Research Internship (PRE)

Field of Study: SIS/TIC
Scholar Year: 2017-2018

Test generation for DNS implementations in OCaml

Fuzz testing with afl-fuzz

Confidentiality Notice
Non-confidential report and publishable on Internet

Author:
Willy TAN

Promotion:
2019

ENSTA ParisTech Tutor:
François PESSAUX

Host Organism Tutor:
Anil MADHAVAPEDDY

Internship from 12/05/2018 to 12/08/2018

Name of the host organism: OCaml Labs
Address: Computer Laboratory, William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Confidentiality Notice

Abstract

One of the main cores of the current network infrastructure is the DNS protocol. Created in 1987, it is still evolving nowadays, the last standard being published in March 2016. As such, older implementations of DNS require constant updates, and newer implementations have to support all the standards defined during the last three decades. In this project, the focus will be on two DNS implementations in OCaml : ocaml-dns, created in 2005 and widely deployed, and μDNS, started in 2017 and still in development. The objective of OCaml Labs is to switch to μDNS as it uses modern OCaml libraries and packages, faster and more secure than the one used in ocaml-dns. Following this objective, the goal of this internship is to test both implementations, put forward possible bugs, check for standard compliance, and find behavioural differences between them, using fuzzing tools.

Keywords :

Fuzz testing, RFC compliance, UDP, DNS packet structure.

Acknowledgment

I would like to express my gratitude towards Anil Madhavapeddy and Gemma Gordon for having me included in the laboratory and for their valuable support. This would not have been possible without Anil's great knowledge on OCaml and DNS.

I also would like to thank Hannes Menhert for his solid knowledge on the DNS protocol and for his great and spontaneous help when I started working on μDNS, David Scott for his advices on the ocaml-dns implementation, and Stephen Dolan for his guidance on the Crowbar library. I am also lucky to have worked with Raphaël Cornet, another intern, who helped me with the fuzzing tools and who gave me many ideas for the project. I am grateful towards the whole OCaml Labs team for their help and their enthusiasm : it was a great experience working in this laboratory.

A special gratitude goes out to Michel Mauny, who introduced me to OCaml Labs, who taught me a lot about OCaml, and who continuously assisted me before I went to the laboratory.

Finally, I would like to thank my friends and my family, especially Olivier Nicole who helped me finding this internship, and Caroline Dam Hieu for her constant support on everything I did.

Contents

Confidentiality Notice **3**

Abstract **5**

Acknowledgment **7**

Contents **9**

Introduction **11**

I Context and nature of the internship **13**

I.1 The OCaml language 13
I.2 OCaml Labs 13
I.3 Domain Name System 14

II Fuzz testing and the Crowbar library **15**

II.1 Definition and tools 15
II.2 Fuzzing with Crowbar 16
II.2.1 Main functions and standard generators 16
II.2.2 Generator creation 17

III DNS Packet generation **19**

III.1 Writing DNS data in OCaml 19
III.2 DNS structure 19
III.2.1 Header 20
III.2.2 Question section 21
III.2.3 Resource record 22
III.2.4 Concatenation of the differents parts 23
III.3 Fuzzing procedure 23

IV Parser testing **25**

IV.1 DNS naming rules 25
IV.2 Message compression 25
IV.3 Fuzz testing with afl-fuzz 26
IV.3.1 Fuzz testing code 26
IV.3.2 Results 26
IV.4 Fuzz test with Crowbar 27
IV.4.1 Hack into the libraries 27
IV.4.2 Fuzz tests with the Crowbar packet generator 28

V Server testing **31**
V.1 DNS server security and rules 31
V.2 Sending DNS packets with OCaml 31
V.3 Server fuzz tests 32
V.3.1 Fuzzing implementation 32
V.3.2 Results 33

Conclusion **35**

Bibliography **37**

Introduction

Nowadays, the Internet is so inherent of our society that one would not think about the world working in the same way without it. It relies on a solid infrastructure and architecture that has been developed since the second half of the past century, and that is still evolving. One of the cores of this infrastructure is the DNS protocol : it is a decentralized naming system for nearly all the resources connected to the Internet, and, most prominently, it gives memorizable names to the IP addresses needed to locate those resources (e.g. `google.com` and its IP address `216.58.198.110`).

As it is in constant evolution, some older implementations have to be regularly updated to follow the changing standards. But as time goes on, it may be more clever to create a newer implementation to comply with those standards as well as to use more modern and optimized tools than updating the former one. This is especially true in the OCaml world which changed a lot since the last decade. That is one of the reasons the μDNS[1] project started in 2017 : to supplant the older implementation, ocaml-dns[2]

As μDNS is mostly untested and as ocaml-dns relies on older libraries, the goal of my internship was, by using a fuzz testing tool, to contribute to the μDNS development with tests but also to find undiscovered bugs or non standard behaviors in both implementations, especially for ocaml-dns which was never fuzzed before despite its age. The project could also be released as a package[3] for more development in the fuzz testing field.

Part I

Context and nature of the internship

I.1 The OCaml language

The OCaml language is a general-purpose programming language which unifies functional, imperative and object-oriented programming. Its simplicity to use, its powerful static type system and type inference ensures great safety, and its native code compiler and byte code compiler produces efficient programs, which is why it is popular in companies such as Facebook, Jane Street, or for cryptocurrencies such as Tezos. The OCaml language is used in softwares such as in the Docker compiler, the unikernel MirageOS, or in the interactive theorem prover Coq.

