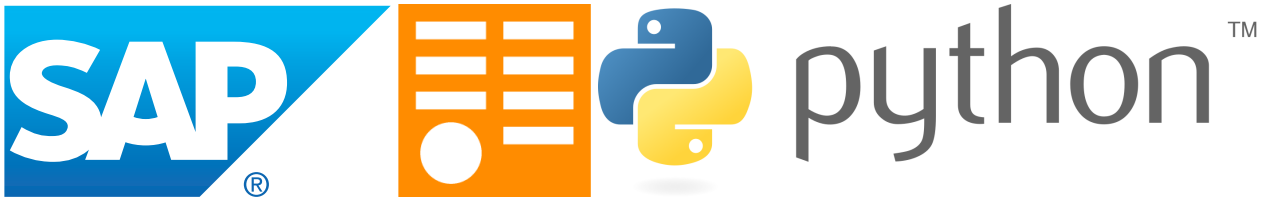
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2018
http://excel.fit.vutbr.cz

Fuzz Testing of Applications Communicating via the OData Protocol

Ľuboš Mjachky



Abstract
Delivering stable and reliable software to customers is difficult. Applications are prone to errors, regardless of the experience level of the developers. Automated testing methods and tools are heavily used in all phases of development life-cycle to reduce chances of bugs escaping to the users. This paper's goal is to design an intelligent and automated testing tool which is able to test business applications. A mishandled action performed within such an application may cause a failure with disastrous consequences. To simulate these actions, one can use testing where test cases contain invalid or random data. This testing technique is called fuzzing or fuzz testing, and it involves providing malformed or mutated data as an input to the program. The proposed tool, namely ODfuzz, is a fuzzing tool ready to test applications communicating via the OData protocol which is a protocol built on existing HTTP and REST methodologies. ODfuzz is generating and fuzzing requests that are to be sent to the server. The requests contain mutated data that pass through various code paths and may result into an application error. ODfuzz was used to test back-end modules of modern SAP applications written in the ABAP language. ODfuzz is also suited for testing any other applications that communicate via the OData protocol as well.

Keywords: SAP — OData — Fuzzing — ABAP — Genetic loop

Supplementary Material: *N/A*

xmjach00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Applications that are built on top of the database and form a business logic layer are often dealing with confidential data. This leads developers to design programs that have to implement security standards. Even with adopting these standards, the application may behave suspiciously in various corner cases. If we talk about corner cases, we mean that such a case occurs once a year but it can result in a data loss or breach. Every serious software is heavily tested to prevent similar scenarios. However, modern applications are very complex, so manual testing is not sufficient.

The aim of this paper is to design and implement an automated tool that is capable of testing applications communicating via the OData protocol. OData protocol is widely used, for instance, in diverse SAP products. It is used for consuming data and represents a mediator between the back-end and front-end.

The designed tool uses a testing technique based on providing mutated or random data to the application's input, also called fuzzing or fuzz testing. The fuzzing provides several advantages over traditional testing methods because it can be easily automated and used in all development phases. A tool that automates the

whole fuzzing process is named a fuzzer.

Despite the fact that the terms fuzzing and OData were introduced a long time ago, there is only one application that exists in the context of OData fuzzing, and it is called *Oyedata*[1]. *Oyedata* is a black-box tester written in C#. It is able to create various templates for generating fuzzed data. The templates are then filled with random data and can be sent to the desired server. However, this tool is not convenient for an extensive testing due to its limited usability. The application has to be operated manually by a human in order to send requests and read responses.

The fuzzer proposed in this work, namely *ODfuzz*, is a client application that sends requests and receives responses from the server. Requests usually contain malformed or randomly generated data which may result afterwards in a server crash on account of an incorrect input handling.

Generating new data without any diagnostics or evaluation appears to be highly inefficient. Therefore, server responses are stored in a NoSQL database and considered for the further analysis. The designed tool uses evolutionary algorithm, in particular, a genetic loop, to create new test cases based on the success of previous. The tests that produced an error are evaluated by a fitness function with a better score and used in next iteration.

ODfuzz was programmed in the Python language and can be run both on Linux and Windows systems. Results of the fuzzer, HTTP responses, tested entities and properties are aggregated and displayed by JavaScript's open-source Pivot Table implementation.

