 6) are used, in order to demonstrate how the stateful methods are applied.

Listing 5: Seed 1

```
POST /author HTTP/1.1
Connection: close
Content-Length: 107
Host: localhost
Content-Type: application/json

{"name":"author_name","authorType":"POET","externalPropertySource":"http
   ↪ ://127.0.0.1:8090/author_property"}
```

Listing 6: Seed 2

```
POST /book HTTP/1.1
Connection: close
Content-Length: 113
Host: localhost
Content-Type: application/json
```

```
{"name":"book_name","bookType":"PAPERBACK","authorId":"$${authorId}$$","
   ↪ isbnSource":"http://127.0.0.1:8090/isbn"}
```

7.1.1 Initialization

Firstly, the PUT and fuzzer are initialized. This results in an empty State and an empty dictionary of comparison instructions (CMPDict).

1. During the startup of the PUT, the Java agent is attached to the Java Virtual Machine (JVM), which instruments the bytecode of classes that are configured to be instrumented. In this example, all classes that are not part of 3rd-party libraries are instrumented. Since coverage is recorded in a coverage map, the size of this map increases when more code is instrumented. Therefore, messages, which contain information about the size of this coverage map, are stored in a queue.

2. Next, the fuzzer itself is booted up. Besides initializing LibFuzzer, the Executor is initialized as well.

   (a) Socket connections to the Java agent are setup

   (b) A thread is started to periodically request and record coverage information

   The messages from the Java agent regarding the size of the coverage map are retrieved. These increase the size of the coverage map which is kept on the Executor.

3. The initial seeds are loaded into the seed corpus by LibFuzzer. This is done in an initial run that executes these seeds and records the newly discovered features and execution time. This is later used by LibFuzzer to determine the entropy of a seed, which is used for prioritization.

7.1.2 Execution

During execution, seeds are processed by LibFuzzer and the Executor. During and after execution, feedback is collected which is sent back to LibFuzzer.

1. Firstly, a seed is selected by LibFuzzer. This is done on a basis of seed entropy. For this example, Seed 1 (Listing 5) is selected.

2. The seed is then mutated by a random selection of mutation operators of LibFuzzer. This also includes the *LLVMCustomMutator*. In the *LLVMCustomMutator*, the seed is parsed to an HTTP structure, of which the parts are mutated separately; in this case, the JSON structure of the HTTP body is mutated: a random selection of key-value pairs is mutated using the built-in mutation operators of LibFuzzer. After this mutation, the custom mutator validates whether the resulting structure is still valid HTTP. Only valid HTTP structures are forwarded to the Executor.

3. The Executor receives the mutated seed as a simple byte-array. This is parsed to an ASCII string.

4. The seed is preprocessed by the fuzzer:

(a) If the seed contains any dynamic variables, these are extracted from the seed and resolved. In the first seed, no dynamic variables are encoded, meaning that this step is skipped.

(b) Any properties that are encoded to be resolved as a random variable are resolved. This is to be able to have parameters that strictly require to be unique for every execution.

5. To determine whether the seed should be executed, the seed is partially parsed to extract the HTTP method and path. This value combination is used as a key in a map that stores the probability of execution. Initially, all probabilities are set to 1. Therefore, this seed is passed along and executed.

6. Finally, the seed is completely parsed to an HTTP structure. Completely valid HTTP structures are used to configure an HTTP library that executes HTTP requests.

7. The request is executed on the PUT, after which the response is collected. This response may trigger several processes, depending on the content of the response:

   (a) If a resource is created, updated or deleted, the State is updated with this new information. This new information about this resource is either received directly in the response or is retrieved later by executing another GET request using information from the *Location* header. In the example execution of Seed 1, the State is updated with the newly created resource. Note that the ID is fictional since it is a random UUID, generated by the server. The State now contains a key *author/92db40b7-9768-441f-bde9-c6fcdea2fedb* which maps to the Author resource

```
{
  "id": "92db40b7-9768-441f-bde9-c6fcdea2fedb",
  "name":"author_name",
  "authorType":"POET",
  "externalPropertySource":"http://127.0.0.1:8090/author_property
     ↪ ",
  "externalProperty": null,
  "books": []
}
```

   (b) If the response contains an interesting status code, a record is made of this execution that contains the entire trace that has led to this exact state. In the example of Seed 1, the execution results in a status code of 200, which is not interesting.

8. Finally, information about the execution is requested from the Java agent using the TCP request/reply interface:

   (a) Coverage feedback, containing the updated coverage map. This information is then used to update LibFuzzer. If the updated coverage map indicates that new branches have been triggered, the execution is stored as an *interesting* execution.

   (b) Information about sinks of which the vulnerability condition is fulfilled

   If the outcome of this execution is not interesting, the probability of executing is reduced. Otherwise, this value is increased by a configurable weight, based on the type of interesting behaviour.

7.1.3 Feedback

During the execution of the request on the PUT, the instrumentation records several things that are sent back to the Executor. This in its turn sends this information back to LibFuzzer, which then uses this information for further iterations of the fuzzing loop.

1. Information about branching statements is recorded by the instrumentation. This is sent back to the Executor using the asynchronous TCP interface. During the execution of Seed 1, the following branches are encountered in the code:

   ```
   if (dto.authorType == AuthorType.NOVELIST)
   ```

   and

   ```
   if (dto.authorType == AuthorType.POET && dto.name == "Hackerman")
   ```

   In the second conditional, the second part is only evaluated when the first part evaluates to true. Two pairs of comparisons are recorded for feedback: (POET, NOVELIST) and (author_name, Hackerman). After processing this feedback, the CMPDict of LibFuzzer now contains these two pairs.

2. Information about triggered sinks is recorded. During the execution of Seed 1, no sinks are triggered.

3. Finally, coverage information is recorded. This is stored within a Coverage map on the Java agent, which is made available using a TCP request/reply interface. In the example of Seed 1, new branches in the code are triggered, thus increasing the coverage.

4. All feedback, aggregated with the HTTP execution results, is considered in classifying an execution as (non) interesting. Since Seed 1 increases coverage, a record is made that contains the entire trace:

   ```
   [TRACE] POST /author (creating resource /author/92db40b7-9768-441f-bde9-
   ↪ c6fcdea2fedb)
   ```

7.2 Fuzzing loop

In the next iteration, Seed 2 is used and the updated state of both LibFuzzer and the State is used. Only the steps that are different from the first iteration are denoted here.