Even though the language is young when one looks at the computer history, its community is growing larger and larger, and newly created organisms such as OCaml Consortium are funding development of the language.

I.2 OCaml Labs

OCaml Labs is a section of the *Computer Laboratory* of the University of Cambridge in the United Kingdom, which role is to develop the OCaml implementation and its ecosystem. Notable projects where OCaml Labs is involved are :

- **Multicore OCaml**, which aim is to provide OCaml with multi-core tools, especially for mutable data and garbage collection;
- **opam**, a source-based package manager for OCaml. It has been created and is maintained by OCamlPro, and INRIA and OCaml Labs continuously helps its development. It provides users with a simple OCaml package installer tool, supports simultaneous compilers installations and Git-friendly development workflow;
- **Crowbar**, a property fuzzing tool combining QuickCheck-style property-based testing and afl-fuzz. Its purpose is to find possible bugs inside softwares with guided input generation.

I.3 Domain Name System

The Domain Name System (DNS) is a hierarchical decentralized naming system for the majority of resources connected to the Internet or to private networks. Its most prominent purpose is to translate simple and memorizable domain names to IP addresses, but more generally speaking, the system associates various information with the domain names.

Domain names consist of one or more parts, called *labels*. Labels are concatenated and separated by dots. Labels are associated with domains, the right-most label conveying the widest domain. The hierarchy descends from right to left: the label on the left represents a subdivision of the domain to the right. Even though it is rarely written, domain names end with a trailing dot, which represents the root of the DNS. Its direct subdomains are called *top-level domains*.

The domain name space can be represented as a tree, with the root being the actual root of the DNS. Each node is associated with a label and its possible *resource records* which hold information of the domain name. The domain name can be retrieved by concatenating the label of the actual node and the names of its parent node, separated by a dot.

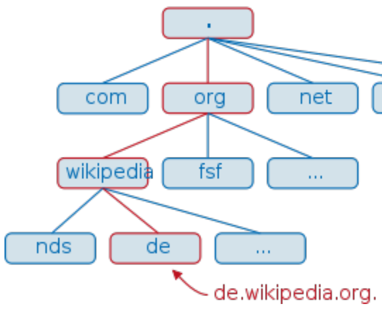


Figure I.1: DNS Tree

The DNS is maintained by a distributed database system, following the client-server model. The nodes of the database system are name servers which store concrete information. There are two types of name servers :

- *authoritative name servers*, which only gives answers to DNS queries following the rules and configuration made by the administrator. Usual answers are concrete information stored within the server, referrals to subdomains which may hold information, or *non-existent domain name error* answers.

- *resolvers*, which represent the client side. They are responsible for initiating queries, asking for a particular information held within a domain name. It is most often done recursively, for example when one queries `www.google.com`, the resolver will query the root name server which will refer to `com`, which in turn will refer to `google`, etc. When possible, the resolver will cache the answer for a determined period of time.

The DNS standards are defined by Request For Comments, which describe the proper configuration and behaviour of name servers. The RFC defining the DNS protocol have been written over three decades, so constant checks must be done to confirm that DNS implementations are up to date.

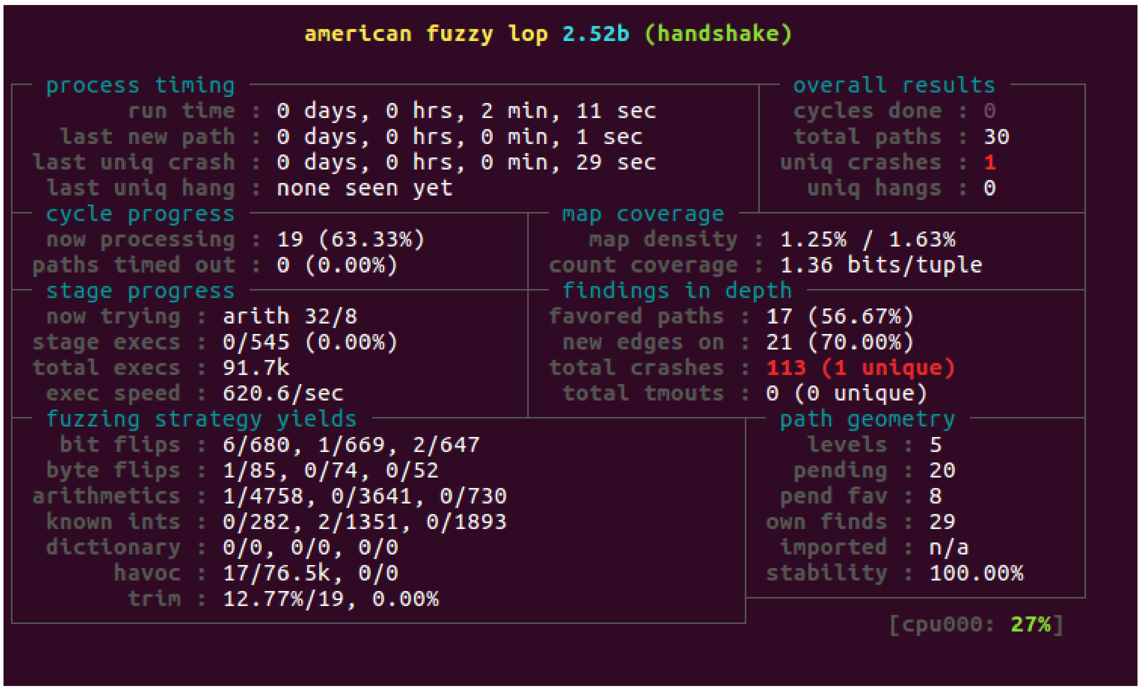
Part II

Fuzz testing and the Crowbar library

II.1 Definition and tools

Fuzzing or *fuzz testing* is an automated software testing based on providing randomly generated data as inputs for a computer program which is then monitored for exceptions, crashes, hangs or potential memory leaks. Most of the time, fuzzers are used when inputs are following a specific structure. An initial valid or invalid input is provided to the fuzzer, which mutates this input to create similar looking inputs that may be accepted by the parsers but may raise exceptions deeper in the program.

