UNIVERSITÉ DE BORDEAUX

SCIENCES ET TECHNOLOGIES

MASTER CSI

TER

Project report: Fuzzing the TCP/IP stack in libslirp userspace

CHAN PENG Maëlie, LAFONTAINE Alisée, MERLE Jean-Charles

Supervised by Samuel THIBAULT

15th of April, 2024

Contents

1 Introduction

1.1 QEMU and libslirp

QEMU stands for Quick Emulator. It is a free open-source software used to emulate a computer's processor. Libslirp is a library providing a virtual network with no configuration nor privileged services on the host.

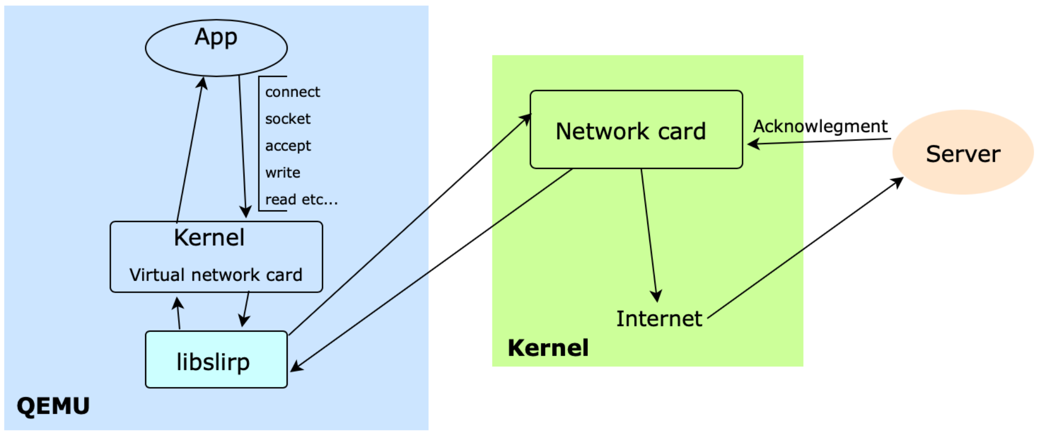


Figure 1.1: libslirp general operation

The goal of this project was to improve and learn about this library using the fuzzing method and its importance when thinking of computer security.

1.2 General concept of fuzzing

Fuzz testing, also known as fuzzing, is a very commonly automated software-testing method used to find but also monitor bugs and crashes that might occur when running a program. It inputs random or invalid data into a computer program and analyzes the software response, such as crashes or successful exit.

The functioning of a fuzzer is really simple yet very effective to report unseen bugs and discover unexpected behaviors that might be used by ill-intentioned people. The fuzzer generates inputs that can have a certain format or size specified by the user and then runs the code with these inputs. If the program exits normally, the fuzzer keeps creating new inputs to run with the code; otherwise, it stops and reports the bug.

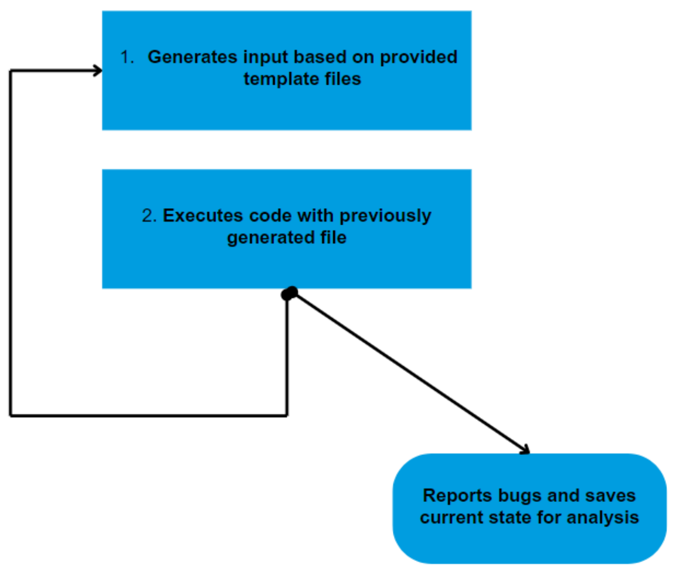

1. Generates input based on provided template files
2. Executes code with previously generated file
Reports bugs and saves current state for analysis

Figure 1.2: Fuzzing in three steps

Fuzzing offers many benefits, as seen before, it helps troubleshooting and finding bugs or crashes. It is thus effective in cost reduction on different levels:

- reduces engineering costs: engineers spend less time looking for bugs as it is an automated method, they only need to correct the bugs, not find them.
- reduces exploits, hence less money needed to repair the damages.

Nevertheless, there is an additional cost being the CPU-use time. Indeed, running the fuzzer and forcing crashes to occur consumes CPU resources that might be needed somewhere else.

1.3 The project

In the past, people looking for bugs and crashes were either curious about the limits of the tools they were using, or were troubleshooting code they had implemented themselves. But now, flaws in the code can be marketed and sold. We can observe an emergence of professionals on the lookout for crashes. So in order to increase the security level of libslirp, we worked on finding and correcting bugs.

Before we did, another student Jérémy Marchand worked on this subject. He greatly contributed to the project by documenting his work (Readme files) but also implementing the first version of the fuzzer (UDP and IP-header testing fuzzers). Thanks to the quality of his work and his effort to document it, we could easily understand what his contributions were and what other improvements were to be made.

This project was a group project. Because we have already worked together before, the division of work and the organization of the meetings came easily. We would have a weekly meeting with our supervising teacher, Mr. Samuel Thibault, where we could ask questions and discuss possible aspects of the project we could work on. Then, during the rest of the week we would assign a task to each member and discuss it through calls and meetings on campus.

2 Fuzzing

Following Jérémy Marchand's steps, we also used the fuzzer base from LLVM libfuzzer. And the method we used was the black box technique, meaning that we are trying to find crashes and bugs without knowing the structure of the program we're doing our research on.

Jérémy Marchand made the decision to implement a fuzzer that would focus on protocols so that the fuzzer would be forced to target only a specific part of the code and would hopefully explore more cases. Furthermore, he combined the libslirp library to the fuzzer by compiling the library as static and he had focused on UDP and IP.

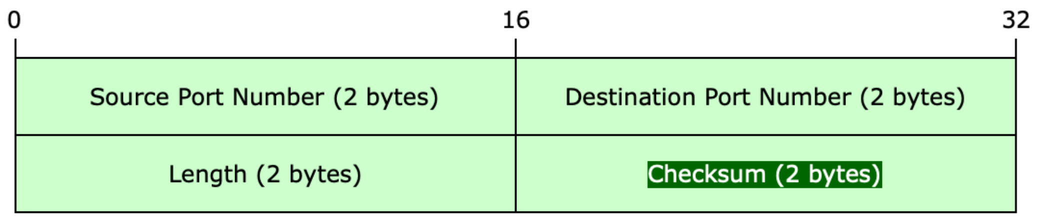
2.1 Creating fuzzers

We established multiple points on which we wanted to focus on this project. The three main ones were the following: make improvements on the use of the fuzzers, analyze and understand fuzzers' results and implement one or multiple fuzzers. Before implementing them, we had to make sure we understood the differences between the protocols, especially regarding the checksum computation.