2. Theoretical Background

In this section we will discuss basic principles of fuzzing, its advantages and disadvantages, and the OData protocol.

2.1 Fuzzing

Fuzzing is a negative testing technique that uses invalid or random data as an input for the tested program in order to discover bugs and other vulnerabilities [1].

In fuzzing, it does not matter what type of software is tested, the following steps are always made [2]:

1. Target identification. Fuzzing targets are generally applications that are reading user input, running in a privileged regime or working with the personal information.
2. Identification of input vectors. Applications take input in numerous formats. The most errors were triggered in the past by an inadequate input validation or sanitization. The identification of input vectors is the key to the success because if the program refuses to allocate memory as a result of a wrong data structure, the whole testing process is limited.
3. Generating fuzzed data. The input data should be generated to reasonably sample entire input state space. To accomplish this, heuristic methods are applied in fuzzers. Fuzzed data are simply data which were created randomly or by mutating existing data. To mutate data usually means to change the byte ordering, alter strings, or flip bits.
4. Executing fuzzed data. Fuzzed data are forwarded to the application's input. This process is fully automated and may include file opening, packet sending, etc.
5. Program monitoring. The program is monitored via process managers, debuggers, or sanitizers. If the tested application runs on a server and we do not have an access to the server, obviously, we cannot monitor the program itself. On the other hand, we can track responses or a performance degradation.
6. Vulnerabilities determination. The last phase of fuzzing is an analysis of the detected bugs and vulnerabilities. Automated collection or aggregation of caught exceptions allows developers to review less reports and focus their attention to more relevant threats.

Fuzzers are divided into two groups. Generation-based and mutation-based fuzzers. Generation-based fuzzers are tied to a particular protocol, application, or file format. These fuzzers are creating new feeds by definitions from its own resources. Code coverage of the tested program basically depends on knowledge of data structures. Mutation-based fuzzers use valid data and modify them. A disadvantage of such fuzzers lies in low functional code coverage. Mutation-based fuzzers test especially code that parses system input [3].

There are a lot of fuzzers available on the market that are able to recognize valid input vectors, generate and execute fuzzed data, and monitor programs. Those are, for example, American fuzzy lop (AFL), Vuzzer, and Peach.

[1] https://www.mcafee.com/us/downloads/free-tools/oyedata.aspx

2.2 The OData Protocol

The main reason to design the OData protocol was to simplify data sharing across disparate applications in enterprises and the cloud. OData is a web protocol built on existing *HTTP* and *REST* methodologies. It uses *HTTP POST*, *GET*, *PUT*, and *DELETE* methods to create, read, update, and delete data, respectively. OData was developed by Microsoft and has been standardized by *OASIS* in 2014 [4].

In OData, data resources are accessible via a Uniform Resource Identifier (URI). The URI is composed of a service name, a resource path, and queries, e.g.:

```
https://svc.com/Categories(15)/Products?$filter=ID lt 8
|−−Service−−|−−−Data source−−−|−−Query−−|
```

The URI mentioned above returns products that are attached to the category identified by number 15 and meet the filter query restriction where the product ID is less than 8. OData also supports $orderby (ordering data by its property values), $top (requesting only top N results), $skip (skipping set of results), $expand (expanding associated entities), $select (projecting only subset of properties), and other query options.

The filter query option supports the use of various types of operators and functions within one query. Table 1 below displays complete list of all operators.

Type	Operator
Logical	eq, ne, gt, lt, le, ge, and, or, not
Arithmetic	add, sub, mul, div, mod
Grouping	(,)

Table 1. Operators of the filter query option

Therefore, it is possible to create queries like:

```
Products?$filter=ID gt 10 and Price lt 50 and Price gt 20
Items?$filter=startswith(Description, 'Beverage') eq true
Items?$filter=(ID eq 1 or ID eq 2) and Price lt 20
```

Data in OData are represented in the Entity Data Model. The structure of entities and associations is declared in a metadata document in the XML format. An example of metadata is shown in Listing 1. There is defined an entity type with two properties *OrderID* and *Subtotal* and one key element *OrderID*.

```
<EntityType Name="Order_Subtotal">
    <Key>
        <PropertyRef Name="OrderID"/>
    </Key>
    <Property Name="OrderID" Type="Edm.Int32"
        Nullable="false"/>
    <Property Name="Subtotal" Type="Edm.Decimal"
        Nullable="true" Precision="19" Scale="4"/>
</EntityType>
```

Listing 1. An entity type of an order subtotal

The *SAP Gateway* enables platforms and devices to consume data from SAP systems via OData services. The OData service implements the OData protocol and exposes a metadata document. The *Gateway Service Builder* (SEGW) application is used to create the OData service. It is automatically generating metadata and runtime objects from the existing database [5].