2 The mutation of this seed may use the CMPDict with the two newly added pairs. Since no value of the seed corresponds to the already seen pairs, no value can be chosen based on these values. However, by random selection, LibFuzzer can select one of the four values.

4 The seed contains a dynamic variable *$${authorId}$$*. This is configured to map to a resource key */author/{id}*. This is dynamically resolved by querying the State for a resource that corresponds to this key. In this case, only the resource *author/92db40b7-9768-441f-bde9-c6fcdea2fedb* corresponds, of which the id is used as the value for this dynamic variable. This results in the processed seed:

```
POST /book HTTP/1.1
Connection: close
Content-Length: 113
Host: localhost
Content-Type: application/json

{"name":"book_name","bookType":"PAPERBACK","authorId":"92db40b7-9768-441f-
    ↪ bde9-c6fcdea2fedb","isbnSource":"http://127.0.0.1:8090/isbn"}
```

7a The state is updated with a new key *book/454800cc-0ab6-4514-ad99-2c0ea97f2d39* and value:

```
{
  "id": "454800cc-0ab6-4514-ad99-2c0ea97f2d39",
  "name":"book_name",
  "bookType":"PAPERBACK",
  "authorId":"92db40b7-9768-441f-bde9-c6fcdea2fedb",
  "isbnSource":"http://127.0.0.1:8090/isbn"
}
```

Also, the following feedback is collected:

1 The following branches are triggered:

```
if (dto.bookType == BookType.PAPERBACK)
```

```
if (dto.bookType == BookType.HARDCOVER && author.name == "Hackerman++")
```

2 Since the first branch is executed, an external call is made. This triggers a sink on the class *org.springframework.web.client.RestTemplate*. This sink records the input parameter *http://127.0.0.1:8090/isbn* and sends a pair back to LibFuzzer. In this pair, possible values that could exploit this sink are inserted. For example, the pair (http://127.0.0.1:8090/isbn, 127.0.0.1:8091) is inserted. This potentially triggers the vulnerable behaviour of the sink.

4 The trace of this execution contains a sequence of requests:

```
[TRACE] POST /author (creating resource /author/92db40b7-9768-441f-bde9-
    ↪ c6fcdea2fedb)
[TRACE] POST /book (using dynamic variable from /author/92db40b7-9768-441f
    ↪ -bde9-c6fcdea2fedb)
```

8 Experiments

To validate whether the implementation is applicable to more realistic applications, a target PUT is selected. Keycloak (v12.0.1) [35] is used since it provides a REST API, runs as a Java-based server application and has a list of known CVEs [36]. This makes it a suitable candidate for this research.

Two fuzzer implementations are used to target this PUT, which are then compared on efficiency and effectiveness. Efficiency is measured by the increase of coverage over time, where effectiveness is measured by the absolute amount of coverage obtained during execution. The importance of effectiveness here is in detecting whether a fuzzer is able to find deeper code paths within the PUT to reach a higher coverage. This can be at the cost of efficiency, but could overall be beneficial to the fuzzing goal. Additional to comparing both metrics on coverage, both fuzzers have a similar approach to finding bugs in the application. This allows us to compare the efficiency and effectiveness of these fuzzers in terms of the total number of unique bugs found.

Since RESTler is a black-box fuzzer, it is expected that a grey-box fuzzer is capable of reaching higher percentages of code coverage, potentially at the cost of efficiency. Also, while both fuzzers have the same approach to classifying a bug in an application, these bugs may be triggered for different reasons. Since RESTler focuses more on sending valid data that corresponds to the OpenAPI specification, less invalid data is sent in comparison to GRFUZZ, which uses LibFuzzer's mutation operators. This inevitably results in a larger input space of invalid data. For that reason, it is to be expected that GRFUZZ finds more bugs that are related to input validation.

8.1 Setup

To be able to compare the results of the research with RESTler, the application is instrumented using Java Code Coverage (JaCoCo). JaCoCo is already part of the Jazzer Java Agent and this allows for collecting metrics on code coverage. Having the same implementation is crucial here to be able to make any conclusions about the results. A small detail is that the JaCoCo agent instruments all classes that match the respective configuration, whereas the Jazzer Java Agent instruments only classes that are loaded onto the JVM by a ClassLoader. To circumvent this, the Jazzer Java agent is extended such that it loads all classes during startup. The result of this is an instrumented set of classes that add up to roughly 460K lines of code, containing 36K branching statements.
The PUT is then subdued to both fuzzers while collecting coverage metrics during execution. This coverage is computed cumulatively, resulting in a time series of coverage over time. Besides recording information about coverage, both fuzzers also record executions that result in an invalid HTTP status code (500). These results are also compared.
The experiments are run on a Dell Latitude with an Intel i7-7600U with 16 GB of RAM. These experiments are run 5 times each and for durations of 15 minutes, 1 hour and 5 hours.
Due to time constraints, only a subset of the API is fuzzed; a set of 47 endpoints (See Appendix 9), which contain dependencies between requests and known CVEs. This set is selected, such that all dependencies within this set can be fulfilled.

8.1.1 RESTler setup

To set up a fuzzing environment for RESTler, an OpenAPI specification is required to generate the grammar for the fuzzer. This specification is lacking for most of the Keycloak versions and an un-

official API specification is used [37]. By manually checking the generated grammar, the OpenAPI specification is corrected, such that it matches the current version.
While setting up RESTler, some limitations of RESTler became apparent: The generated grammar is insufficient to fuzz Keycloak correctly; none of the endpoints work out of the box. Some of these issues can be resolved by providing additional information in the OpenAPI specification, such as data model examples or constraints for specific fields. Constraints that are enforced by validation on the server-side, such as enumeration values, or boundaries on input length. Not all of these constraints can be encoded in an OpenAPI specification, which requires a tester to provide additional logic to provide RESTler with information about these constraints. Value generators can be used for specific fields of specific requests, which would allow a tester to encode all constraints of the application. Since RESTler is a grammar-based black-box fuzzer, having a correct grammar is crucial here. Without feedback on why a request fails, RESTler has to make assumptions on what could be a valid input, based on this grammar. Some examples of this are generators that provide valid UUID values, which are used for resource identifiers, or generators that generate random string values. Since RESTler uses dictionaries to generate values for specific types of fields, this does not guarantee that these values are unique. By enriching these dictionaries, a larger set of values is used by RESTler, which could be helpful during fuzzing. This however conflicts with the concept of black-box fuzzing, which now requires in-depth program analysis to provide RESTler with a grammar that results in successful requests.
Another defect of RESTler is that it does not consider resource identifiers that are returned by sending a *Location* header on resource creation. This is common in REST and also how resources are created in Keycloak. To support this, RESTler provides a way to annotate headers in the OpenAPI spec, which can then be used as value providers during the fuzzing execution. Some additional logic has to be implemented here to extract resource identifiers from the full URI which is returned in the *Location* header. Note that this is a minor issue, which will eventually be implemented by RESTler.