The fuzzer used during this internship is **afl-fuzz**[4], created by Michal Zalewski in 2014. The tool is supported by the OCaml language, and provides a complete and easy to use fuzzing tool.



```
                    american fuzzy lop 2.52b (handshake)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 2 min, 11 sec      │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec       │  total paths : 30     │
│ last uniq crash : 0 days, 0 hrs, 0 min, 29 sec      │ uniq crashes : 1      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────┬─ map coverage ─────┴───────────────────────┤
│  now processing : 19 (63.33%)   │    map density : 1.25% / 1.63%           │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.36 bits/tuple         │
├─ stage progress ────────────────┼─ findings in depth ────────────────────────┤
│  now trying : arith 32/8        │ favored paths : 17 (56.67%)               │
│ stage execs : 0/545 (0.00%)     │  new edges on : 21 (70.00%)               │
│ total execs : 91.7k             │ total crashes : 113 (1 unique)            │
│  exec speed : 620.6/sec         │  total tmouts : 0 (0 unique)              │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ────────┤
│   bit flips : 6/680, 1/669, 2/647               │    levels : 5          │
│  byte flips : 1/85, 0/74, 0/52                  │   pending : 20         │
│ arithmetics : 1/4758, 0/3641, 0/730             │  pend fav : 8          │
│  known ints : 0/282, 2/1351, 0/1893             │ own finds : 29         │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a        │
│       havoc : 17/76.5k, 0/0                     │ stability : 100.00%    │
│        trim : 12.77%/19, 0.00%                  ├────────────────────────┘
└─────────────────────────────────────────────────┘          [cpu000: 27%]
```

Figure II.1: afl-fuzz interface

Here is an example of a fuzzable program :

```
let () =
  let s = read_line () in
  let arr = Array.to_list (Array.init (String.length s) (String.get s))
  in match arr with
    |['h';'i';'d';'d';'e';'n'] -> failwith "uh␣oh"
    | _ -> ();;
```

The behaviour of this program is simple : it reads inputs from the computer with `read_line()` and if the input matches exactly the word `"hidden"`, it raises an exception. This program can be fuzzed with `afl-fuzz` as it shows an entry point in the `read_line ()` call. The fuzzer then generates inputs from an initial one (here I chose `"startingword"`), and provides instrumentation so that when it reaches unknown paths, it will mutate from the input that reached that unknown path. For example, if the fuzzer manages to generate a word starting with `'h'`, the program will behave differently as the first character matches the `'h'` of the word `"hidden"`.

However, most of the programs are not as simple as the one written above, and inputs may be way more complex, which is the case of the DNS protocol. `afl-fuzz` in itself may not be enough to generate relevant data, and a more guided tool, Crowbar, may be used in that case.

II.2 Fuzzing with Crowbar

`Crowbar`[5] is a recent testing library developed by Stephen Dolan during 2017. It provides the user with random data generation functions, which can then be given as inputs to programs or can be used to check specific properties. Data generation can be combined with `afl-fuzz` instrumentation for more efficient code coverage. In that case, the initial input will serve as a seed for data generation, and `afl-fuzz` will mutate that seed to generate future data. If `afl-fuzz` is not used, the program will generate 5000 random generated values instead, without looking at code coverage, and stop at the first failure raised from either a crash or a violated property.

II.2.1 Main functions and standard generators

`Crowbar` provides many 'classic' generators (`int`, `range`, `float`, `bytes`, etc), and tools to create our own generators. Those can then be used with property-testing functions. Here is an example of `Crowbar` usage :

```
let () =
  Crowbar.add_test ~name:"Int␣generation␣test" [Crowbar.int] (fun n ->
    Crowbar.check (n > 1000 && n < 5000))
;;
```

The function `add_test` from `Crowbar` takes three arguments : a name for the test, a list of generators, and a test function. The generator used here is an `int` generator, which can generate an integer from the lowest possible integer to the highest one. When the program is executed without `afl`, up to 5000 integers will be generated, and the program will stop and raise a failure if a generated integer isn't in the interval $]1000; 5000[$

One important thing to note is that `add_test` does not specifically require a property check in the test function, only that it is a `unit`-type function. The function will only raise a test failure error when it crashes, as when `afl-fuzz` is used without `Crowbar`.