In SAP, the metadata document contains additional metadata that can be leveraged by client libraries or tools. Such an enhancement gives extra information about entities and properties to better interact with the service. Listing 2 illustrates how are the SAP attributes embedded into the metadata document. For example, *sap:filterable* attribute is applied in order to restrict the use of the *CustomerId* property in the filter queries.

```
<EntityType Name="CustomerInvoice"
    sap:content−version="1">
    <Key>
        <PropertyRef Name="CustomerId"/>
    </Key>
    <Property Name="CustomerId" Type="Edm.String"
        sap:creatable="false" sap:updatable="false"
        sap:sortable="false" sap:filterable="false"/>
</EntityType>
```

Listing 2. An entity type of a customer invoice

To read feed, a client application sends an *HTTP GET* request with a specific URI. After that, the corresponding getter methods are called to satisfy the user's demand. In the SAP environment, those methods are pre-generated by SEGW or programmed by developers in the ABAP language. The response returned from the server contains the desired data in the JSON or XML format. Data can be modified or created in the similar way.

At the moment, client libraries allow developers to parse metadata document into objects related to the concrete programing language. The OData protocol is supported by many libraries. The best-known libraries are Apache Olingo (Java), OpenUI5 (JavaScript, open-source version of SAPUI5), Pyslet (Python), ODataLib (C#) and ODataCpp (C++).

3. ODfuzz Overview

ODfuzz is a fuzzer that is fuzzing requests designated to be sent via the OData protocol. Protocol's structure remains the same but the actual data not. The proposed testing tool is briefly introduced in Subsections 3.1 and 3.2.

3.1 Fuzzer Components

Figure 1 demonstrates from what parts is *ODfuzz* composed of. *ArgParser* handles user input. The fuzzer

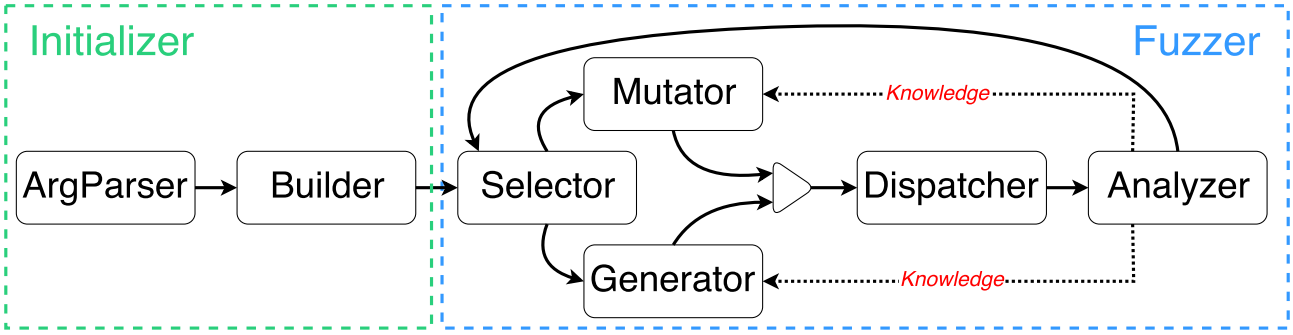


Figure 1. Components of *ODfuzz*. From left to right, *ArgParser* parses command line arguments, *Builder* initializes data structures and objects that will by used by the fuzzer, *Selector* is iterating over generated objects (for example, filterable entities) and selects a method for creating fuzzed data, *Mutator* mutates existing data, *Generator* generates new data, *Dispatcher* sends fuzzed data to the OData service, and, finally, *Analyzer* analyzes responses received from the server.

takes the following command line arguments: An OData service URL, logs directory (where logs from the fuzzer are stored), stats directory (where statistics are stored), and an option allowing the fuzzer to send multiple requests at once.