2.1.1 Making a TCP fuzzer

A fuzzer for the UDP protocol had already been created by our previous colleague, we could analyze how it had been made and reuse part of the code. One major aspect of making the fuzzer was to compute the correct checksum. The checksum is used to ensure data integrity and reliability when exchanging data through either UDP or TCP protocols. The checksum can give information regarding errors caused by noises or hardware malfunctions for example; and if the calculation is not correct, the packet will be dropped.

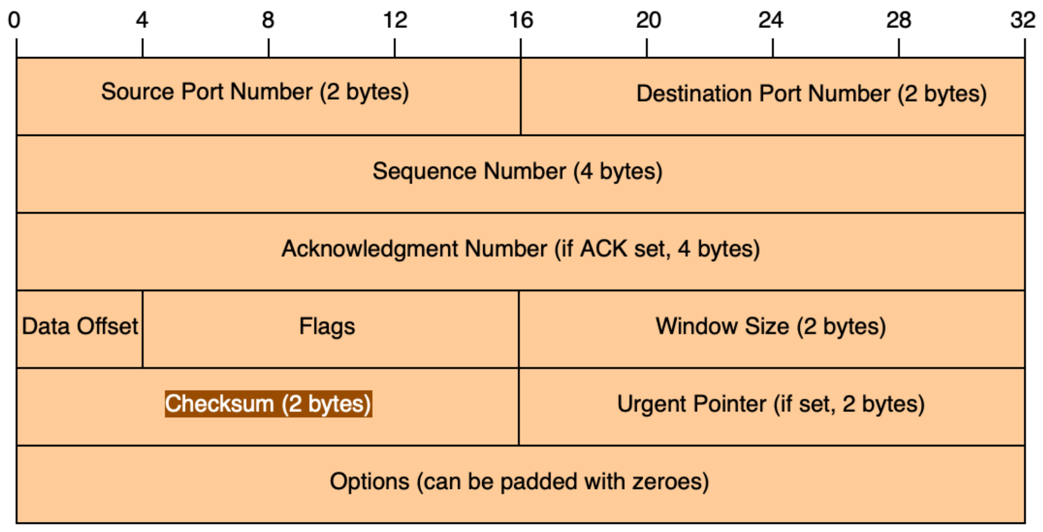
UDP and TCP header packets don't have the same size. While UDP header packets have a consistent size of eight bytes for one packet, TCP header packets can vary in size between twenty bytes to sixty bytes depending on the chosen options.



0
16
32
Source Port Number (2 bytes)
Destination Port Number (2 bytes)
Length (2 bytes)
Checksum (2 bytes)

Figure 2.1: UDP header

If any payload data is to be sent, it will be added to the UDP header. On the other hand, TCP headers contain more information and can be larger:



0
4
8
12
16
20
24
28
32
Source Port Number (2 bytes)
Destination Port Number (2 bytes)
Sequence Number (4 bytes)
Acknowledgment Number (if ACK set, 4 bytes)
Data Offset
Flags
Window Size (2 bytes)
Checksum (2 bytes)
Urgent Pointer (if set, 2 bytes)
Options (can be padded with zeroes)

Figure 2.2: TCP header

```
uint16_t total_length = ntohs(*(uint16_t *) ip_data + 1);
//to get the size of the TCP packet, we have torely on the IP
   -header's length field
uint16_t tcp_size = total_length - (uint16_t)ip_hl_in_bytes;
// total_length != MaxSize
```

The checksum computation in UDP and TCP both take into account the UDP/TCP packet (header and data) and a pseudo IP header :

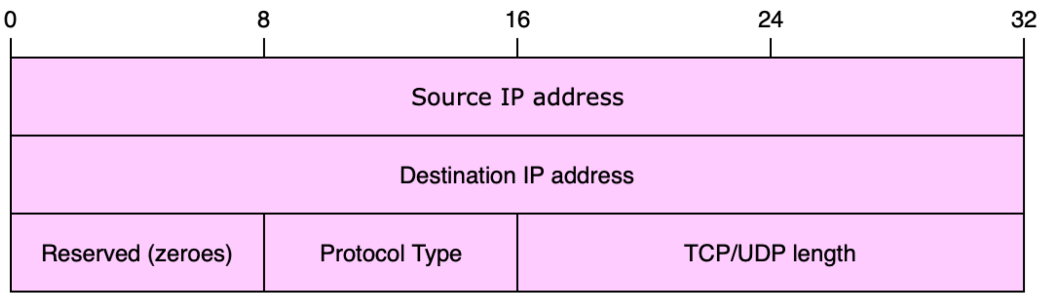


Figure 2.3: Pseudo IP header

Since TCP and UDP take in the same pseudo IP header, we used what was done previously for UDP. The only difference is the length field, since TCP header doesn't have a length field like the UDP header do, we had to :

- take the packet size and divide it by 256 to obtain the most significant byte.
- compute the remainder of the division of packet size by 256 to get the least significant byte.

```
*(Data_to_mutate + mutated_size + 10) = (uint8_t)(
    mutated_size / 256);
*(Data_to_mutate + mutated_size + 11) = (uint8_t)(
    mutated_size % 256);
```

Once the checksum had been modified, we added a trivial printf to check if our computation was correct. The cross indicates when the checksum is wrong and is then dropped:

```
CHECKSUM passed 0
CHECKSUM passed 0
CHECKSUM passed 0
CHECKSUM x : 27153
CHECKSUM x : 13773
CHECKSUM passed 0
CHECKSUM x : 64968
CHECKSUM x : 63944
CHECKSUM x : 23192
CHECKSUM passed 0
CHECKSUM x : 35298
CHECKSUM x : 3523
CHECKSUM x : 22973
CHECKSUM x : 1065
CHECKSUM passed 0
CHECKSUM passed 0
CHECKSUM passed 0
CHECKSUM x : 27153
CHECKSUM x : 13773
CHECKSUM passed 0
CHECKSUM x : 64968
CHECKSUM x : 63944
```

2.1.2 Making header and data only TCP and UDP fuzzers

To fuzz only the header or only the data, we had to compute the size of the header:

- For UDP, it is quite easy since the header size is always 8 bytes.
- For TCP, because of the options, the header size can vary. Because of that, we had to use the OffSet field (in the TCP header), which indicates the number of 4 bytes words in the header. To get the size (in bytes) we just have to multiply that by 4:

```
uint8_t tcp_header_size =
               (*(start_of_tcp + 12) >> 4) * 4;
```

Now we can use that for the fuzzing, for example with TCP:

```
size_t mutated_size = LLVMFuzzerMutate(Data_to_mutate,
    tcp_header_size, tcp_header_size);
```

To get the data size we can simply subtract the length of the TCP/UDP header to the TCP/UDP packet length.

```
size_t mutated_size = LLVMFuzzerMutate(Data_to_mutate +
    tcp_header_size, tcp_data_size, tcp_data_size);
```

The computation of the checksum follows the same process as explained previously.

2.1.3 Making an ICMP fuzzer

We also created an ICMP fuzzer. Unlike UDP and TCP, the checksum computation only takes into account the ICMP header and the data (no pseudo IP-header). Furthermore, ICMP is closer to IP than to TCP or UDP because it operates at the network layer (layer 3) of the OSI model, just like IP.