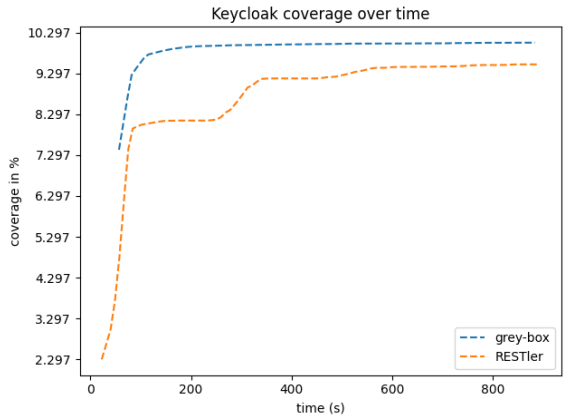
8.2 Results

In Figure 8 the efficiency and effectiveness of both fuzzers are displayed. The first observation that can be made is that GRFUZZ outperforms RESTler in terms of effectiveness and efficiency; it is able to reach higher coverage, in a shorter amount of time. Both fuzzers display similarities in the curves, where the largest part of the coverage is achieved in the initial phase. RESTler does have points in time where a larger increase is achieved, however, this is explainable by how RESTler runs its fuzzing campaign; sequences are scheduled based on their length, which results in generations of length n. An n-th generation will only be able to execute sequences that contain n number of requests. RESTler executes these generations in sequence, which explains the sudden peak after a certain duration.

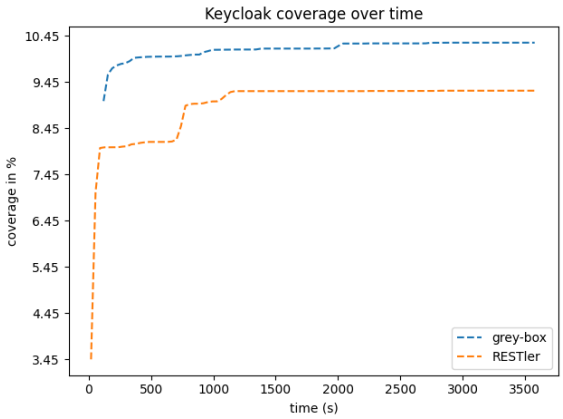
Table 3: # of unique bugs found

duration	fuzzer	
	RESTler	GRFUZZ
15 min.	0	8
1 hour	6	10
5 hours	6	12

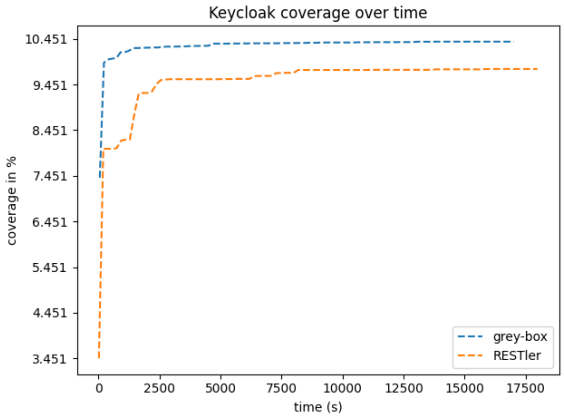
The bugs that are found by both fuzzers can all be classified as input validation exceptions. For



(a) 15 minutes



(b) 1 hour



(c) 5 hours

Figure 8: Keycloak coverage over time

some input validations, i.e. forbidden characters, the default response of Keycloak is to reply with an internal server error. Except for one, none of the found issues seem to be related to any underlying vulnerability. These found issues are aggregated to an absolute number of unique bugs found. This is displayed in Table 3. 4 out of 6 of the issues found by RESTler are also found by this research's fuzzer.

8.3 Additional results

Since RESTler is a black-box fuzzer, it has limited capabilities for detecting and triggering vulnerabilities. In this respect, it is incomparable to this research. However, the findings of this research are significant.

In the context of finding vulnerabilities, effectiveness can also be measured in terms of the number of vulnerabilities found over the total amount of vulnerabilities. Of course, this total amount can only be established by the known vulnerabilities of an application, and therefore does not consider new vulnerabilities. Since this research is focused on finding SSRF vulnerabilities, we can scope the total number of vulnerabilities to a set of known CVEs that classify as SSRF vulnerabilities. Added to this, is the set of unknown vulnerabilities that are found during this research. This adds up to a set of two SSRF vulnerabilities. This now allows us to measure the effectiveness of running the fuzzer for certain amounts of time. Note that these numbers are highly application dependent, but should give

an example of how different fuzzing durations affect the outcome.

8.3.1 SSRF vulnerability

During the experiments, the one known SSRF vulnerability was found, alongside an unknown SSRF vulnerability. The trace of the unknown SSRF vulnerability can be seen in Appendix 8. This trace displays the result of a sequence of three requests:

1. POST /auth/admin/realms
2. POST /auth/admin/realms/{realm}/clients
3. POST /auth/admin/realms/{realm}/push-revocation

For the second and third requests, the *realm* variable of the first POST request is required. The vulnerable sink of this sequence is within the *org.apache.http.impl.client.CloseableHttpClient* class. The *execute* method receives a parameter that is retrieved from a dynamic parameter that is set in the second request. In the third request, the sink is triggered by retrieving this dynamic parameter and constructing a URI to execute an external request. Since this dynamic parameter is controllable by the API user, this user has the possibility to forge vulnerable requests.

The known SSRF vulnerability[1] can be triggered by a single GET request. This request does not depend on any other request and does not use any internal state that is set by previous requests.