The *Builder* module creates a set of queryable entities. Those are entities that have no constrains which forbid the usage of an entity in querying. Constraints are identified by analyzing restrictions declared in a metadata document. In the SAP OData services, some entities may not support filtering, if they haven't got any filterable property. *Builder* removes such entities from the corresponding queryable group.

Selector, along with *Analyzer*, is the core part of the fuzzer. *Selector* determines what data will be created and what entity will be used in a query. It is utilizing objects that were built before.

ODfuzz uses genetic algorithm to create new test cases. In the first place, *ODfuzz* creates an initial population by generating queries that contain random data. An amount of initial population may vary because of different number of properties in every OData service. To create a diverse population, 20 queries are generated per property. Each query option, for example, $filter, $top, $skip, is attached to the particular entity. It can be used in a combination of multiple options by using the *&* operator, e.g.:

```
Products?$filter=ID gt 10 or ID lt 100&$skip=10&$top=5
```

The filter query option may consists of an infinite number of possible combinations of properties, operators, operands, etc. To generate such an option, and to combine it with other options, *ODfuzz* adopts rules defined in a context free grammar. *ODfuzz* implements a recursive descent generator to generate new queries. For the filter query option, generator randomly takes one property from a list of properties, and then grammar rules are applied in order to generate the rest of the query.

The initial population has to be evaluated by a fitness function. Because of that, the queries are sent to the server. The *Analyzer* module is analyzing HTTP codes and response times from the server responses. The analyzed objects represent numeric values and are passed to the fitness function. In the domain of fuzzing, the fitness value denotes the best test case that triggers an error.

Fitness value of an individual is computed from the HTTP status code, query length, and the response time. For instance, if the HTTP response code equals to 500, partial score is set to 100, otherwise, partial code is set to 0. Length of the query is evaluated by a division of threshold length and the query length. The threshold length is a maximum length of a query that should be evaluated at all. The response time is assessed by the same criteria. A final score is the sum of all partial scores.

Mating of the initial population leads to creating new children. Children are generated by a crossover after selecting the fittest parents. Tournament selection is used in order to select new parents. Random individuals are picked from the database and the fittest pair is chosen for mating. A query is divided into several parts. For example, *$filter=ID gt 10 or ID lt 100&$skip=10&$top=5* is divided into the $filter, that consists of *ID gt 10* and *ID lt 100*, $skip and $top part. One of these parts are randomly replaced with the same contextual part from the other individual, and vice versa. Cut-points are the logical operators *and*, *or*, *&*.

To newly created children is applied a mutation with a 50% probability. Mutations are characteristic for entity properties that hold a string or an integer value. The following types of the mutations are presented:

- an ASCII character replacement,

- a string truncation by one or more characters,
- incrementing or decrementing numeric values.

A query is sent to the server after the crossover and mutation, and is evaluated by the fitness function. After that, new individuals are mating again.

Genetic algorithms are used in many fuzzers in order to disclose more bugs, to get minimal data sample that triggers an error, and to get better code coverage eventually.

3.2 Implementation Details

ODfuzz is programmed in Python and is supposed to run in an infinite loop until it is canceled by a keyboard interruption.

Random generators are initialized with a seed to ensure tests reproducibility. This is achieved by invoking the *seed* method from the *random* module.

HTTP requests are sent by the *requests*[2] library. Fuzzer supports sending asynchronous requests. Asynchronous requests are sent by multiple threads which makes the fuzzer run many times faster than sending the requests one by one and waiting for the response.

Metadata are parsed and transformed to Python objects by the *PyOData* library. *PyOData* was recently developed by SAP employees. The library can treat advanced SAP attributes defined in a metadata document.

Responses received from the server, generated queries, and the score of the fitness function are saved to the NoSQL database *mongoDB*[3]. This database is platform independent and can be maintained on the local machine. *ODfuzz* uses *PyMongo*, a Python support library, to access and store data in *mongoDB*. Entity sets, properties, query options, error codes are logged in a Comma Separated Values (CSV) file as well. Data stored in CSV files are used for grouping and displaying aggregations via Pivot Table. Pivot Table is an interactive tool programmed in JavaScript[4]. A parsed CSV file is rendered into an HTML table by calling the *pivotUI* method.