The ICMP header is located right after the IP header and is structured like follow:

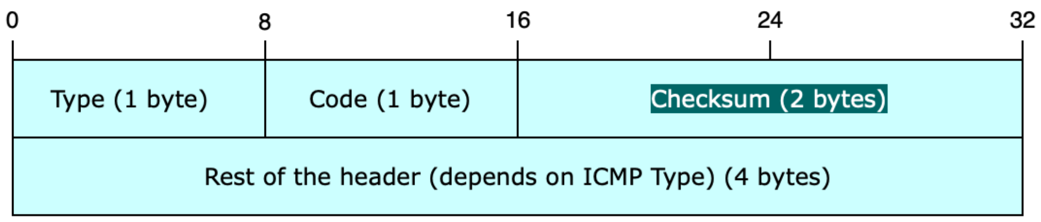


Figure 2.4: ICMP header

Like previously, we reused the base code from J.Marchand for IP. We set at 0 the current checksum using the structure above, and we also defined the size of ICMP, its total length and starting point:

```
uint8_t *start_of_icmp = ip_data + ip_hl_in_bytes;
        uint16_t total_length = ntohs(*((uint16_t *)ip_data +
    1));
        uint16_t icmp_size = (total_length - ip_hl_in_bytes);
```

2.1.4 Making the captures

To implement a fuzzer, we had to think about what we wanted the fuzzer to actually do. What aspects of the program was it meant to test?
An important step in the process of creating a fuzzer is to know what it will test. In order to guide it, we had to feed relevant files so that the fuzzer could modify it while exploring new paths in the code.

We wanted to create captures that presented a very general aspect. To make those captures we used the well-known command **tcpdump**.

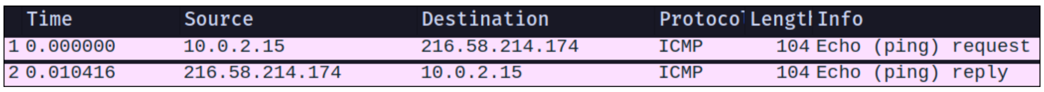
For ICMP, a simple ping request was enough. To do so we pinged **google.com** and captured it.

```
$ sudo tcpdump -w icmp_capture.pcap icmp
```

As well as:

```
$ ping google.com
```

The capture, once cleaned up, looks something like the following:



	Time	Source	Destination	Protoco	Lengtl	Info
1	0.000000	10.0.2.15	216.58.214.174	ICMP	104	Echo (ping) request
2	0.010416	216.58.214.174	10.0.2.15	ICMP	104	Echo (ping) reply

Figure 2.5: ICMP ping request and reply

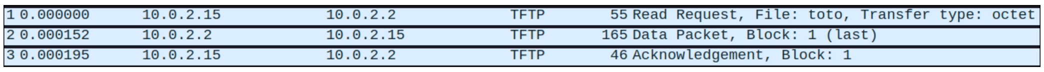
To capture TFTP packets we started by creating a random file /tmp/toto then we ran **qemu** with a debian image and the following options:

```
$ qemu-system-x86_64 -enable-kvm -drive file=debian.img -net
   nic -net user,tftp=/tmp -m 1024
```

Then once inside the VM we installed the TFTP package and ran the command:

```
$ sudo tcpdump -w tftp_capture.pcap &
$ tftp 10.0.2.2 -v -m octet -c get toto
```

Here's the result:



```
1 0.000000      10.0.2.15          10.0.2.2           TFTP        55 Read Request, File: toto, Transfer type: octet
2 0.000152      10.0.2.2           10.0.2.15          TFTP       165 Data Packet, Block: 1 (last)
3 0.000195      10.0.2.15          10.0.2.2           TFTP        46 Acknowledgement, Block: 1
```

Figure 2.6: TFTP capture

Creating a TCP capture revealed to be more challenging. As with our TFTP capture, we began by running a VM using qemu.

```
$ qemu-system-x86_64 -enable-kvm debian.img -net nic -net
   user,guestfwd=tcp:10.0.2.5:1234-cmd:cat -m 1028
```

Then as a guest we ran a basic python code that acted as a TCP client sending 4ko of data to capture multiple packets and we captured it, once again using the same **tcpdump** command.

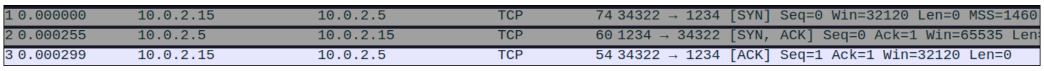
```python
import socket

HOST = '10.0.2.5'
PORT = 1234
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
   client_socket:
  client_socket.setsockopt(socket.SOL_SOCKET, socket.
   SO_SNDBUF, 1460)
  client_socket.connect((HOST, PORT))
  data = b"A"*(4000)
  client_socket.sendall(data)
  print("Sent", len(data), "bytes")
```

A problem we ran into during this capture was the segmentation of the data being sent. The size of the data chunks we were seeing on **Wireshark** were always way above the Maximum Segment Size (MSS) that we wanted (1500 octets).
After trying various different things we finally found the reason for that: TCP Segmentation Offload (TSO). When enabled, the network adapter divides larger data chunks into TCP segments instead of the CPU. To disable it we used the following command:

```
$ sudo ethtool -K eth0 tso off
```

Below is an extract of the TCP capture:



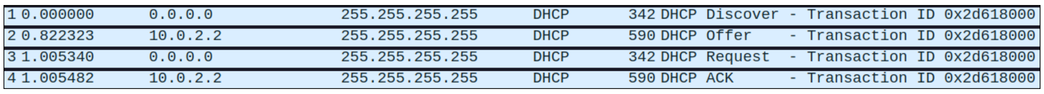
```
1 0.000000    10.0.2.15    10.0.2.5     TCP    74 34322 → 1234 [SYN] Seq=0 Win=32120 Len=0 MSS=1460
2 0.000255    10.0.2.5     10.0.2.15    TCP    60 1234 → 34322 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len
3 0.000299    10.0.2.15    10.0.2.5     TCP    54 34322 → 1234 [ACK] Seq=1 Ack=1 Win=32120 Len=0
```

Figure 2.7: TCP three-way handshake

Last but not least, we also did a DHCP capture. To do so, we executed the following as root:

```
$ tcpdump -w dhcp_capture.pcap &
$ ifdown eth0
$ ifup eth0
```

ifdown eth0 deactivates the network interface eth0. And on the other hand, **ifup eth0** brings it back up activating it. Doing so allows us to capture DHCP traffic.



```
1 0.000000    0.0.0.0     255.255.255.255    DHCP    342 DHCP Discover  - Transaction ID 0x2d618000
2 0.822323    10.0.2.2    255.255.255.255    DHCP    590 DHCP Offer     - Transaction ID 0x2d618000
3 1.005340    0.0.0.0     255.255.255.255    DHCP    342 DHCP Request   - Transaction ID 0x2d618000
4 1.005482    10.0.2.2    255.255.255.255    DHCP    590 DHCP ACK       - Transaction ID 0x2d618000
```

Figure 2.8: DHCP traffic

Now armed with our new captures, we can now try to run the fuzzers.

