GSoC 2020: Report-1: 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 and the current progress and next steps.

Fuzzing:

Fuzzing or fuzz testing is an automated software testing technique in which a program is tested by passing unusual, unexpected or semi-random input generated data to the input of the program and repeatedly doing so, trying to crash the program and detect potential bugs or undealt corner cases.
There are several tools available today that enable this which are known as fuzzers. An effective fuzzer generates semi-valid inputs that are "valid enough" in that they are not directly rejected by the parser, but do create unexpected behaviors deeper in the program and are "invalid enough" to expose corner cases that have not been properly dealt with.

Fuzzers can be of various types like dumb vs smart, generation-based vs mutation-based and so on. A dumb fuzzer generates random input without looking at the input format or model but it can follow some sophisticated algorithms like in AFL, though considered a dumb fuzzer as it just flips bits and replaces bytes, still uses a genetic algorithm to create new test cases, where as a smart fuzzer will follow an input model to generate semi-random data that can penetrate well in the code and trigger more edge cases. Mutation and generation fuzzers handle test case generation differently. Mutation fuzzers mutate a supplied seed input object, while generation fuzzers generate new test cases from a supplied model.

Some examples of popular fuzzers are: AFL(American Fuzzy Lop), Syzkaller, Honggfuzz.

RumpKernel

Kernels can have several different architectures like monolithic, microkernel, exokernel etc. An interesting architecture is the “anykernel” architecture, according to wikipedia, “The "anykernel" concept refers to an architecture-agnostic approach to drivers where drivers can either be compiled into the monolithic kernel or be run as a userspace process, microkernel-style, without

code changes.” Rumpkernel is an implementation of this anykernel architecture. In case of the NetBSD rumpkernel, all the NetBSD subsystems like file system, network stack, drivers etc are compiled into standalone libraries that can be linked into any application process to utilize the functionalities of the kernel, like the file system or the drivers. This allows us to run and test various components of NetBSD kernel as a library linked to programs running in the user space.

This idea of rumpkernel is really helpful in fuzzing the components of the kernel. We can fuzz separate subsystems and allow it to crash without having to manage the crash of a running Operating system. Also the fact that context switching has an overhead in case syscalls are made to the kernel, Rumpkernel running in the userspace can eliminate this and save time.(Also since the spectre-meltdown vulnerabilities, system calls have become more costly due to the security reasons)

HonggFuzz + Rumpkernel Network Stack:

In this project we will use the outlined Rumpkernel’s network stack and a fuzzer called the honggfuzz. Honggfuzz is a security oriented, feedback-driven, evolutionary, easy-to-use fuzzer. It is maintained by google.(https://github.com/google/honggfuzz)

The project is hosted on github at: https://github.com/NJnisarg/fuzznetrump/ .The Readme can help in setting it up. We first require NetBSD installed either on a physical machine or on a virtual machine like qemu. Then we will build the NetBSD distribution by grabbing the latest sources(https://github.com/NetBSD/src). We will enable fuzzer coverage by using MKSANITIZER and MKLIBCSANITIZER flags and using the ASan(Address) and UBSan(Undefined Behavior) sanitizers. These sanitizers will help the fuzzer in catching bugs related to undefined behavior and address and memory leaks. After that we will grab one of the fuzzer programs(example: src/hfuzz_ip_output_fuzz.c) and chroot into the newly built distribution from the host NetBSD OS. Then we will compile it using hfuzz-clang by linking the required rumpkernel libraries (for example in our case: rump, rumpnet, rumpnet_netinet and so on). This is where we use the rumpkernel as libraries linked to our program. The program will access the network stack of the linked rumpkernel and the fuzzer will fuzz those components of the kernel. The compiled binary will be run with honggfuzz. Detailed steps are outlined in the Readme of the linked repository.

Our Approach for network stack fuzzing:

We have planned to fuzz various protocols at different layers of the TCP/IP stack. We have started with mostly widely used yet simple protocols like IP(v4), UDP etc. Along the progress of the project, we will be adding support for more L3(and above) protocols like ICMP, IP(v6), TCP as well as L2 protocols like Ethernet as a bit later phase.

The network stack has 2 paths:

1. Input/ingress path
2. Output/egress path

A packet is sent down the network stack via a socket from an application from the output path, whereas a packet is said to be received on a network interface into the network stack via the input path. Each network protocol has major input and output APIs for the packet processing. Example IP protocol has an ip_input() function to process an incoming packet and an ip_output() function to process an outgoing packet. We are planning to fuzz each protocol's output and input APIs by sending the packet via the output path and receiving the packet via input path respectively.

In order to fuzz the output and input path, the network stack setup and configuration we have is as follows:

- We have a TUN device to which we can read and write a packet.
- We have a socket that is bound to the TUN device, which can send and receive packets
- In order to fuzz the input path, we "write" a packet to the TUN interface, which emulates a received packet on the network stack.
- In order to fuzz the output path, we send a packet via the socket to the TUN interface to fuzz the output path.

For carrying out all the above setup, we have separated out the common function for creating and configuring the TUN device and socket into a file called "net_config.c"

Also in order to reduce the rejection of packets carrying random data for trivial cases like checksum or header len, we have created functions that create/forge semi-random packets using the input data from honggfuzz. We manipulate certain fields in the packet to ensure that it does not get rejected trivially by the network stack and hence can reach and trigger deeper parts of the code. These functions for packet creations are located in the "pkt_create.c" file. For each protocol we fuzz, we add these functions to forge the headers and the packet. Currently we have support from UDP and IP(v4).

With these building blocks we have written programs like hfuzz_ip_output_fuzz.c, hfuzz_ip_input_fuzz.c etc, which setup the TUN device and socket using net_config.c and after taking the random data from honggfuzz, use it to forge a packet and send it down or up the stack. We compile these programs using hfuzz-clang as mentioned above and run it under honggfuzz to fuzz the network stack's particular APIs.

Current Progress:

Following things were worked upon in the first phase:

- Getting honggfuzz functional for NetBSD(thanks to Kamil for those patches)
- Coming up with the strategy for network configuration and packet creation. Writing utilities for the same.

- Adding fuzzing code for protocols like IP(v4) and UDP.
- Carrying out fuzzing for those protocols.

Next Steps:

As next steps following things are planned for upcoming phase:

- Making changes and improvements by taking suggestions from the mentors.
- Adding support for ICMP, IP(v6), TCP and later on for Ethernet.
- Analyze and come up with effective ways to improve the fuzzing by focusing on the packet creation part.
- Standardize the code to be extensible for adding future protocols.