Results
The results in Table 4 are extracted from the same experiments as in the previous section. The results display the time of the experiment and the average number of vulnerabilities found over five unique runs. What can be observed from these results is that the more complex, unknown SSRF vulnerability is detected sooner than the simple, known SSRF vulnerability. This is counter-intuitive, but can be explained by some of the methods used in the architecture of GRFUZZ; the known SSRF vulnerability only shows up once in the output of the fuzzer. Although this request is executed a large number of times, these requests are not recorded, since their execution has not led to any new coverage or any new response. Following from this observation is that this will result in a low probability of execution for this specific request. The requests of the unknown SSRF vulnerability, however, show up multiple times in the output of the fuzzer.

Table 4: Vulnerabilities found by GRFUZZ

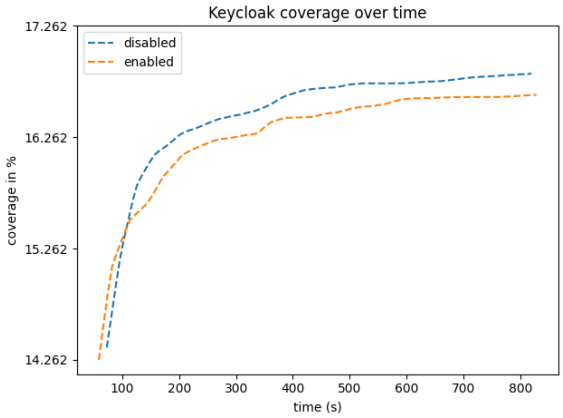
	Unknown SSRF	Known SSRF
Ground truth	1	1
15 min.	0.2	0
1 hour	1	0.6
5 hours	1	1

[1] Keycloak CVE-2020-10770 `https://www.cvedetails.com/cve/\gls{cve}-2020-10770/`

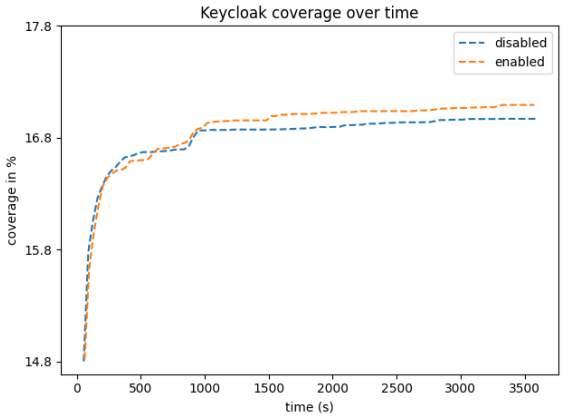
8.3.2 Smart fuzzing

Another improvement GRFUZZ is the technique of smart fuzzing. Although this is a similar technique to *Deferred Parsing* in [28], their results only compare their complete fuzzer, of which this technique is but a small improvement. To compare this method within GRFUZZ, two experiments are run, of which one has this technique disabled and the other one enabled. Again, these experiments are run five times each, for durations of 15 minutes, 1 hour and 5 hours. A similar setup as in Section 8.1 is used, apart from a larger set of endpoints that is being fuzzed. This is similar between the two experiments.

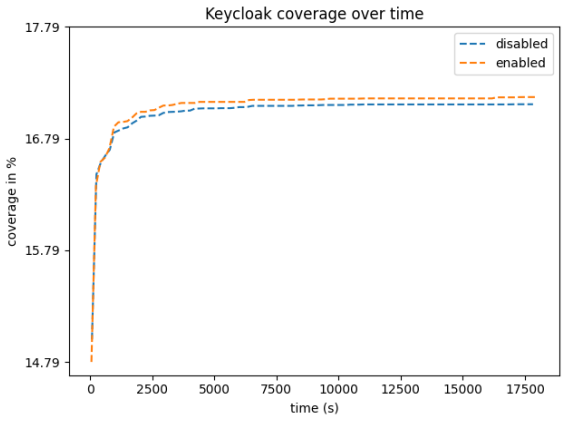
Results
Following from Figure 9 we can see that using this method has a negative impact in the short term, but a slightly positive impact after approximately 15 minutes. This effect is most likely due to the slight overhead that is caused by the implementation of this technique; since this technique is implemented after the seed scheduling and mutation of the grey-box fuzzer, every request that is skipped will have these two steps as additional overhead. Since new paths are found more easily in the early stages of the fuzzing campaign, this overhead is slowing the fuzzer down. However, after these 15 minutes, it can be seen that this technique results in a slight increase in the overall coverage found. By skipping executions that are no longer likely to exhibit interesting behaviour, the fuzzer will focus more on seeds that have a higher chance of finding new paths.



(a) 15 minutes



(b) 1 hour



(c) 5 hours

Figure 9: Keycloak coverage over time, with smart fuzzing enabled and disabled, using GRFUZZ

9 Discussion

By comparing two inherently different fuzzers, it is difficult to make any definitive conclusions about whether certain methods, employed by the fuzzers, are advantageous compared to others. This is also noted by [38]; it is extremely likely that differences found in the results are explicable by different engineering decisions made by the researcher. In this case, RESTler has very few similar properties to this research. Besides being a web-application fuzzer that coheres to RESTful architecture patterns, both implementations are extremely dissimilar: RESTler uses a grammar-based generation strategy, whereas GRFUZZ uses a coverage-guided, based mutation strategy. Also, RESTler is much less random than GRFUZZ; RESTler uses explicit request sequences to exhibit stateful behaviour, whereas this research uses a more random, dynamic approach. For this reason, only a few metrics are considered in the results. Even with these few metrics, it is difficult to produce exactly similar environments that minimize any external influences on the results. For example, for both test setups, JaCoCo is used to record coverage. However, since Jazzer uses the internal API of JaCoCo, and the test setup with RESTler uses the JaCoCo Java agent, there are slight differences. JaCoCo instruments all code on the classpath, whereas the Jazzer Java agent only instruments classes that are loaded by the JVM. This difference is made up for by explicitly loading all classes at runtime using the Jazzer Java agent. Of course, this is not exactly identical to the RESTler test setup. The performance deficit, however, is negligible since memory usage has not proven to be an issue during the fuzzer executions. In short, the differences in test setups are dealt with using a best-effort approach; not all environmental properties can be identical and this has had some impact on the results.