2.2 Fuzzing only header or data

We added a few more options regarding the choice of protocol to fuzz. We especially focused on IPv4 protocols such as TCP and UDP. One can run the fuzzer on either of these protocols but we added the possibility to run the fuzzers only on data or on the header to target specific aspects. A precision to add is that for TCP fuzzing only the header was relevant as libslirp doesn't currently implement encapsulated protocol in TCP, but we still did implement the fuzzer for future use.

To mutate only the header or data for either UDP or TCP protocols we first computed the size of the header and its total size, then, we computed the mutated size using the LLVMFuzzerMutate function:

```
size_t mutated_size = LLVMFuzzerMutate(Data_to_mutate,
    ref_packet_size, max_size);
```

The second argument is the size of the packet used as a reference while the second is the maximum size the packet can take when being created by the fuzzer. In our case, we want the generated packets to be either only the size of the data or only the size of the header. Depending on the part being fuzzed, we replaced ref_packet_size and max_size by the size protocol_header_size or by protocol_data_size.

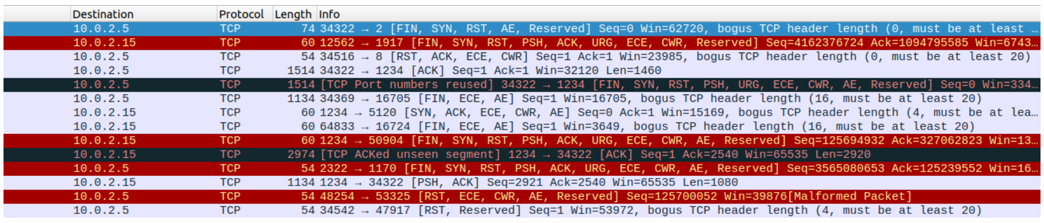
3 What the fuzzers found: an analysis

During our tests we did not encounter any crashes but the fuzzers found interesting packets.

3.1 TCP results

We previously discussed how and which captures were used to guide the fuzzer. All the packets created and saved by the fuzzers are incorrect captures triggering different parts of the code tested.

On the packet capture below, we can see that the packet contains many associated flags:

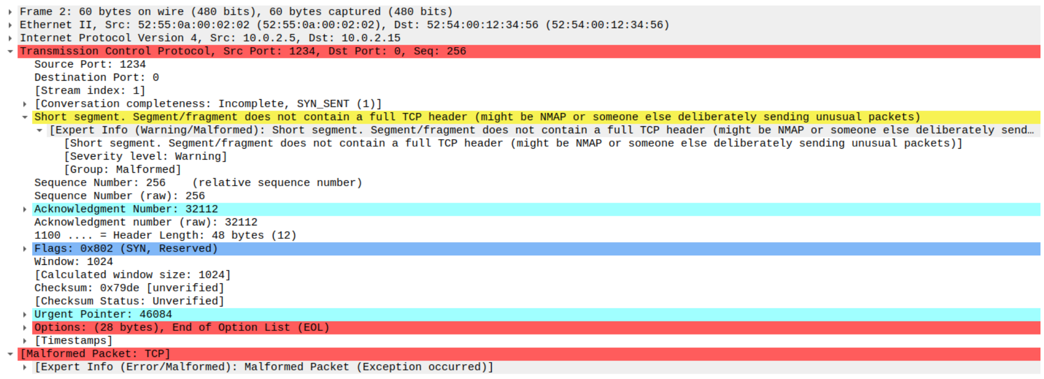


```
Destination      Protocol Length Info
10.0.2.5         TCP        74 34322 → 2 [FIN, SYN, RST, AE, Reserved] Seq=0 Win=62720, bogus TCP header length (0, must be at least …
10.0.2.15        TCP        60 12562 → 1917 [FIN, SYN, RST, PSH, ACK, URG, ECE, CWR, Reserved] Seq=4162376724 Ack=1094795585 Win=6743…
10.0.2.5         TCP        54 34516 → 8 [RST, ACK, ECE, CWR] Seq=1 Ack=1 Win=23985, bogus TCP header length (0, must be at least 20)
10.0.2.5         TCP      1514 34322 → 1234 [ACK] Seq=1 Ack=1 Win=32120 Len=1460
10.0.2.5         TCP      1514 [TCP Port numbers reused] 34322 → 1234 [FIN, SYN, RST, PSH, URG, ECE, CWR, AE, Reserved] Seq=0 Win=334…
10.0.2.5         TCP      1134 34369 → 16705 [FIN, ECE, AE] Seq=1 Win=16705, bogus TCP header length (16, must be at least 20)
10.0.2.15        TCP        60 1234 → 5120 [SYN, ACK, ECE, CWR, AE] Seq=0 Ack=1 Win=15169, bogus TCP header length (4, must be at lea…
10.0.2.15        TCP        60 64833 → 16724 [FIN, ECE, AE] Seq=1 Win=3649, bogus TCP header length (16, must be at least 20)
10.0.2.15        TCP        60 1234 → 50904 [FIN, SYN, RST, PSH, ACK, URG, ECE, CWR, AE, Reserved] Seq=125694932 Ack=327062823 Win=13…
10.0.2.15        TCP      2974 [TCP ACKed unseen segment] 1234 → 34322 [ACK] Seq=1 Ack=2540 Win=65535 Len=2920
10.0.2.5         TCP        54 2322 → 1170 [FIN, SYN, RST, PSH, ACK, URG, ECE, CWR, AE, Reserved] Seq=3565080653 Ack=125239552 Win=16…
10.0.2.15        TCP      1134 1234 → 34322 [PSH, ACK] Seq=2921 Ack=2540 Win=65535 Len=1080
10.0.2.5         TCP        54 48254 → 53325 [RST, ECE, CWR, AE, Reserved] Seq=125700052 Win=39876[Malformed Packet]
10.0.2.5         TCP        54 34542 → 47917 [RST, Reserved] Seq=1 Win=53972, bogus TCP header length (4, must be at least 20)
```

Figure 3.1: TCP Malformed packet 1

As a reminder, flags in TCP are used to describe communications' states. The most well known flags being SYN and ACK used to establish and confirm a connection. Even if flags can sometimes be used together, the combination above is normally impossible. Indeed, SYN is used to initiate a connection while FIN is used to terminate it and RST to reset it, these flags are contradictory.

