GSoC 2020: Report-2: Fuzzing the NetBSD Network Stack in a Rumpkernel Environment

Introduction:

The objective of this project is to fuzz the various protocols and layers of the network stack of NetBSD using rumpkernel. This project is being carried out as a part of GSoC 2020. This blog post is regarding the project, the concepts and tools involved, the objectives, the current progress and the next steps.
You can read the previous post/report here.

Overview of the work done:

The major time of phase 1 and 2 was spent in analyzing the input and output paths of the particular protocols being fuzzed. During that time, 5 major protocols of the internet stack were taken up:

1. IPv4 (Phase 1)
2. UDP (Phase 1)
3. IPv6 (Phase 2)
4. ICMP (Phase 2)
5. Ethernet (Phase 2)

Quite a good amount of time was spent in understanding the input and output processing functions of the particular protocols, the information gathered was to be applied in packet creation code for that protocol. This is important so that we know which parts of the packet can be kept random by the fuzzer based input and which part of the packet need to be set to properly fixed values. Fixing some values in the data packet to be correct is important so that the packet does not get rejected for trivial cases like IP Protocol Version or Internet Checksum. (The procedure to come up with the decisions and the code design and flow is explained in IPv4 Protocol section as an example)

For each protocol, mainly 2 things needed to be implemented:

1. The Network Config: the topology for sending and receiving packets example using a TUN device or a TAP device, the socket used, and so on. Configuring these devices was the first step in being able to send or receive packets
2. Packet Creation: Using the information gathered in the code walkthrough of the protocol functions, packet creation is decided where certain parts of the packet are kept fixed and others random from the fuzzer input itself. Doing so we try to gain maximum code coverage. Also, one thing to be noted here, we should not randomly change the fuzzer

input, rather do it deterministically following the same rules for each packet, otherwise the evolutionary fuzzer cannot easily grow the corpus.

In the next section, a few of the protocols will be explained in detail.

Protocols

In this section, we will talk about the various protocols implemented for fuzzing and talk about the approach taken to create a packet.

IPv4:

IPv4 stands for the Internet protocol version 4. It is one of the most widely used protocols in the internet family of protocols. It is used as a network layer protocol for routing and host to host packet delivery using an addressing scheme called IP Address(A 32-bit address scheme). IP Protocol also handles a lot of other functions like fragmentation and reassembly of packets to accommodate for transmission of packets over varying sizes of physical channel capacities. It also supports the concept of multicasting and broadcasting (Via IP Options).

In order to come up with a strategy for fuzzing, the first step was to carry out a code walkthrough of relevant functions/APIs and data structures involved in the IPv4 protocol. For that the major files and components studied were:

- ip_input() => Which carries out the processing of an incoming packet at the network layer for IPv4 (src here)
- ip_output() => Which carries out the processing of an outgoing packet at the network layer for IPv4 (src here)
- struct ip => Represents the IP header (src here)

These sections of code represent the working of the input and output processing paths of IPv4 protocol and the struct ip is the main IPv4 header. On top of that other APIs related to mbuf (The NetBSD packet), ip_forward(), IP assembly and fragmentation, etc. were also studied in order to determine information about packet structure that could be followed.

In order to be able to reach these various aspects of the protocol and be able to fuzz it, we went forward with packet creation that took care of basic fields of the IP header so that it would not get rejected in trivial cases as mentioned before. Hence we went ahead and fixed these fields:

- *IP Version*: Set it to 0x4 which is a 4-bit value.
- *IP Header Len*: Which is set to a value greater than or equal to sizeof(struct ip). Setting this to greater than that allows for IP Options processing.
- *IP Len*: Set it to the size of the random buffer passed by fuzzer.
- *IP Checksum*: We calculate the correct checksum for the packet using the internet checksum algorithm.

Other fields were allowed to be populated randomly by fuzzer input. Here is an illustration of the IPv4 header with the fields marked in red as fixed.