During the experiments of GRFUZZ, some findings came forward due to scalability constraints. One of the key aspects of GRFUZZ's architecture is the handling of feedback, produced by instrumentation. Specifically feedback regarding branching statements. In the small testing application, the number of instructions did not exceed the maximum number of instructions that could be handled by the TCP interface. However, when applying this to Keycloak, the size of the instrumented code became more than a thousand times larger. This inherently results in many more branching statements, thus increasing the number of comparison instructions that are received through feedback. However this number is not linear to the size of the application. It became apparent that not all produced messages could be handled by the TCP interface. This resulted in a message queue being overloaded with messages. By providing an expiration time to these messages, the queue will periodically remove outdated messages, thus preventing itself from overflowing. Of course, this will mean that not all messages will be handled and some information regarding comparison instructions is lost. On average, 10-70% of the messages are lost. This could have a huge impact on the effectiveness of the fuzzer. Other tactics could be used in order to throttle the number of messages that need to be processed during execution, such as throttling the requests executed by the fuzzer.

One of the weaknesses of the white-box techniques used in this setup is that it does not consider the origin of the data. Without these constraints, all branching statements are considered by the fuzzer. This results in a much larger space of values sent as feedback to LibFuzzer, which in its turn becomes less efficient in selecting the values in order to pass branch conditionals. Of course, this is a limitation which came with the decision of using Jazzer. While this provides a lot of useful techniques and utilities, this is one of the limitations. A possible solution to this would have been to filter feedback values based on all information of a request's trace. Unfortunately due to the asynchronous messaging of this feedback channel, this is infeasible; in order to couple feedback to a request's trace, either this feedback needs to be processed synchronously or this feedback needs to have some unique

identifier which allows it to be coupled to a specific request. Synchronous processing would have a huge impact on the efficiency of the fuzzer, whilst the other option does require revisiting the entire instrumentation framework that Jazzer provides.

Finally, the stateful approach in this research was designed by redesigning the stateful approach taken in RESTler. The need to define request sequences explicitly is removed, such that test cases or seeds can be parameterized by their direct dependencies only. RESTler on the other hand can use the information of request sequences to define a minimal approach in covering all paths, increasing the efficiency of the fuzzer. Both approaches have strengths and weaknesses, where the approach in GRFUZZ becomes increasingly more efficient when the number of dependencies increases. However, this also is less efficient due to paths being explored that are already covered by other test cases. While some methods are already used to steer this random process into making more informed decisions on what to do next, there are still many options left to increasing the efficiency of this random process.

10 Future work

Since the area of fuzzing for vulnerabilities in web applications is rather new, the space of possibilities is endless. Within this research, only a limited set of techniques is explored, leaving room for many other improvements.

10.1 White-box methods

Concolic fuzzing and dynamic taint analysis are considered to be white-box methods. However, the concrete implementation of these techniques using LibFuzzer drops some details of these techniques. This makes the actual implementation more grey-box. Although these methods already show some improvements, perhaps these methods could be implemented more exact. The theory behind these techniques seems highly applicable in the context of web applications; in order to achieve a high coverage efficiently, a fuzzer must be aware of how to structure this input and more specifically, which specific input values should be used by the fuzzer. With the complexity of web applications, these techniques are challenging to implement and potentially require a new look at these techniques that considers stateful behaviour.

Additionally, the techniques of smart fuzzing and the white-box techniques have the potential to be integrated. The computed probability of smart fuzzing entails that a certain path is likely to exhibit interesting behaviour. This interesting behaviour is now determined by new coverage, sinks and HTTP status codes. This is all knowledge about prior executions. White-box methods, however, reveal potential interesting behaviour in terms of uncovered paths, encountered during execution. Or potential sinks that could be exploited, with the corresponding payload. This could improve the metric of the probability of exhibiting interesting behaviour.

10.2 Architectural improvements

In this research, the fuzzer architecture is made up of LibFuzzer and an intermediary layer to handle everything concerning HTTP, state and other specific environmental properties of a web application. This architecture is sub-optimal and could be improved by incorporating these two layers into a single fuzzer. By doing so, fewer trade-offs have to be made that are imposed by having a limited interface between the two components. For example, seed scheduling could incorporate feedback, triggered by specific scenarios on the intermediary layer.

10.3 Vulnerability detection methods

As shown in Section 3.3, over half of the OWASP top 10 vulnerabilities can be found using DAST tools. Some existing DAST tools are specifically constructed to find these kinds of vulnerabilities. Since one of the main strengths of a grey-box fuzzer is to achieve high coverage of a PUT, perhaps these two types of tools can be integrated. For example, when the fuzzer encounters a sink that is potentially vulnerable to a SQL injection attack, this knowledge could be shared with a DAST tool that then attempts to exploit this endpoint. Of course, in the context of RESTful web applications, this integration needs to have similar stateful capabilities. Perhaps the stateful approach taken in this research can be fully detached from the fuzzer implementation using interfaces. This could mean that some proxy implementation could handle everything related to state; any DAST tool that needs to be compliant with RESTful properties of a software system simply would have to be compliant with the

interface of this proxy.

Another technique that is common in fuzzer architectures, is the use of *ensemble* fuzzing. By combining multiple heterogenic fuzzers, a more diverse input space can be created. One of the requirements is the capability of dealing with a shared seed corpus. This would mean that there needs to be a unified way of composing seeds. For example, stateful dependencies are encoded in the seeds, but generally, fuzzers are not capable of handling these dynamic properties.

Finally, the proposed technique in this research is widely applicable to many classes of vulnerabilities. For example, a SQLi vulnerability can easily be described by a sink and a condition when it is exploited; a database interface that executes a query could be used to detect whether certain user input is used to execute this query. When this user input can be used to craft certain, valid queries, this interface is most likely vulnerable to a SQLi attack. Any vulnerability that can be described by these two requirements is suitable for this technique.

10.4 Grammar-based mutation

Grammar-based mutation is applied in this research by mutating a specific HTTP structure into another valid HTTP structure. However, the grammar of inputs for web applications is more detailed than this structure. A common approach is to use some form of a grammar specification, such as an OpenAPI specification, in order to reason about valid and invalid mutations. This allows for more complex mutation operators to be used, which mutate an existing seed into another seed that still corresponds to the grammar. Currently, key-pair values in the JSON structure are mutated, but this could be extended such that the entire JSON structure can be mutated.