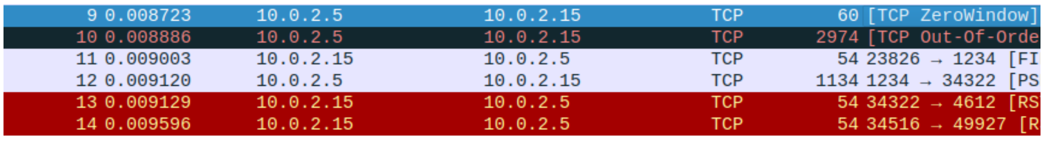
Another interesting example is the following one:



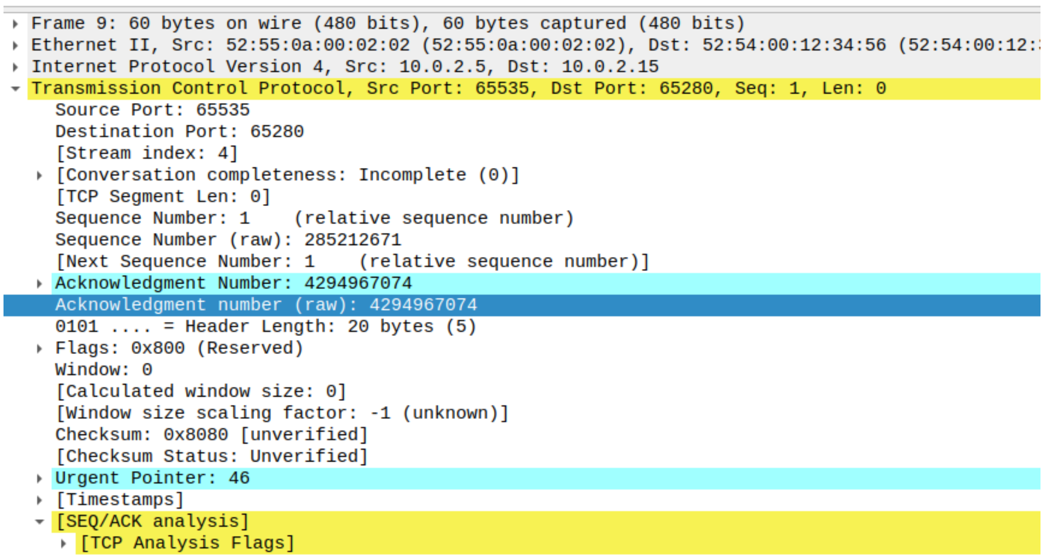
```
▸ Frame 2: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
▸ Ethernet II, Src: 52:55:0a:00:02:02 (52:55:0a:00:02:02), Dst: 52:54:00:12:34:56 (52:54:00:12:34:56)
▸ Internet Protocol Version 4, Src: 10.0.2.5, Dst: 10.0.2.15
▾ Transmission Control Protocol, Src Port: 1234, Dst Port: 0, Seq: 256
    Source Port: 1234
    Destination Port: 0
    [Stream index: 1]
  ▸ [Conversation completeness: Incomplete, SYN_SENT (1)]
  ▾ Short segment. Segment/fragment does not contain a full TCP header (might be NMAP or someone else deliberately sending unusual packets)
    ▾ [Expert Info (Warning/Malformed): Short segment. Segment/fragment does not contain a full TCP header (might be NMAP or someone else deliberately send…
        [Short segment. Segment/fragment does not contain a full TCP header (might be NMAP or someone else deliberately sending unusual packets)]
        [Severity level: Warning]
        [Group: Malformed]
    Sequence Number: 256    (relative sequence number)
    Sequence Number (raw): 256
  ▸ Acknowledgment Number: 32112
    Acknowledgment number (raw): 32112
    1100 .... = Header Length: 48 bytes (12)
  ▸ Flags: 0x802 (SYN, Reserved)
    Window: 1024
    [Calculated window size: 1024]
    Checksum: 0x79de [unverified]
    [Checksum Status: Unverified]
  ▸ Urgent Pointer: 46084
  ▸ Options: (28 bytes), End of Option List (EOL)
  ▸ [Timestamps]
▾ [Malformed Packet: TCP]
  ▸ [Expert Info (Error/Malformed): Malformed Packet (Exception occurred)]
```

Figure 3.2: TCP Malformed packet 2

The packet's header is incomplete, hence it is malformed and wiresharks detects it as potentially dangerous. Indeed, sending unusual packets to a target to analyze its responding behavior is common when wanting to launch an attack.



```
    9 0.008723      10.0.2.5          10.0.2.15         TCP         60 [TCP ZeroWindow]
   10 0.008886      10.0.2.5          10.0.2.15         TCP       2974 [TCP Out-Of-Orde
   11 0.009003      10.0.2.15         10.0.2.5          TCP         54 23826 → 1234 [FI
   12 0.009120      10.0.2.5          10.0.2.15         TCP       1134 1234 → 34322 [PS
   13 0.009129      10.0.2.15         10.0.2.5          TCP         54 34322 → 4612 [RS
   14 0.009596      10.0.2.15         10.0.2.5          TCP         54 34516 → 49927 [R
```



```
▸ Frame 9: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
▸ Ethernet II, Src: 52:55:0a:00:02:02 (52:55:0a:00:02:02), Dst: 52:54:00:12:34:56 (52:54:00:12:
▸ Internet Protocol Version 4, Src: 10.0.2.5, Dst: 10.0.2.15
▾ Transmission Control Protocol, Src Port: 65535, Dst Port: 65280, Seq: 1, Len: 0
    Source Port: 65535
    Destination Port: 65280
    [Stream index: 4]
  ▸ [Conversation completeness: Incomplete (0)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 285212671
    [Next Sequence Number: 1    (relative sequence number)]
  ▸ Acknowledgment Number: 4294967074
    Acknowledgment number (raw): 4294967074
    0101 .... = Header Length: 20 bytes (5)
  ▸ Flags: 0x800 (Reserved)
    Window: 0
    [Calculated window size: 0]
    [Window size scaling factor: -1 (unknown)]
    Checksum: 0x8080 [unverified]
    [Checksum Status: Unverified]
  ▸ Urgent Pointer: 46
  ▸ [Timestamps]
  ▾ [SEQ/ACK analysis]
    ▸ [TCP Analysis Flags]
```

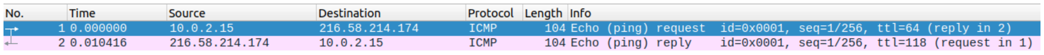
Figure 3.3: TCP Malformed packet 3

And lastly, we can see here that the window size is 0 ([TCP ZeroWindow]) meaning that the receiver's TCP buffer is full and therefore it cannot accept more data. Yet we can see here that our fuzzer keeps sending data to possibly trigger a buffer overflow, which is a very commonly used attack method.

The error was well handled and no crash occurred meaning that the code doesn't have any vulnerability related to the previous packet misconfigurations.

3.2 ICMP results

With this fuzzer, we rapidly obtained a lot of interesting files. The capture fed was a simple ping request and reply like so:

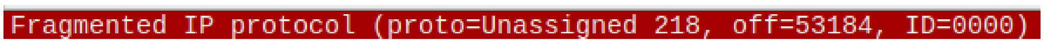


```
No.  Time      Source          Destination     Protocol Length Info
   1 0.000000  10.0.2.15       216.58.214.174  ICMP      104 Echo (ping) request  id=0x0001, seq=1/256, ttl=64 (reply in 2)
   2 0.010416  216.58.214.174  10.0.2.15       ICMP      104 Echo (ping) reply    id=0x0001, seq=1/256, ttl=118 (request in 1)
```

Figure 3.4: ICMP Malformed packet 1

This initial packet was of size 264 ko; but when looking at the size of the obtained files we noticed that almost all packets were of size 132 ko or less. Why? One hypothesis could be the following: the reply doesn't affect the request, hence it will always be the same and is useless in our bug search. Maybe the fuzzer understood that and decided to focus only on what's useful: the request.