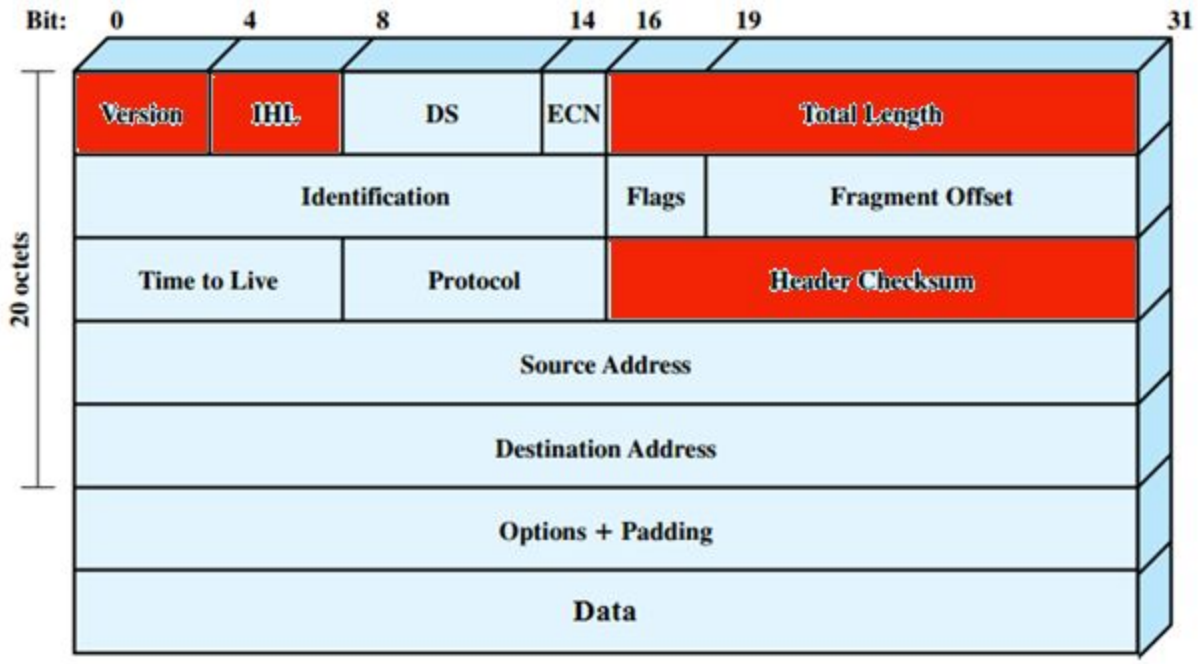

Figure IPv4 Header

The packet creation code lies in the following section inside [pkt_create.c]. Another important component is the network configuration [located here net_config] where the code related to configuring a TUN/TAP device is present. All the code uses the rumpkernel exposed APIs and syscalls (prepended with rump_sys_) so as to utilize the rumpkernel while executing the application binary. After packet creation and network config is handled the main fuzzing function is written where a series of steps are followed:

1. We call rump_init() to initialize the rumpkernel linked via libraries
2. We set up the Client and server IP addresses
3. We set up the TUN device by calling the network config functions described above
4. We create the packet using the packet creation function utilizing the random buffer passed by the fuzzer and transforming that into a semi-random buffer.
5. Pass this forged packet into the network stack of the rumpkernel linked with the application binary by calling rump_sys_write on the TUN device setup.

IPv6:

IPv6 stands for the Internet protocol version 4. It is the successor of the IPv4 protocol. It came into existence in order to overcome the addressing requirements that could not fit in a 32-bit IPv4 address. It is used as a network layer protocol for routing and host to host packet delivery using an addressing scheme called IPv6 Address(A 128-bit address scheme). It also supports

almost similar other functions as IPv4 except for some things like fragmentation, broadcast(instead uses multicast).

In order to be able to reach these various aspects of the protocol and be able to fuzz it, we went forward with packet creation that took care of basic fields of the IP header so that it would not get rejected in trivial cases as mentioned before. Hence we went ahead and fixed these fields:

- *IP Version*: Set it to 0x6 which is a 4-bit value.
- *IP Hop Limit*: This is an alias for TTL. Set it to a maximum possible value of 255(8 bits).