II.2.2 Generator creation

Crowbar provides useful functions for generator creation. Two main functions will be described as they are a staple in DNS query generation : `map` and `dynamic_bind`. The first one takes a list of generators and a function which takes those generators' outputs to create a new one. For example :

```
let sum_as_string_generator =
  Crowbar.map [Crowbar.int; Crowbar.int] (fun a b ->
    Printf.sprintf "%d+%d" a b);;
```

`sum_as_string_generator` can then be used in tests, and will generate strings such as `"-4546+15"`, `"0+0"`. However, `map` does not express the generation of a value whose generator itself depends on a previously generated value. In that case, `dynamic_bind` has to be used. That function takes two arguments, a single generator, and a function which output is a generator (and not a value as `map` did). However, because that function is built with a dynamic structure, the compiler can't analyze it statically. As a consequence, some intern optimizations may not work, so it should be avoided unless it is necessary. Here is an example where `dynamic_bind` is needed :

```
let gender = Boy | Girl;;
let gender_gen = Crowbar.choose [
  Crowbar.const Boy;
  Crowbar.const Girl];;

let boy_name = Crowbar.choose [
  Crowbar.const "François";
  Crowbar.const "David"];;

let girl_name = Crowbar.choose [
  Crowbar.const "Tracy";
  Crowbar.const "Agatha"];;

let which_name = Crowbar.dynamic_bind gender_gen (fun gdr ->
  match gdr with
  |Boy -> boy_name
  |Girl -> girl_name);;
```

The purpose of this generator is to choose the name of a newborn according to its gender. As the gender is needed to know whether to choose a male name or a female name, `map` can't be used with those types. The functions `choose` and `const` from `Crowbar` are also called here : the first one chooses randomly a generator in a list of generators, and the second one is a generator of constant value.

Part III

DNS Packet generation

One of the requirements of fuzz testing is to find entry points. For the DNS protocol, there are plenty of them. However, the focus will be only on the main entry points, the parsers and servers. Exploitation of those entry points will require proper DNS packet generators as `Crowbar` will be used.

III.1 Writing DNS data in OCaml

Both ocaml-dns and μDNS use `cstruct`-type objects to send and receive DNS data through UDP protocol. `cstruct`[6] is a library that allows users to manage `C`-like structures, as if they were memory buffers. The `cstruct` library provides the function `of_hex`, which converts a string written in hexadecimal to a `cstruct` object, thus the generator can be defined using only hexadecimal strings, and `of_hex` will only be called during the fuzzing process.

Knowing this, I defined many functions and tools to craft hexadecimal strings in the packet generation library, such as the generator `hex` which generates one byte as an hexadecimal string, `hex_times n` which creates a hexadecimal string generator on `n` bytes, or `hex_concat hex1 hex2` which concatenates two hexadecimal string generators, which will serve as the base of the packet generator.

III.2 DNS structure

All DNS packets can be divided in five sections :

```
+---------------------+
|        Header       |
+---------------------+
|       Question      | the question for the name server
+---------------------+
|        Answer       | RRs answering the question
+---------------------+
|      Authority      | RRs pointing toward an authority
+---------------------+
|      Additional     | RRs holding additional information
+---------------------+
```

The header holds the information about the message in itself, whether it is a query or an answer, the number of queries or answers, if there was an error, etc. The question section contains fields that describe the query. The answer, authority and additional section have the same format, a possibly empty list of concatenated resource records, which will be described later.

III.2.1 Header

Header structure

The header is described as below, and has the same format for all the packets.

```
                                   1  1  1  1  1  1
     0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |                      ID                       |
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |                    QDCOUNT                    |
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |                    ANCOUNT                    |
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |                    NSCOUNT                    |
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |                    ARCOUNT                    |
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The `ID` field identifies the query and has to be copied in the answer so that they can be matched.

The `QR` bit indicates whether the packet is a query or an answer.

The `Opcode` field specifies the type of query or response, whether it is a standard query, an update to the database, etc.

The `AA` bit indicates whether the answer was issued by a name server with authority on the name section.

The `TC` bit indicates whether the packet was truncated because of size issues.

The `RD` bit indicates whether a recursion is desired during the sequence of DNS exchanges.

The `RA` bit indicates whether recursion support is available in the answering name server.

The `Z` field is reserved for future use.

The `RCODE` field indicates if there was an error, and the type of error according to the number in that field.

The QDCOUNT, ANCOUNT, NSCOUNT, ARCOUNT fields indicates the numbers of questions, answers, authority and additional information in the message.

OCaml implementation of headers

Each two-bytes field is defined by default with the function `hex_times 2`, with an option for the ID field to be an incrementing counter. For the flags and codes line, `Crowbar.range n` which generates integers up to `n` is combined with logical shifts on the left to represent them on two bytes.

The function `make_header id flags_and_codes qdcount .. arcount` then concatenates each of the generators into a single one. By default, each field is as random as possible, using the default generators used above. If one wants to use their own generator, such as constant ones to fix some fields, they may do so with the optional arguments.

III.2.2 Question section

Question section structure

The question section has the following structure :

```
                                    1  1  1  1  1  1
      0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                                               |
    /                     QNAME                     /
    /                                               /
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                     QTYPE                     |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                     QCLASS                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The QNAME field is for the queried domain name(s). A domain name is represented as a series of labels, prepended with their length on one byte.

The QTYPE field identifies the type of the query, whether it is for a IPv4 query (A type), a mail server address (MX type), etc.

The QCLASS field describes the class of the query. Nearly all of the queries use the IN class, the other classes CHAOS and HESIOD being marginal.

OCaml implementation of the question section

Generation for the QTYPE and QCLASS fields is simple as it only uses `hex_times 2`. As the length needs to be prepended to each label of a domain name, the generation is a bit more complex. First, labels must be defined :

```
let label = Crowbar.map [Crowbar.list1 hex;hex_range ~min:1 255]
                        (fun list last ->
    let shortlen = Printf.sprintf "%x" (List.length list + 1) in
    let length = prepend_zero shortlen 2 in
String.concat "" (length::list@[last]));;
```

A randomly sized list of one byte `hex` generators created with `Crowbar.list1 hex` is used as an argument to `Crowbar.map`, which is concatenated in the function and prepended with the list length converted to a hexadecimal string. The last byte can't be null as it identifies the end of a domain name, so I added a non null hex generator in the initial list. The function `prepend_zero` adds zeros to the hexadecimal string to that the length value holds on exactly one byte.