Here is an example of what the fuzzer generated: the request is fragmented and the protocol field which corresponds to the protocol and is either IPv4 or IPv6 is unassigned:



```
Fragmented IP protocol (proto=Unassigned 218, off=53184, ID=0000)
```

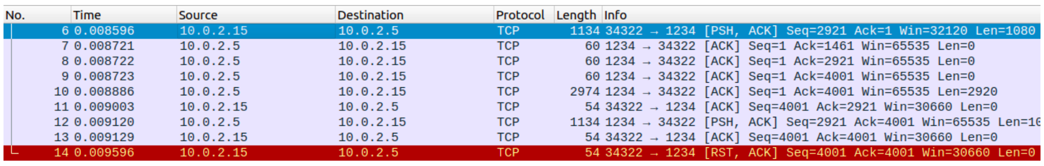
Figure 3.5: Excerpt of an ICMP capture

More examples of what the fuzzer generated can be found in the IN_ICMP folder.

3.3 Only data or header results

The packets that we got from only fuzzing the header in TCP and UDP did not show any significant difference compared to the ones we got from fuzzing both the data and the header.

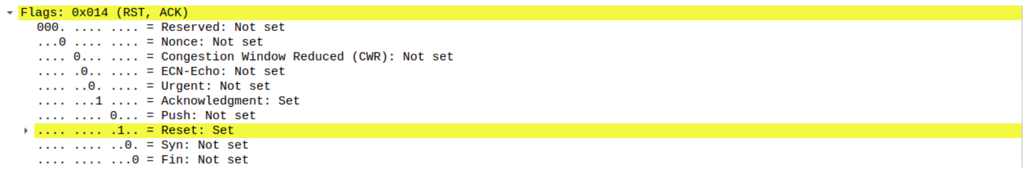
For the data only TCP fuzzer, we did not get that many files (4 in 10 minutes). This is probably because libslirp doesn't actually implement the protocol embed in TCP, so only the TCP header only matters, and not so much the data. The files contains only one malformed packed :



```
No.  Time      Source     Destination Protocol Length Info
   6 0.008596  10.0.2.15  10.0.2.5    TCP      1134 34322 → 1234 [PSH, ACK] Seq=2921 Ack=1 Win=32120 Len=1080
   7 0.008721  10.0.2.5   10.0.2.15   TCP        60 1234 → 34322 [ACK] Seq=1 Ack=1461 Win=65535 Len=0
   8 0.008722  10.0.2.5   10.0.2.15   TCP        60 1234 → 34322 [ACK] Seq=1 Ack=2921 Win=65535 Len=0
   9 0.008723  10.0.2.5   10.0.2.15   TCP        60 1234 → 34322 [ACK] Seq=1 Ack=4001 Win=65535 Len=0
  10 0.008886  10.0.2.5   10.0.2.15   TCP      2974 1234 → 34322 [ACK] Seq=1 Ack=4001 Win=65535 Len=2920
  11 0.009003  10.0.2.15  10.0.2.5    TCP        54 34322 → 1234 [ACK] Seq=4001 Ack=2921 Win=30660 Len=0
  12 0.009120  10.0.2.5   10.0.2.15   TCP      1134 1234 → 34322 [PSH, ACK] Seq=2921 Ack=4001 Win=65535 Len=10
  13 0.009129  10.0.2.15  10.0.2.5    TCP        54 34322 → 1234 [ACK] Seq=4001 Ack=4001 Win=30660 Len=0
  14 0.009596  10.0.2.15  10.0.2.5    TCP        54 34322 → 1234 [RST, ACK] Seq=4001 Ack=4001 Win=30660 Len=0
```

Figure 3.6: TCP (data only), malformed packet

The issue seems to stem from the flags:



```
Flags: 0x014 (RST, ACK)
  000. .... .... = Reserved: Not set
  ...0 .... .... = Nonce: Not set
  .... 0... .... = Congestion Window Reduced (CWR): Not set
  .... .0.. .... = ECN-Echo: Not set
  .... ..0. .... = Urgent: Not set
  .... ...1 .... = Acknowledgment: Set
  .... .... 0... = Push: Not set
  .... .... .1.. = Reset: Set
  .... .... ..0. = Syn: Not set
  .... .... ...0 = Fin: Not set
```

Figure 3.7: TCP (data only), flags issue

So as we said just before, it really seems that libslirp only looks for issues with the TCP header and not the data.

The fuzzing of only the data in UDP only gave us an empty file (in five minutes of fuzzing).

4 Improving code

Another part of improving the fuzzers was to improve its usability. To make it more clear and easy as one wants to focus on finding bugs and gain time by automating trivial tasks.

4.1 Meson files, scripts and readme

By creating new fuzzers, we increased the number of options of fuzzers to be run and so we modified the corresponding meson file to make it more easy to run and then analyze. Until now, the both IP and UDP relevant files found by the fuzzers were redirected in one folder "IN". To make it more clear for post-fuzzing files analysis, we created one subfolder for each protocol.

It will be easier for the fuzzers as each folder contains captures we made that will be used as "pattern example" but also for us as we will go directly look in a certain folder to obtain results based on a protocol.The results of the fuzzers would now be stored in corresponding directories, such as IN_TCP/Header_Only or IN_ICMP.

Previously, to run a fuzzer, the following command line was used:

```
build/fuzzing/fuzz-icmp fuzzing/IN_ICMP -detect_leaks=0
```

But it could get longer when specifying options. This is why we created a script (fuzz.sh) where you only need to specify the fuzzer you want to run making it more understandable and easy to use:

```
build/fuzzing/fuzz-tcp-d fuzzing/IN_TCP/Data_Only -
    detect_leaks=0
```

Became

```
./fuzz.sh tcp-d
```

You can find all of these modifications and new additions in the readme file.