Other fields were allowed to be populated randomly by fuzzer input. Allowing the payload length value to be randomly populated allowed the processing of various “next headers” or ”Extension headers”. Extension headers carry optional Internet Layer information and are placed between the fixed header and the upper-layer protocol header. The headers form a chain, using the Next Header fields. The Next Header field in the fixed header indicates the type of the first extension header; the Next Header field of the last extension header indicates the type of the upper-layer protocol header in the payload of the packet. Further work can be done to set the value of the next header chain and form packets for multiple scenarios with a combination of various next headers.

UDP:

UDP stands for User Datagram Protocol. It is one of the simplest protocols and is designed to be simple so that it simply carries payload with minimal overhead. It does not have many options except for checksum information and ports in order to demultiplex the packet to the processes.
Since UDP runs at the transport layer and hence is wrapped up in an IP header. Since we do not want to fuzz the IP code section, we form a well-formed IP header so that the packet does not get rejected in the IP processing section. We only randomize the UDP header using the fuzzer input. We used previously built out IP packet creation utilities to form the IP header and then use the fuzzer input for the UDP header.

In UDP, we fix the following fields:

- *UDP Checksum*: Set it to zero in order to avoid checksums.

ICMP:

ICMP stands for Internet control message protocol. This protocol is sometimes called a sister protocol of IP protocol and is used as a troubleshooting protocol at the network layer. It is used for major 2 purposes:

1. Error messages
2. Request-Reply Queries.

ICMP has a lot of options and is quite generic in the sense that it handles a lot of error messages and queries. Although ICMP is generally considered at the network layer, it is

actually wrapped inside an IP header, hence it has its own protocol number(= 1). Again similar to UDP, we wrap the ICMP headers inside IP headers, hence we do not randomize the IP header and only the ICMP headers using fuzzer input.

In order to test various ICMP messages and queries, we could not fix values for the *type* and *code* fields in the ICMP header since they decide the ICMP message type. Also if we allowed random input, most of the packets would get rejected since the number of options of type and code fields is limited and most other values would discard the packet while processing. Hence we came up with a solution where we *deterministically* modified the input bits from the fuzzer corresponding to the *code* and *type* fields. For the *type* field, we simply took a modulo of the number of types(ICMP_NTYPES macro used here). For the value of *code*, we had to fix values in a certain range based on the *type* value set already. This technique allowed us to cover all different ICMP message types via the fuzzer input. We also ensured that the input buffer was not modified completely randomly, since that is a bad practice for a feedback-driven fuzzer like ours. Apart from this, we fixed the ICMP Checksum field as well by calculating the checksum using the internet checksum algorithm.

Ethernet:

Ethernet protocol defined by the IEEE 802.3 standard is a widely used data link layer protocol. The ethernet packet called a frame carries an IP(or the network layer protocol) datagram. The header is simple with Link-Layer Addresses called MAC address (used for switching at data link layer which is a part of addressing), for source and destination each of 6 octets(=48 bytes) present, followed by a 4 octet Ethertype and QTag field. This is followed by payload and finally, the FCS(frame check sequence) which is a four-octet cyclic redundancy check (CRC) that allows detection of corrupted data within the entire frame as received on the receiver side.

In the case of Ethernet protocol fuzzing, we had to use a TAP device instead of a TUN device, since the TUN device supports passing an IP packet to the network stack, whereas a TAP device accepts an ethernet frame.

For packet creation, we set the source and destination MAC address and let the payload and ethertype be randomly populated by the fuzzer.

Current Progress and Next steps

The project currently has reached a stage where many major internet family protocols have been covered for fuzzing. As described above a structured approach to fuzzing them has been taken by forming packets based on the internal workings of the protocols. Also as mentioned in the previous post, the Rumpkernel environment is being used for fuzzing all these protocols. In

order to get better results as compared to raw fuzzing, we have taken these steps. In the next report, we shall talk about and compare the coverage of raw fuzzing with our approach.

For the next phase of GSoC, the major focus would be to validate this process of fuzzing by various methods to check the penetration of packets into the network stack as well as the code coverage. Also, the code would be made more streamlined and standardized so that it can be extended for adding more protocols even beyond the scope of the GSoC project.