Once labels are defined, domain names can now be generated by concatenating a randomly sized list of labels with a zero byte at the end.

```
let name = Crowbar.map [Crowbar.list1 label] @@ (fun l ->
String.concat "" (l@["00"]));;
```

III.2.3 Resource record

Resource record structure

A resource record has the following structure:

```
                                    1  1  1  1  1  1
      0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    /                                               /
    /                      NAME                     /
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                      TYPE                     |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                     CLASS                     |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                      TTL                      |
    |                                               |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                   RDLENGTH                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--|
    /                     RDATA                     /
    /                                               /
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The `NAME` field is the same as the `QNAME` field : it represents one or multiple domain names.

The `TYPE` field describes the type of the query. The `TYPE` set is a subset of the `QTYPE` set.

The `CLASS` field describes the class of the query. Once again, most of the times, it is the `IN` class which is used.

The `TTL` field is the Time To Live of the resource record, in seconds. It indicates the number of seconds the resource record can be considered valid in caches and transactions.

The `RDLENGTH` field is the length of the `RDATA` field.

The RDATA field is a variable length string of octets that describes the resource. Its format depends of the TYPE and CLASS defined above. For example, if the type defined is A and the class is IN, this field will contain an IPv4 address.

OCaml implementation of resource records

Once more, the TYPE, CLASS, TTL fields use a simple hex_times function. The NAME field uses the same domain name generation as the one for the question section.

For the RDLENGTH and RDATA fields, as they are variable length fields, dynamic_bind is used to bind an integer (arbitrarily chosen) to hex_times. It is important to understand that generating valid RDLENGTH and RDATA fields require a specific generator for the 86 different valid RR types. Considering the time it would take to craft one for each of them and the length of my internship, I decided to not focus on this part.

III.2.4 Concatenation of the differents parts

Each part of a DNS message have been defined with Crowbar : the last step is to combine them into a full packet. To do so, we can write the function make_packet, which concatenates each generator using hex_concat_list. Each of the generators can be substituted with specifically crafted ones, using optional arguments :

```
let make_packet ?(header=make_header ()) ?(query=make_query ())
   ?(answer=resource_record ()) ?(authority=resource_record ()) ?(
   additional=resource_record ()) () =
hex_concat_list [header;query;answer;authority;additional];;
```

III.3 Fuzzing procedure

Generators created with this function can now be used to fuzz any entry point using DNS messages. To delve deeper into the DNS structure, the procedure I chose to fuzz with the packet generator is as it follows :

1. Begin with the most general packet possible
2. Start the fuzzing process for a few minutes
3. Process the crashes and check for valid error handling and compliance
4. Modify the packet generator to stop the generation of the checked value
5. Repeat from step 2

Part IV

Parser testing

One of the main entry points one would think of when fuzzing a DNS server would be its parsing process. There should be two major points to test, corrupt query processing and standard compliance :

- DNS servers must distinguish valid messages from corrupt messages that may be sent through the network from malicious users or due to bugs, that could cause some issues or in the worst case crashes,
- if standard compliance is not followed, communication between different servers could prove difficult as a valid query from one server may be considered invalid from another one.

IV.1 DNS naming rules

According to RFC1035[7] and RFC2181[8], DNS servers must allow any kind of syntax for domain names, as long as each label has less than 63 characters and the whole name has less than 255 characters including the dot separators. Thus, a domain name such as `www.aéé(\%(àçè_.*-++` must be processed as a valid name. However, the purpose of domain names is to be easily memorized and used at a global scale, that is why names are often chosen with simple ASCII characters.

Each label has its length prepended on one byte, and the last label must be a zero byte, except if it is a pointer, as described in the next section.

IV.2 Message compression

The DNS protocol was defined in 1987, when computers were less efficient that today's computers. As a consequence, a compression scheme was defined to reduce the amount of bytes used in each message. If a domain name is used multiple times in the same message, each repetition would be replaced by a pointer to the first occurrence.

The pointer takes the form of a two bytes sequence, with the first two bits set to 1, and the rest is the byte position of the first occurrence. The parser can identify those pointers by calling a logical `AND` between the considered sequence and a mask with the first bits set to 1. This flag allows pointers to be distinguished from labels, as the length byte before the label is less than 63 (`0011 1111` in binary format).

The two flags `00` and `11` are the only ones allowed, any other combination should be considered as corrupt data.

IV.3 Fuzz testing with afl-fuzz

IV.3.1 Fuzz testing code

It is important to begin with `afl-fuzz` before using `Crowbar`. The use of the `Crowbar` library is useful to guide the fuzz testing, but it narrows the set of possible messages. Because `afl-fuzz` is less guided, it explores a wider portion of the whole implementation. With that in mind, I wrote a program similar to the example in II.1 :