4.2 Code factorization and coverage

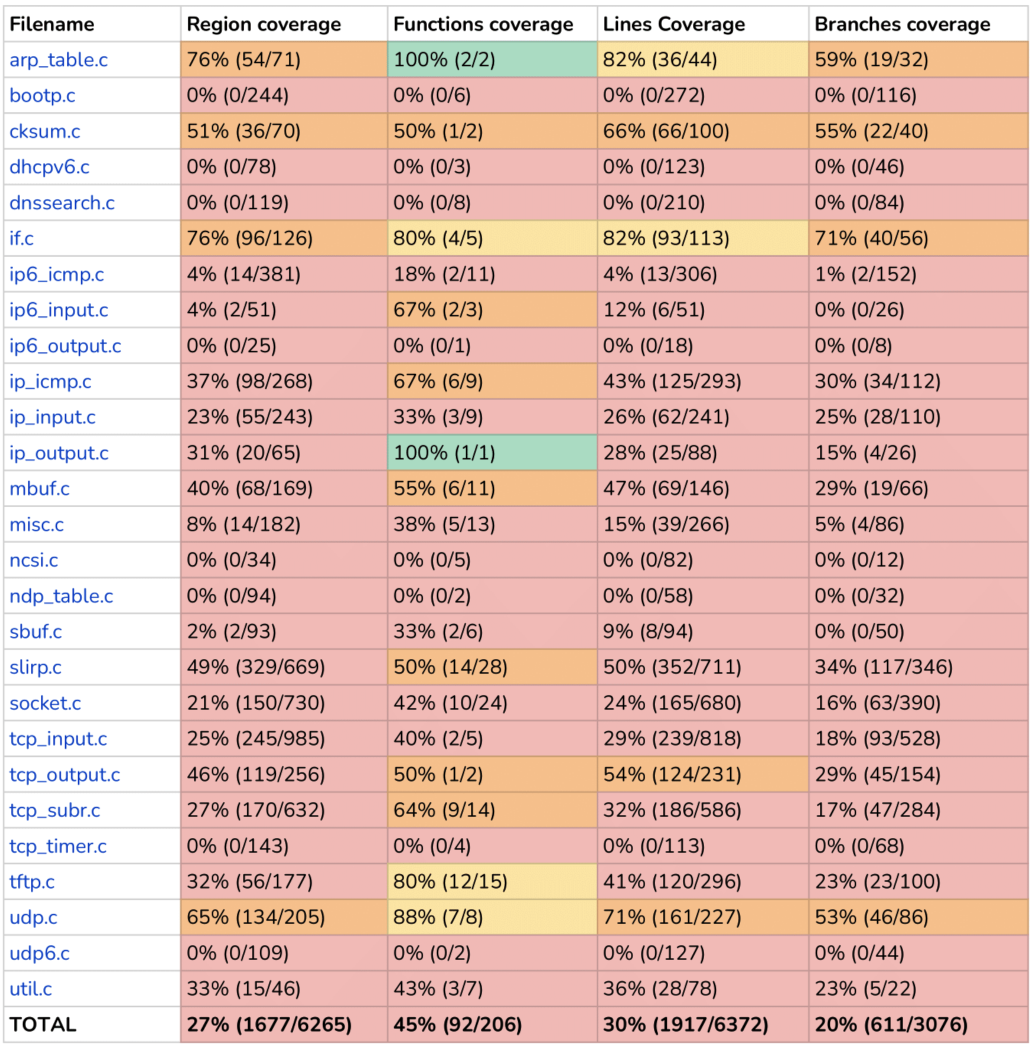
Another way of improving readability for future colleagues was to factorize code. The structures, used for the fuzzers, were the same among all of them, so we grouped them under one header file: "slirp_base_fuzz.h". Even if the fuzzers had similar code, we couldn't factorize further as each code was slightly different for one another.

Before, the coverage.py file was implemented in such a way that it would only focus on UDP but with the addition of new fuzzers, we had to adapt the coverage file so that it could cover all the current fuzzers, but also future fuzzers if they were to be added. The usage can also be found in the readme file.

5 Conclusion

5.1 Coverage results

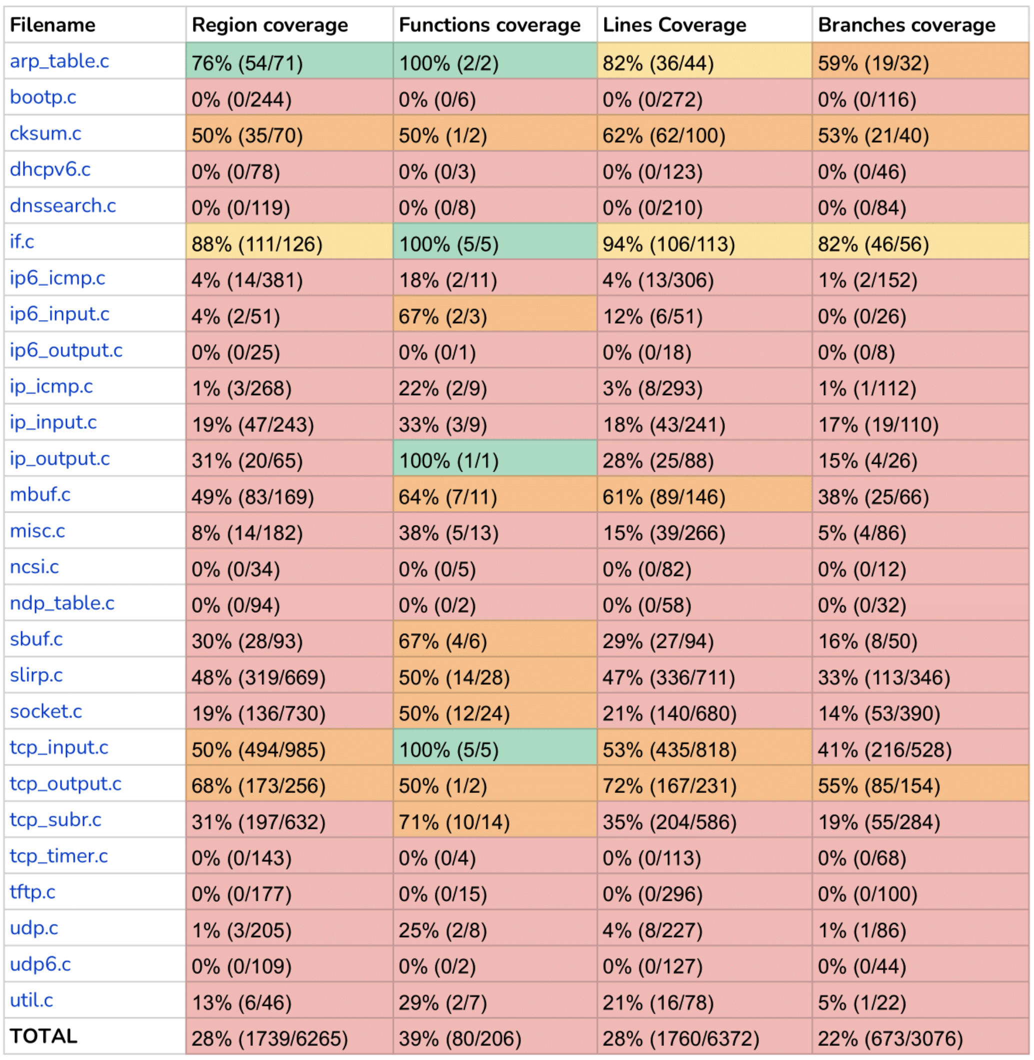
The first time we ran the coverage file on the UDP fuzzer we found many missed branches and functions related to other protocols:



Filename	Region coverage	Functions coverage	Lines Coverage	Branches coverage
arp_table.c	76% (54/71)	100% (2/2)	82% (36/44)	59% (19/32)
bootp.c	0% (0/244)	0% (0/6)	0% (0/272)	0% (0/116)
cksum.c	51% (36/70)	50% (1/2)	66% (66/100)	55% (22/40)
dhcpv6.c	0% (0/78)	0% (0/3)	0% (0/123)	0% (0/46)
dnssearch.c	0% (0/119)	0% (0/8)	0% (0/210)	0% (0/84)
if.c	76% (96/126)	80% (4/5)	82% (93/113)	71% (40/56)
ip6_icmp.c	4% (14/381)	18% (2/11)	4% (13/306)	1% (2/152)
ip6_input.c	4% (2/51)	67% (2/3)	12% (6/51)	0% (0/26)
ip6_output.c	0% (0/25)	0% (0/1)	0% (0/18)	0% (0/8)
ip_icmp.c	37% (98/268)	67% (6/9)	43% (125/293)	30% (34/112)
ip_input.c	23% (55/243)	33% (3/9)	26% (62/241)	25% (28/110)
ip_output.c	31% (20/65)	100% (1/1)	28% (25/88)	15% (4/26)
mbuf.c	40% (68/169)	55% (6/11)	47% (69/146)	29% (19/66)
misc.c	8% (14/182)	38% (5/13)	15% (39/266)	5% (4/86)
ncsi.c	0% (0/34)	0% (0/5)	0% (0/82)	0% (0/12)
ndp_table.c	0% (0/94)	0% (0/2)	0% (0/58)	0% (0/32)
sbuf.c	2% (2/93)	33% (2/6)	9% (8/94)	0% (0/50)
slirp.c	49% (329/669)	50% (14/28)	50% (352/711)	34% (117/346)
socket.c	21% (150/730)	42% (10/24)	24% (165/680)	16% (63/390)
tcp_input.c	25% (245/985)	40% (2/5)	29% (239/818)	18% (93/528)
tcp_output.c	46% (119/256)	50% (1/2)	54% (124/231)	29% (45/154)
tcp_subr.c	27% (170/632)	64% (9/14)	32% (186/586)	17% (47/284)
tcp_timer.c	0% (0/143)	0% (0/4)	0% (0/113)	0% (0/68)
tftp.c	32% (56/177)	80% (12/15)	41% (120/296)	23% (23/100)
udp.c	65% (134/205)	88% (7/8)	71% (161/227)	53% (46/86)
udp6.c	0% (0/109)	0% (0/2)	0% (0/127)	0% (0/44)
util.c	33% (15/46)	43% (3/7)	36% (28/78)	23% (5/22)
TOTAL	**27% (1677/6265)**	**45% (92/206)**	**30% (1917/6372)**	**20% (611/3076)**

Figure 5.1: UDP coverage

Of course, this coverage is for the UDP protocol, but it gave us an oversight of the possibilities we could try to test with our fuzzer. Our main goal was to explore more functions and branches and possibly find crashes. As we saw previously we haven't found any crashes, but we did increase the number of paths taken by the fuzzer:



Filename	Region coverage	Functions coverage	Lines Coverage	Branches coverage
arp_table.c	76% (54/71)	100% (2/2)	82% (36/44)	59% (19/32)
bootp.c	0% (0/244)	0% (0/6)	0% (0/272)	0% (0/116)
cksum.c	50% (35/70)	50% (1/2)	62% (62/100)	53% (21/40)
dhcpv6.c	0% (0/78)	0% (0/3)	0% (0/123)	0% (0/46)
dnssearch.c	0% (0/119)	0% (0/8)	0% (0/210)	0% (0/84)
if.c	88% (111/126)	100% (5/5)	94% (106/113)	82% (46/56)
ip6_icmp.c	4% (14/381)	18% (2/11)	4% (13/306)	1% (2/152)
ip6_input.c	4% (2/51)	67% (2/3)	12% (6/51)	0% (0/26)
ip6_output.c	0% (0/25)	0% (0/1)	0% (0/18)	0% (0/8)
ip_icmp.c	1% (3/268)	22% (2/9)	3% (8/293)	1% (1/112)
ip_input.c	19% (47/243)	33% (3/9)	18% (43/241)	17% (19/110)
ip_output.c	31% (20/65)	100% (1/1)	28% (25/88)	15% (4/26)
mbuf.c	49% (83/169)	64% (7/11)	61% (89/146)	38% (25/66)
misc.c	8% (14/182)	38% (5/13)	15% (39/266)	5% (4/86)
ncsi.c	0% (0/34)	0% (0/5)	0% (0/82)	0% (0/12)
ndp_table.c	0% (0/94)	0% (0/2)	0% (0/58)	0% (0/32)
sbuf.c	30% (28/93)	67% (4/6)	29% (27/94)	16% (8/50)
slirp.c	48% (319/669)	50% (14/28)	47% (336/711)	33% (113/346)
socket.c	19% (136/730)	50% (12/24)	21% (140/680)	14% (53/390)
tcp_input.c	50% (494/985)	100% (5/5)	53% (435/818)	41% (216/528)
tcp_output.c	68% (173/256)	50% (1/2)	72% (167/231)	55% (85/154)
tcp_subr.c	31% (197/632)	71% (10/14)	35% (204/586)	19% (55/284)
tcp_timer.c	0% (0/143)	0% (0/4)	0% (0/113)	0% (0/68)
tftp.c	0% (0/177)	0% (0/15)	0% (0/296)	0% (0/100)
udp.c	1% (3/205)	25% (2/8)	4% (8/227)	1% (1/86)
udp6.c	0% (0/109)	0% (0/2)	0% (0/127)	0% (0/44)
util.c	13% (6/46)	29% (2/7)	21% (16/78)	5% (1/22)
TOTAL	28% (1739/6265)	39% (80/206)	28% (1760/6372)	22% (673/3076)

Figure 5.2: TCP coverage

We can notice a couple of things: the tcp_input.c function was fully explored, other tcp related functions such as socket, or tcp_subr also increased. Some functions such as cksum.c did not increase. Indeed, the cksum function either computes the IPv4 or the IPv6 checksum, and so far we have only worked on IPv4.

Overall, the coverage of the TCP protocol is quite good.

5.2 Future things to do

We hope that in the future other programmers and students will contribute to this project. Because we were limited in time we could only focus on some aspects of the project. Hence, it would be interesting to try to improve the coverage of the IPv6 based functions.

For both TCP and UDP, it is important to note that the total length is different from MaxSize. When creating packets to fuzz the Mutating function takes two size parameters as arguments, one is the actual size of the packet provided and the other is the maximum size of the packet generated by the fuzzer. In our case, we limit the packet size to its current size; but it would be interesting to try giving more freedom to the fuzzer to see what would happen with longer packets. To do that, we should modify the size to which a packet can be extended to MaxSize minus the size of the IP_header.

Moreover, packets transferred using TCP protocol are assigned a random number as declared by the RFC, but in our case this causes a problem as the implementation of it by the libslirp does not work as we would like. An idea would be to disable this random number assignment but we would need to generate new traces corresponding to this new structure.

As mentioned by J.Marchand in his report he suggested the idea of creating one custom mutator that would handle multiple protocols at once. He also found memory leaks during his analysis.

5.3 What we learnt

Through this project we managed to fulfill two goals. First, we learnt about a powerful and commonly used troubleshooting technique and tool. Understanding what are the principles and methods used to fuzz and how it can impact overall security is crucial. With the acquired knowledge we'll be able to apply it to our future work.

Moreover, it also gave us the opportunity to work on an open source project for the first time and gave us a little insight of how projects are managed in the professional world. Thinking that this project will be improved by other people in the future and that our code will be reused is rewarding. And lastly, we wish to express our thanks to Mr. Samuel Thibault for his guidance and patience with us.

6 Appendix

libslirp gitlab link:
https://gitlab.freedesktop.org/sthibaul/libslirp

fuzzing branch link:
https://gitlab.freedesktop.org/sthibaul/libslirp/-/tree/fuzzing?ref_type=heads

libfuzzer:
https://llvm.org/docs/LibFuzzer.html

Explanations on different protocols:
https://web.maths.unsw.edu.au/~lafaye/CCM/internet/tcpip.htm

Networking in qemu:
https://wiki.qemu.org/index.php/Documentation/Networking

Flags in TCP:
https://www.it-connect.fr/chapitres/gerer-les-flags-tcp-et-licmp-avec-nftables/

Marchand, Jérémy (2021) *Attaquons libslirp, la pile tcp/ip de qemu*