10.5 Extracting interesting results

With the proposed solution for dealing with stateful properties of a RESTful architecture, retaining traceability of what has led to a specific state is extremely important. The current implementation is rather verbose, where a lot of noise is collected in the output logs. Since multiple executions end up in similar states, it would be possible to minimize the paths toward a specific state. Deduplication could be done by selecting the trace with the smallest path.

11 Conclusion

During this research, an approach is proposed to deal with the stateful properties of a RESTful architecture. This approach is more random than RESTler's approach in dealing with state. The disadvantages of this approach are outweighed by the benefits; comparing GRFUZZ to RESTler, GRFUZZ is able to detect more bugs and vulnerabilities, while also achieving higher coverage in less time. Using lightweight white-box methods, GRFUZZ gains substantial advantages compared to a black-box fuzzer. Specifically, the capabilities of detecting vulnerabilities show promising results: in large, real-world applications, complex vulnerabilities can be found using dynamic taint analysis. During this research, implementations have been proposed for detecting SSRF vulnerabilities, which could be applied to detect other types of significant vulnerabilities as well.

Secondly, we propose a solution that incorporates the feedback from the PUT into scheduling decisions. Whilst the results are not groundbreaking, the results still show a slight positive effect on the efficiency of the fuzzer. Since this technique is somewhat limited because of design decisions, this effect potentially becomes more apparent when it is fully incorporated into the fuzzer's architecture.

During this research, it became apparent that this field is rather unexplored. This left many options on the table, of which only a few have been implemented in the proposed architecture. In the future work section, a few of these other options are elaborated on; further research can be done by implementing white-box methods to increase the effectiveness of the fuzzer.

Bibliography

[1] "Owasp." https://owasp.org/. Accessed: 2022-02-01.

[2] "Owasp top 10:2021." https://owasp.org/Top10/. Accessed: 2022-04-12.

[3] "What is devsecops?." https://www.redhat.com/en/topics/devops/what-is-devsecops. Accessed: 2022-04-12.

[4] "Owasp dependency-check project — owasp." https://owasp.org/www-project-dependency-check/. Accessed on 2022-05-03.

[5] "Sonarqube covers the owasp top 10 — sonarqube." https://www.sonarqube.org/features/security/owasp/. Accessed on 2022-05-03.

[6] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, p. 32–44, dec 1990.

[7] F. Gauthier, B. Hassanshahi, B. Selwyn-Smith, T. N. Mai, M. Schlüter, and M. Williams, "Backrest: A model-based feedback-driven greybox fuzzer for web applications," *CoRR*, vol. abs/2108.08455, 2021.

[8] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *ICSE 2019*, November 2019.

[9] "Owasp zap - fuzzing." https://www.zaproxy.org/docs/desktop/addons/fuzzer/. Accessed: 2022-04-12.

[10] "Using burp intruder - portswigger." https://portswigger.net/burp/documentation/desktop/tools/intruder/using#fuzzing-for-vulnerabilities. Accessed: 2022-04-12.

[11] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology*, vol. 2, pp. 115–150, May 2002.

[12] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," 2018.

[13] "Gitlab.org / security-products / protocol-fuzzer-ce · gitlab." https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce. Accessed: 2022-04-12.

[14] "Mozillasecurity/funfuzz: A collection of fuzzers in a harness for testing the spidermonkey javascript engine." https://github.com/MozillaSecurity/funfuzz. Accessed: 2022-04-12.

[15] "Endava/cats." https://github.com/Endava/cats. Accessed: 2022-04-12.

[16] P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," 01 2008.

[17] J. Li, X. Xu, L. Liao, and L. Li, "Concolic execute fuzzing based on control-flow analysis," in *2015 11th International Conference on Computational Intelligence and Security (CIS)*, pp. 385–389, 2015.

[18] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), p. 337–340, Springer-Verlag, 2008.

[19] B. Chess and J. West, "Dynamic taint propagation: Finding vulnerabilities without attacking," *Information Security Technical Report*, vol. 13, no. 1, pp. 33–39, 2008.

[20] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," 02 2017.

[21] "american fuzzy lop." `https://lcamtuf.coredump.cx/afl/`. Accessed on 2022-05-03.

[22] "libfuzzer – a library for coverage-guided fuzz testing." `https://llvm.org/docs/LibFuzzer.html`. Accessed on 2022-05-03.

[23] W. Al-Kahla, A. S. Shatnawi, and E. Taqieddin, "A taxonomy of web security vulnerabilities," in *2021 12th International Conference on Information and Communication Systems (ICICS)*, pp. 424–429, 2021.

[24] "Cwe - cwe-209: Generation of error message containing sensitive information." `https://cwe.mitre.org/data/definitions/209.html`. Accessed on 2022-05-04.

[25] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Checking security properties of cloud services rest apis," Tech. Rep. MSR-TR-2019-1, Microsoft, February 2019. Revised version published in ICST'2020, March 2020.

[26] "Openrce/sulley: A pure-python fully automated and unattended fuzzing framework." `https://github.com/OpenRCE/sulley`. Accessed: 2022-04-12.

[27] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," 2022.

[28] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *CoRR*, vol. abs/1811.09447, 2018.

[29] P. Godefroid, B.-Y. Huang, and M. Polishchuk, "Intelligent rest api data fuzzing," Tech. Rep. MSR-TR-2019-37, Microsoft, November 2019. Revised version published in ESEC/FSE'2020, November 2020.

[30] "Github - codeintelligencetesting/jazzer - coverage-guided, in-process fuzzing for the jvm." `https://github.com/CodeIntelligenceTesting/jazzer`. Accessed on 2022-04-22.

[31] "Structure aware fuzzing." `https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md`. Accessed on 2022-05-04.

[32] "Apache httpclient 4.5.13 api." `https://hc.apache.org/httpcomponents-client-4.5.x/current/httpclient/apidocs/`. Accessed: 2022-02-23.

[33] "Rfc 2616 - hypertext transfer protocol – http/1.1." `https://datatracker.ietf.org/doc/html/rfc2616#section-6`. Accessed: 2022-05-03.

[34] "A10 server side request forgery (ssrf) - owasp top 10:2021." `https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_(SSRF)/`. Accessed on 2022-05-04.