```
(*For udns*)
let main () =
  let s = read_line () in
  let testcase =
    try Some (Cstruct.of_hex s)
    with
    | e -> None
  in
  match testcase with
  | None -> ()
  | Some cstr ->
    (match Dns_packet.decode cstr with
     | Ok (parsed,_) -> Format.printf "%a\n\n" Dns_packet.pp
        parsed
     | Error e -> Fmt.failwith "%a" Dns_packet.pp_err e)}
```

This code works essentially in the same way : the program reads an input, then tries to decode it. If there is an error raised, the fuzzer will have found a crash. One test is done in the first `try ... with` block, because bad hexadecimal conversions, without relation to the parser, tend to pollute a lot the test folder. The last `match ... with` block is specific to μDNS which uses monads that have to be unwrapped. The code for ocaml-dns is essentially the same, without the monad unwrapping.

IV.3.2 Results

The fuzzing was done with an initial query of type `A`, class `IN`, asking for the domain name `www.northeastern.edu`. The ID was 56130, and the question section was the only non-empty section.

ocaml-dns results

A few minutes after starting the fuzzing process, thousand of crashes have been raised. Processing them with a handwritten script shows that most of the errors come from bad inputs : in those cases, the byte length before a label does not correspond to that label length, leading to parsing errors raised. Those crashes are not that important as they are easily caught in a DNS server.

One of the other errors was that some packets had pointers that would refer to an offset that exceeds the packet length, leading to a parsing error. But, I noticed that one of them had a wrong flag for a pointer :

```
Reading file : ./forAFL/query_output/crashes/id:000013,sig
   :06,src:000001,op:flip1,pos:30
```

```
Content : Db42 0100 0001 0000 0000 0000 8377 7777 0c6e 6f72
    7468 6561 7374 6572 6e03 6564 7500 0001 0001
Fatal error: exception (Failure "Name.parse_pointer:␣Cannot␣
    dereference␣pointer␣to␣(887)␣at␣position␣(12)")
```

The byte before the sequence of six 7 indicates a length of 131, which is `1000 0011` in binary. The first two bits are neither two zeros or two ones, meaning that the parser should have raised an error. In fact, the ocaml-dns parser processed wrongly that label as a pointer. More specifically, it considers that a label `lab` is a pointer if that label meets the condition `(lab land 0b0_11000000)!= 0`. But, if `lab` is equal for example to `0b0_10000000`, the condition is met even though it does not describe a pointer, which explains the error above. A solution would be to replace the former condition with `lab >= 0b1100_0000`[9].

Other noticeable errors raised were when the parser would meet unknown values for some fields. Those errors are raised so they can be caught at a higher layer.

μDNS results

Similarly to ocaml-dns, a few minutes is enough to produce thousands of crashes. Most of them were unknown values handling, such as classes different from the usual `IN`, or unknown types. Bad flags for labels are also correctly handled.

Though, the parser refused to consider valid domain names with characters that are not hyphen and alphanumeric. For example, `www.~ortheastern.edu` was refused. It is a non standard behavior that could lead to some issues if other servers use non alphanumeric characters. However, the developer of μDNS Hannes Menhert is aware of this issue : the decision to refuse those queries is intentional as he wants to see where this strictness will lead to, and this strictness may be dropped for the final release.

First conclusion

The results for the fuzz tests using a non guided fuzzing are quite satisfying : it uncovered some bugs and issues in a few minutes with a quite simple program. However, the fuzzer spends most of its time on issues that are properly handled by the parser, and does not go deep into the complex DNS structure, as packets generated a few hours after the start still look the same as the first ones.

IV.4 Fuzz test with Crowbar

Because a non guided fuzzing does not cover the whole DNS structure, it seems mandatory to use `Crowbar` to unveil potential errors coming from the parser.

IV.4.1 Hack into the libraries

Because the packet generator is a generic way to fuzz, I first tried to fuzz from the inner structure of the DNS implementations. Using specific generators for each library could prove to be efficient and could reveal inner uncaught problems.

ocaml-dns implementation

Fuzzing the parser required to make a generator for each type used in the code. The library `ppx_deriving_crowbar`[10] provides a macro to create a `Crowbar` generator for enumerated types, which I used for most of them. However, there were more complex sum types, which use types from other libraries, such as :

```
type rdata =
  | A of Ipaddr.V4.t
  | AAAA of Ipaddr.V6.t
  | AFSDB of Cstruct.uint16 * Name.t
  | CNAME of Name.t
  | DNSKEY of Cstruct.uint16 * dnssec_alg * string
  | DS of Cstruct.uint16 * dnssec_alg * digest_alg * string
... (*27 more different types*)
```

Generators for those types had to be written from scratch, and each type called here also had to have a generator. Once each generator had been defined, the fuzzing procedure I chose was to encode the generated structure into binary data, and then parsing it.

ocaml-dns results

The only errors raised were from unknown values handling, so that higher layers in the DNS protocol can handle them. All the values were correctly handled, and no combination led to strange bugs.

Sometimes, the process of encoding then parsing a packet led to a different query between the initial one and the parsed one. But, this came from the fact that numbers higher than the possible value allowed by their byte length were truncated. This issue has no impact on the whole implementation, as those values were generated outside of the message creation chain and then put inside the last creating function. There is no possibility that it happen in reality, and does not lead to concrete problems.

The only interesting point to note is for domain names : names that had more than 255 characters in total were correctly encoded then parsed, which should not be allowed according to the defined DNS standards. In fact, the name parsing function didn't check the sum of each label's length. To solve this issue, a variable for the whole length has to be added to the inner recursive label parsing function, which is then checked at each call.[9]

Second conclusion / What about μDNS ?

Hacking into the ocaml-dns library proved to be not as efficient as one would think. The amount of time spent to create a generator for each type, especially for the more complex ones, was not worth the result, considering that it was specific to ocaml-dns and could not be used for other implementations. Knowing that, added to the fact that μDNS uses Generalized Algrebraic Data Types, difficult to handle with `Crowbar`, I decided to stop going in this direction.

IV.4.2 Fuzz tests with the Crowbar packet generator

Fuzz tests with the Crowbar packet generator should allow to target some fields of the parser. If one field is not handled correctly, fixing the generator to specific values can help to delve deeper into the DNS structure, compared to the non guided fuzz testing.

Implementation

With the packet generator already defined, the code looks rather simple :