4. Performed Experiments

An implemented prototype was able to find defects in one SAP application running in the cloud. Experiments were performed on the separated SAP development system. All errors were triggered by valid URIs and a combination of $filter query options. Table 2 shows what bugs were discovered by running the fuzzer multiple times for five hours:

Entity set	Error
1. Regions, Order_Qrs	SAPSQL_DATA_LOSS
2. PrinterDevices	/IWBEP/CM_MGW_RT/032
3. ProductDetails	500 Connection Timed Out

Table 2. Found errors

4.1 The SAPSQL_DATA_LOSS Error

An error which appeared in the *Regions* entity is dumped when the data are lost while copying a value. Let's imagine that we have one property that holds a string value with the maximum length of 10 characters. In metadata, the maximum length of this property is mistakenly set to 12 characters. The *SAPSQL_DATA_LOSS* error is produced by sending a request which contains a string with length of 12 characters. The fuzzer discovers this error almost immediately. The real problem here is not a wrong declaration of the maximum length but rather an inappropriate handling of data in the back-end modules. A detailed analysis proved that the back-end is expecting a different type of property at a specific place. Queries that contained the logical operator *or* with a shuffled order of properties produced an error. After replacing *or* operator to *and*, server did not fire off an error message.

The same problem applies to the *Order_Qrs* entity. What will happen when the customers decide to change the format of requests? Back-end components will have to be refactored too, instead of the front-end only.

4.2 The /IWBEP/CM_MGW_RT/032 Error

The */IWBEP/CM_MGW_RT/032* error was triggered by using the *Location* property and the logical operator *and*. This error refers to a wrong assumption that a combination of same properties in the filter query option is not supported. In SAP, the usage of same properties is forbidden by declaring *sap:filter-restriction="single-value"*. Such a restriction was missing in the metadata document. By changing the *and* operator to *or*, the error was not raised anymore.

4.3 The Connection Timed Out Error

The *ProductDetails* entity contains a property *Email* of type *String* which can be as long as 80 characters. By generating a random string with length of 60 characters, the server response was hanging for 5 minutes which resulted in the Connection Timed Out error. The session was terminated due to exhausted heap size as stated in the system log.

[2] http://docs.python-requests.org/en/master/

[3] https://docs.mongodb.com/

[4] https://pivottable.js.org/examples/

5. Conclusions

This paper introduced a tool that is capable of testing applications communication via the OData protocol by sending odd requests to the server. *ODfuzz* is implemented in Python and has reasonable results in its early phase of development.

The fuzzer discovered non-critical bugs that were already reported and revised by SAP developers. Most of the bugs were related to a data loss while copying values from internal variables. With the same testing approach it is possible to expose errors related to the ABAP conversion routines. These routines are functions that converts one type of C-like pointer to another. The ABAP interpreter does not address any issues when the data were corrupted during a conversion.

ODfuzz allows QA specialists to run the fuzzer in the presently deployed applications to reveal hidden bugs. Also, *ODfuzz* can be used for durability testing since sending a bunch of malformed data may lead to a performance degradation as well.

The Final version of the fuzzer can be used to work in two different modes. In the first mode, the fuzzer will run until it is explicitly interrupted by the *SIGINT* signal. Fuzzer's functionality will be deliberately reduced in the second mode and shall be used in continuous integration or delivery systems presented in SAP. The tool will perform simple checks and will generate data for entities declared in a metadata document without use of the genetic algorithm.

In the future, property values can by generated by its type domains defined in ABAP. Such domains may be retrieved from the OData service by calling supplementary function imports. This will make the fuzzer more autonomous and intelligent, so there will be no need to define restrictions as it is now.

Acknowledgements

I would like to thank my supervisor prof. Ing. Tomáš Vojnar, Ph.D. for his help. Also, I would like to express my gratitude to Jakub Filák for the suggestions and ideas.

References

[1] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008. ISBN: 978-1-59693-214-2.

[2] Toby Clarke. *Fuzzing for software vulnerability discovery*. Technical Report RHUL-MA-2009-4, Department of Mathematics, Royal Holloway, University of London, 2009.

[3] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. *Fuzzing: The State of the Art*. Technical Report DSTO-TN-1043, Defence Science and Technology Organization, Department of Science, Australia, 2012.

[4] Microsoft. *OData - The Protocol for REST APIs*, 2017. http://www.odata.org/documentation/.

[5] *OData and SAP NetWeaver Gateway*. SAP Press, 2014. ISBN: 978-1-59229-907-2.