[35] "Keycloak." https://www.keycloak.org/. Accessed: 2022-04-20.

[36] "Redhat keycloak : List of security vulnerabilities." https://github.com/ccouzens/keycloak-openapi. Accessed on 2022-05-04.

[37] "Github - ccouzense/keycloak-openapi - openapi definitions for keycloak's admin api." https://github.com/ccouzens/keycloak-openapi. Accessed on 2022-05-04.

[38] M. Boehme, C. Cadar, and A. ROYCHOUDHURY, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021.

A Code snippets

Listing 7: Regex Solver

```
    fun generate(p: Pattern, prefix: String = "", min: Int = 0, max: Int =
        ↪ 50): String?
    {
        if (min <= 0 && max >= 0 && p.matcher(prefix).matches()) return prefix
        if (max <= 0) return null
        chars.shuffle()
        chars.forEach { c ->

            val m: Matcher = p.matcher(prefix + c)
            if (m.matches() || m.hitEnd())
            {
                return generate(p, prefix + c, min - 1, max - 1) ?:
                    ↪ return@forEach
            }
        }
        return null
    }
```

Listing 8: Example trace

```
Reason(s):
Exception has been found during execution.
Illegal HTTP Status has been found.
------------------
Previous coverage: 12566
New coverage: 12566
Input data:
POST $$/auth/admin/realms/{realm}/push-revocation$$ HTTP/1.1
Connection: close
Host: localhost
Content-Type: application/json


------------------
Processed data:
POST /auth/admin/realms/ACCOUNT/push-revocation HTTP/1.1
Connection: close
Host: localhost
Content-Type: application/json


------------------
Trace:
[TRACE] Wed Mar 02 18:45:14 CET 2022
```

```
Input:
POST /auth/admin/realms HTTP/1.1
Connection: close
Content-Length: 1210
Host: localhost
Content-Type: application/json

{"id":"e9d500d8-4beb-4f82-a992-545f51160f81","realm":"ACCOUNT","notBefore":0,"
    ↪ revokeRefreshToken":false,"refreshTokenMaxReuse":0,"accessTokenLifespan
    ↪ ":300,"accessTokenLifespanForImplicitFlow":899,"ssoSessionIdleTimeout
    ↪ ":1800,"ssoSessionMaxLifespan":36000,"ssoSessionIdleTimeoutRememberMe
    ↪ ":0,"ssoSessionMaxLifespanRememberMe":0,"offlineSessionIdleTimeout
    ↪ ":2592000,"offlineSessionMaxLifespanEnabled":false,"
    ↪ offlineSessionMaxLifespan":5184000,"clientSessionIdleTimeout":0,"
    ↪ clientSessionMaxLifespan":0,"accessCodeLifespan":60,"
    ↪ accessCodeLifespanUserAction":300,"accessCodeLifespanLogin":1800,"
    ↪ actionTokenGeneratedByAdminLifespan":43200,"
    ↪ actionTokenGeneratedByUserLifespan":300,"enabled":true,"sslRequired":"
    ↪ external","registrationAllowed":false,"registrationEmailAsUsername":
    ↪ false,"rememberMe":false,"verifyEmail":false,"loginWithEmailAllowed":
    ↪ true,"duplicateEmailsAllowed":false,"resetPasswordAllowed":false,"
    ↪ editUsernameAllowed":false,"bruteForceProtected":false,"permanentLockout
    ↪ ":false,"maxFailureWaitSeconds":900,"minimumQuickLoginWaitSeconds":60,"
    ↪ waitIncrementSeconds":60,"quickLoginCheckMilliSeconds":1000,"
    ↪ maxDeltaTimeSeconds":43200,"failureFactor":30,"defaultRoles":["
    ↪ offline_access","uma_authorization"]}

Response:
Request URL: http://localhost:8080/auth/admin/realms
Request Method: POST
Status Code: 201
:status: 201
content-length: 0
date: Wed, 02 Mar 2022 17:45:20 GMT
location: http://localhost:8080/auth/admin/realms/ACCOUNT
referrer-policy: no-referrer
strict-transport-security: max-age=31536000; includeSubDomains
x-content-type-options: nosniff
x-frame-options: SAMEORIGIN
x-xss-protection: 1; mode=block


[TRACE] Wed Mar 02 18:54:51 CET 2022
Input:
POST /auth/admin/realms/ACCOUNT/clients HTTP/1.1
Connection: close
Host: localhost
```

```
Content-Type: application/json
Content-Length: 991

{"clientId":"{ ","name":" ","adminUrl":"http://0.0.0.0:8091/","
    ↪ alwaysDisplayInConsole":false,"access":{"view":true,"configure":true,"
    ↪ manage":true},"attributes":{},"authenticationFlowBindingOverrides":{},"
    ↪ authorizationServicesEnabled":false,"bearerOnly":false,"
    ↪ directAccessGrantsEnabled":true,"enabled":true,"protocol":"openid-
    ↪ connect","description":"rest-api","rootUrl":"${authBaseUrl}","baseUrl
    ↪ ":"/realms/heroes/account/","surrogateAuthRequired":false,"
    ↪ clientAuthenticatorType":"client-secret","defaultRoles":["manage-account
    ↪ ","view-profile"],"redirectUris":["/realms/heroes/account/*"],"
    ↪ webOrigins":[],"notBefore":0,"consentRequired":false,"
    ↪ standardFlowEnabled":true,"implicitFlowEnabled":false,"
    ↪ serviceAccountsEnabled":false,"publicClient":false,"frontchannelLogout":
    ↪ false,"fullScopeAllowed":false,"nodeReRegistrationTimeout":0,"
    ↪ defaultClientScopes":["web-origins","role_list","profile","roles","email
    ↪ "],"optionalClientScopes":["address","phone","offline_access","
    ↪ microprofile-jwt"]}

Response:
Request URL: http://localhost:8080/auth/admin/realms/ACCOUNT/clients
Request Method: POST
Status Code: 201
:status: 201
content-length: 0
date: Wed, 02 Mar 2022 17:54:56 GMT
location: http://localhost:8080/auth/admin/realms/ACCOUNT/clients/615ecee0-6fc4
    ↪ -4738-887f-5390c2914284
referrer-policy: no-referrer
strict-transport-security: max-age=31536000; includeSubDomains
x-content-type-options: nosniff
x-frame-options: SAMEORIGIN
x-xss-protection: 1; mode=block


[TRACE] Wed Mar 02 18:56:07 CET 2022
Input:
POST /auth/admin/realms/ACCOUNT/push-revocation HTTP/1.1
Connection: close
Host: localhost
Content-Type: application/json



Response:
Request URL: http://localhost:8080/auth/admin/realms/ACCOUNT/push-revocation
```

```
Request Method: POST
Status Code: 500
:status: 500
content-length: 25
content-type: application/json
date: Wed, 02 Mar 2022 17:56:08 GMT
referrer-policy: no-referrer
strict-transport-security: max-age=31536000; includeSubDomains
x-content-type-options: nosniff
x-frame-options: SAMEORIGIN
x-xss-protection: 1; mode=block

{"error":"unknown_error"}

------------------
Exceptions:
Server Side Request Forgery detected on org.apache.http.impl.client.
    ↪ CloseableHttpClientcom.code_intelligence.jazzer.api.
    ↪ FuzzerSecurityIssueHigh: Server Side Request Forgery detected on org.
    ↪ apache.http.impl.client.CloseableHttpClient
        at com.code_intelligence.jazzer.api.ExceptionUtil.handleSecurityIssue(
            ↪ ExceptionUtil.java:22)
        at com.code_intelligence.jazzer.sanitizers.ApacheHttpClientSSRF.
            ↪ hookHttpClientPostDoExecute(ApacheHttpClientSSRF.kt:124)
        at org.apache.httpcomponents.core//org.apache.http.impl.client.
            ↪ CloseableHttpClient.execute(CloseableHttpClient.java:83)
        at org.apache.httpcomponents.core//org.apache.http.impl.client.
            ↪ CloseableHttpClient.execute(CloseableHttpClient.java:108)
        at org.keycloak.keycloak-services@12.0.1//org.keycloak.connections.
            ↪ httpclient.DefaultHttpClientFactory$1.postText(
            ↪ DefaultHttpClientFactory.java:86)
        at org.keycloak.keycloak-services@12.0.1//org.keycloak.protocol.oidc.
            ↪ OIDCLoginProtocol.sendPushRevocationPolicyRequest(
            ↪ OIDCLoginProtocol.java:400)
        at org.keycloak.keycloak-services@12.0.1//org.keycloak.services.
            ↪ managers.ResourceAdminManager.sendPushRevocationPolicyRequest(
            ↪ ResourceAdminManager.java:344)
        at org.keycloak.keycloak-services@12.0.1//org.keycloak.services.
            ↪ managers.ResourceAdminManager.pushRevocationPolicy(
            ↪ ResourceAdminManager.java:327)
        at org.keycloak.keycloak-services@12.0.1//org.keycloak.services.
            ↪ managers.ResourceAdminManager.lambda$pushRealmRevocationPolicy$1(
            ↪ ResourceAdminManager.java:304)
        ....
------------------
```