```
let udns_packet_test packet =
  Crowbar.add_test ~name:"Udns␣packet␣generation" [packet] @@
      (fun packet ->
      let cstr =
      try
        Cstruct.of_hex packet
      with
      | e -> Printf.printf "%s\n%s\n%!" (Printexc.to_string e
        ) (Printexc.get_backtrace ());raise e
      in
      let packet = Dns_packet.decode cstr in
      match packet with
      | Ok (p,_) -> Format.printf "%a\n\n%!" Dns_packet.pp p
      | Error e -> Format.printf "%a\n\n%!" Dns_packet.pp_err
          e)
;;
```

Because we are only looking for crashes and not specific properties, there is no explicit property check done, so the code works similarly to the non guided one. The main difference comes from the input, which is generated with the packet generator given as argument. As the generator should be well written, `of_hex` conversion should not raise any errors.

ocaml-dns results

Following the fuzzing procedure in III.3, I managed to find a bug in the opcode handling. Even though there are sixteen possible values for opcodes, only five values are used[11] : `0 (QUERY), 1 (IQUERY, obsolete), 2 (STATUS), 4 (NOTIFY), 5 (UPDATE)`. Other unused opcodes may be used in the future, so the parser must accept those unused opcodes. However, ocaml-dns opcode type is a plain enumerated type. There is no integer associated with the value `RESERVED`, so a simple solution would be to add an integer type to the enumerated type of the opcode and change the integer conversion. As the former function uses macros, the generated functions would have to be rewritten.[9]

μDNS results

Fuzzing the μDNS parser implementation did not yield interesting results. Every combination of values is handled, following the standard compliance or not. Behaviors not following the standards are issues known by the developer, who may solve them at the release if the strictness is loosened.

Third conclusion

The simplicity of using the packet generator leads to much more efficient fuzzing. Pinpointing specific fields in the generator leads to clearer and more efficient debug processing. Now that the parsers has been thoroughly tested, we can now check for server bugs.

Part V

Server testing

As the DNS servers are mandatory infrastructures for the Internet, it is important to test their behavior, or if they have vulnerabilities. They have one sole entry point, which is data reception.

V.1 DNS server security and rules

As there are many rules defining the standards for DNS servers, only the main ones that will appear in the fuzzing process will be covered here.

The DNS protocol follows the robustness principle : *"Be conservative in what you do, be liberal in what you accept from others"*. Every parsable packet should be processed, but with security concerns. One of the essential rules when sending a request is to check whether the received answer was from the queried server. For example, the ID section of an answer should match the ID of the initial query, and the question section should be fully copied in the answer, to avoid packet spoofing.

The answer in case of a query that asks for a non-existent domain name or in the considered authoritative DNS server should be a `NXDOMAIN` rcode, meaning that it does not exist. For packets that can not be recognized because of unknown values, the `NOTIMP` rcode should be used.

V.2 Sending DNS packets with OCaml

The `lwt` library[12] provides a way to send data using the UDP protocol. We have to write two parts for DNS exchanges which are sending packets and receiving them. The library is quite complicated to use, so I used already written functions of the ocaml-dns implementation to make send and receive functions compatible with `Crowbar`.

For the sending process, the function that will used is `send_packet`, which takes two arguments, a `cstruct`-type object and a file descriptor corresponding to the IP address and port of the sender, and sends the object to that packet, without length restriction, to the server, which is by default `127.0.0.1#53`.

For the receiving process, the function `receive` will be used. It takes two arguments, the parsing function corresponding the the fuzzed implementation and a file descriptor, and waits for any response to come, and then parses it. I had the possibility to add checks on ID and question section for the waiting process, but it slows down considerably the fuzzing process as there are thousands of packets sent per second in parallel.

V.3 Server fuzz tests

V.3.1 Fuzzing implementation

Fuzzing the servers in a non guided way will prove to be way less efficient than guiding it, as it will spend a long time on the parser which was already tested. That is why only `Crowbar` fuzz testing will be done. As we now know which generators produce parsable packets, we can send them to the servers to see if they behave correctly when receiving those packets.

There are two ways to fuzz test servers : a send-and-receive fuzz test, and a send-only fuzz test. A send-only test may be used to simulate a Denial of Service attack because it is faster than a send-and-receive process, but as the program does not receive actual answers, the fuzzer is not able to know whether it covers parts of the code or not. Behavior testing is done with the send-and-receive process, and can cover more parts of the server code.

The send-only fuzz test has a rather simple code :

```
let send_only ?(packet=query_packet) () =
  Crowbar.add_test ~name:"Packet␣send" [packet] @@ (fun packet ->
      let cstr = Cstruct.of_hex packet in
Lwt_main.run (currentfd () >>= fun ofd -> send_packet cstr ofd))
    ;;
```

As the aim is to provoke a Denial of Service, the test does not print the sent query. The operator >>= is a monadic bind : `currentfd ()` is a monadic file descriptor that has to be bound to the `send_packet` function. Because `send_packet` returns a `lwt` thread and not an actual sending process, it has to be run with the function `run`.

The send-and-receive fuzz test works essentially in the same way :

```
let udns_send_and_receive ?(packet=query_packet) () =
  Crowbar.add_test ~name:"udns␣send␣and␣receive" [packet] @@ (fun
      packet ->
      let cstr = Cstruct.of_hex packet in
      let send_pkt = match Dns_packet.decode cstr with
      |Ok (p,_) -> p
      |Error e -> Format.printf "%a\n\n%!" Dns_packet.pp_err e;
                  Crowbar.bad_test ()
      in
      Format.printf "Packet␣:\n\n%a\n%!" Dns_packet.pp send_pkt;
      let recv_pkt =
        Lwt_main.run (currentfd () >>=
                      (fun ofd -> send_packet cstr ofd >>=
                         fun _ -> receive udns_parse ofd)) in
      Format.printf "Answer:\n\n%a\n\n%!" Dns_packet.pp recv_pkt)
```

Notable parts that are :

- A first test to drop or not the initial generated packet. There may still be errors linked to the parser, so they should be dropped and not considered as crashes with the function `bad_test`;
- the `receive` function is bound to the `send_packet` function so that it runs in the same lwt thread sequence.

V.3.2 Results

ocaml-dns results

The server example in the ocaml-dns repository only serves as a demonstration of query processing, and only processes `IN` class, `A` type queries, so actual behavior cannot be tested properly. In the ocaml-dns implementation, the answer behavior is solely decided by the DNS administrator. As such, fuzzing with send-and-receive did not yield interesting behaviors.

However, as long as the size check modification I suggested in IV.4.1 is not implemented, it is possible to noticeably slow down the server by sending maximum sized UDP packets : the maximum size of a UDP packet is 65535 bytes, which is huge compared to the 512 byte restriction on DNS packets.

μDNS results

The μDNS servers did show more interesting results, as some packets could crash them. μDNS does not process queries if their question section contains more than one question. If that is the case, a `FORMERR` answer will be sent, with the question section copied. However, the size allowed by μDNS for the error answer was only sized for a single question. It means that packets with a question section of great size made μDNS try to copy the whole section in a small buffer, which led to a crash.[13]

After being notified of that issue, the solution chosen by the developer was to copy only the first question if the section were to exceed the limit, but that choice could lead to some issues if some servers were to check for question section equality and would then dismiss the answer.

Another crash was also found thanks to fuzz testing : queries with opcode set to `NOTIFY` and query flag set to true could crash the resolvers. As the `NOTIFY` opcode is reserved for authoritative name servers and not resolvers, the developer decided to ignore the request and consider it as an empty request. But as consequence, because the query flag was set to true, it went through a function that considered empty queries as impossible, and thus raised an `assert false`, which crashed the resolver.[14]

Conclusion

DNS servers have to follow many rules defined on more than thirty articles defining the standards. It was not possible during my internship to cover every one of them, so I decided to focus on the most essential ones. Those parts have been thoroughly tested, so if the solutions I suggested are merged, there is nearly no chance that more bugs will be found with the fuzz testing. There are still more paths to explore, especially for resource record generation, and in the DNS security[15]. A new protocol, DNSSEC, has been defined for DNS security, and as it is still recent, its implementation is yet untested.

This project will possibly be released as an `opam` package so that people can use the code as a foundation for future DNS fuzz testing, especially the packet generator tool.

On a personal note, this project allowed me to discover the whole world of OCaml. I gained a huge amount of knowledge in this three months internship, and new ways to develop in this language, different from the school teachings. Moreover, developing in OCaml was a great way to strenghten my code writing skills because of the strictness of the language, which will prove useful in the computer science world. It is the first time I made a contribution to the development of a language, so it was a great experience. I am really grateful to the whole OCaml Labs team to have given me this opportunity, and I may work again in the OCaml world in the future.

Bibliography

[1] μDNS repository. https://github.com/roburio/udns, 2017.

[2] ocaml-dns repository. https://github.com/mirage/ocaml-dns, 2005.

[3] Internship project repository. https://github.com/Willy-Tan/afl-dns, 2018.

[4] American Fuzzy Lop website. http://lcamtuf.coredump.cx/afl/, 2014.

[5] Crowbar repository. https://github.com/stedolan/crowbar, 2017.

[6] Cstruct repository. https://github.com/mirage/ocaml-cstruct/, 2012.

[7] RFC1035 - Domain names - Implementation and specification. https://tools.ietf.org/html/rfc1035, 1987.

[8] RFC2181 - Clarification to the DNS Specification. https://tools.ietf.org/html/rfc2181, 1997.

[9] ocaml-dns pull request, correction suggestions. https://github.com/mirage/ocaml-dns/pull/154, 2018.

[10] ppx_deriving_crowbar repository. https://github.com/yomimono/ppx_deriving_crowbar/, 2018.

[11] RFC6895 - Domain Name System (DNS) IANA Considerations. https://tools.ietf.org/html/rfc6895, 2013.

[12] Lwt manual. https://ocsigen.org/lwt/manual/, 2018.

[13] μDNS server crash issue. https://github.com/roburio/udns/issues/5, 2018.

[14] μDNS resolver crash issue. https://github.com/roburio/udns/issues/6, 2018.

[15] RFC4033 - DNS Security Introduction and Requirements. ttps://tools.ietf.org/html/rfc4033, 2005.