Listing 9: List of Keycloak endpoints, covered in experiments

```
GET /realms
POST /realms
GET /realms/{realm}
PUT /realms/{realm}
GET /realms/{realm}/client-scopes/
POST /realms/{realm}/client-scopes/
GET /auth/admin/realms/{realm}/client-scopes/{clientScopeId}
PUT /auth/admin/realms/{realm}/client-scopes/{clientScopeId}
DELETE /auth/admin/realms/{realm}/client-scopes/{clientScopeId}
GET /auth/admin/realms/{realm}/admin-events
DELETE /auth/admin/realms/{realm}/admin-events
GET /auth/admin/realms/{realm}/clients
POST /auth/admin/realms/{realm}/clients
GET /auth/admin/realms/{realm}/groups
POST /auth/admin/realms/{realm}/groups
GET /auth/admin/realms/{realm}/groups/count
GET /auth/admin/realms/{realm}/groups/{groupId}
PUT /auth/admin/realms/{realm}/groups/{groupId}
DELETE /auth/admin/realms/{realm}/groups/{groupId}
POST /auth/admin/realms/{realm}/groups/{groupId}/children
GET /auth/admin/realms/{realm}/groups/{groupId}/management/permissions
PUT /auth/admin/realms/{realm}/groups/{groupId}/management/permissions
GET /auth/admin/realms/{realm}/groups/{groupId}/members
GET /auth/admin/realms/{realm}/groups/{groupId}/role-mappings
GET /auth/admin/realms/{realm}/groups/{groupId}/role-mappings/clients/{clientId
    ↪ }
POST /auth/admin/realms/{realm}/groups/{groupId}/role-mappings/clients/{
    ↪ clientId}
DELETE /auth/admin/realms/{realm}/groups/{groupId}/role-mappings/clients/{
    ↪ clientId}
GET /auth/admin/realms/{realm}/groups/{groupId}/role-mappings/clients/{clientId
    ↪ }/available
GET /auth/admin/realms/{realm}/groups/{groupId}/role-mappings/clients/{clientId
    ↪ }/composite
GET /auth/admin/realms/{realm}/groups/{groupId}/role-mappings/realm
POST /auth/admin/realms/{realm}/groups/{groupId}/role-mappings/realm
DELETE /auth/admin/realms/{realm}/groups/{groupId}/role-mappings/realm
GET /auth/admin/realms/{realm}/groups/{groupId}/role-mappings/realm/available
GET /auth/admin/realms/{realm}/roles
POST /auth/admin/realms/{realm}/roles
GET /auth/admin/realms/{realm}/roles/{roleName}
PUT /auth/admin/realms/{realm}/roles/{roleName}
DELETE /auth/admin/realms/{realm}/roles/{roleName}
GET /auth/admin/realms/{realm}/roles/{roleName}/composites
POST /auth/admin/realms/{realm}/roles/{roleName}/composites
DELETE /auth/admin/realms/{realm}/roles/{roleName}/composites
```

```
GET /auth/admin/realms/{realm}/roles/{roleName}/composites/clients/{clientId}
GET /auth/admin/realms/{realm}/roles/{roleName}/composites/realm
GET /auth/admin/realms/{realm}/roles/{roleName}/groups
GET /auth/admin/realms/{realm}/roles/{roleName}/management/permissions
PUT /auth/admin/realms/{realm}/roles/{roleName}/management/permissions
GET /auth/admin/realms/{realm}/roles/{roleName}/users
```