CS412 Fuzzing Lab Report

Ahmed Abdelmalek (344471), Benjamin Bürki (311199),
André Peiry (342289), Charles Rousset (300785)

Abstract

This report documents the CS412 fuzzing lab project, in which we targeted the open-source `libpcap` framework (`https://github.com/the-tcpdump-group/libpcap`), a portable, system-level library for capturing network packets that underpins critical tools such as `tcpdump` and Wireshark—underscoring the importance of ensuring its robustness and security through extensive fuzz testing.

1 Part1

Fuzzing With Seeds

Fuzzer Setup and Build

The harness `fuzz_pcap`, located at `libpcap/testprogs/fuzz/fuzz_pcap.c` [1], was built and run using the standard OSS-Fuzz helper scripts:

```
git clone https://github.com/google/oss-fuzz.git
cd oss-fuzz

python3 infra/helper.py build_image libpcap
python3 infra/helper.py build_fuzzers libpcap
```

Running the Fuzzer With Seeds

Typically, OSS-Fuzz handles seed extraction automatically. In our case the archive was manually unpacked inside the container before invoking the fuzzer.

```
python3 infra/helper.py shell libpcap

cd /out
unzip fuzz_pcap_seed_corpus.zip -d /tmp/corpus_pcap

/out/fuzz_pcap /tmp/corpus_pcap -max_total_time=14400
```

Corpus Summary

Over a four-hour run:

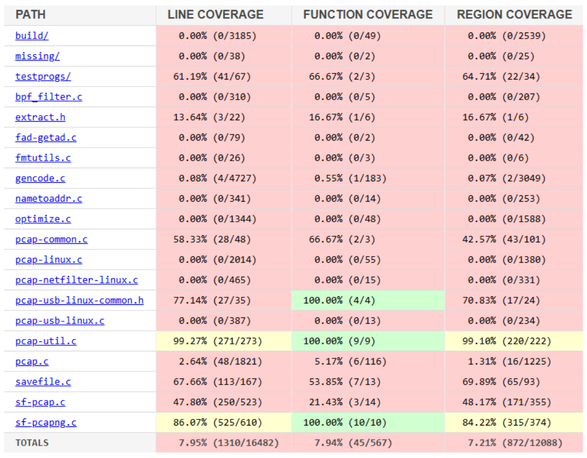
- Total inputs tested: **88,870,018**
- Coverage points hit: **361**
- Final corpus size: **864 inputs**, **2.3MB total**
- Execution speed: **6171 exec/s**

Coverage Report

After retrieving the corpus, the following steps were performed to generate the coverage report:

```
python3 infra/helper.py build_fuzzers --sanitizer coverage libpcap

python3 infra/helper.py coverage libpcap \
  --corpus-dir build/out/corpus_pcap \
  --fuzz-target fuzz_pcap
```



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
build/	0.00% (0/3185)	0.00% (0/49)	0.00% (0/2539)
missing/	0.00% (0/38)	0.00% (0/2)	0.00% (0/25)
testprogs/	61.19% (41/67)	66.67% (2/3)	64.71% (22/34)
bpf_filter.c	0.00% (0/310)	0.00% (0/5)	0.00% (0/207)
extract.h	13.64% (3/22)	16.67% (1/6)	16.67% (1/6)
fad-getad.c	0.00% (0/79)	0.00% (0/2)	0.00% (0/42)
fmtutils.c	0.00% (0/26)	0.00% (0/3)	0.00% (0/6)
gencode.c	0.08% (4/4727)	0.55% (1/183)	0.07% (2/3049)
nametoaddr.c	0.00% (0/341)	0.00% (0/14)	0.00% (0/253)
optimize.c	0.00% (0/1344)	0.00% (0/48)	0.00% (0/1588)
pcap-common.c	58.33% (28/48)	66.67% (2/3)	42.57% (43/101)
pcap-linux.c	0.00% (0/2014)	0.00% (0/55)	0.00% (0/1380)
pcap-netfilter-linux.c	0.00% (0/465)	0.00% (0/15)	0.00% (0/331)
pcap-usb-linux-common.h	77.14% (27/35)	100.00% (4/4)	70.83% (17/24)
pcap-usb-linux.c	0.00% (0/387)	0.00% (0/13)	0.00% (0/234)
pcap-util.c	99.27% (271/273)	100.00% (9/9)	99.10% (220/222)
pcap.c	2.64% (48/1821)	5.17% (6/116)	1.31% (16/1225)
savefile.c	67.66% (113/167)	53.85% (7/13)	69.89% (65/93)
sf-pcap.c	47.80% (250/523)	21.43% (3/14)	48.17% (171/355)
sf-pcapng.c	86.07% (525/610)	100.00% (10/10)	84.22% (315/374)
TOTALS	7.95% (1310/16482)	7.94% (45/567)	7.21% (872/12088)

Figure 1: HTML report overview showing total line coverage: **7.95%**

The full HTML coverage report is available in the `part1/report/w_corpus` folder.

Observations

Although overall line coverage reached was around 8%, this is typical in OSS-Fuzz scenarios with limited runtime. Coverage was focused on core `PCAP` and `PCAP Next Generation (PCAPNG)` parsers (e.g., `savefile.c`, `pcap-common.c`, `sf-pcapng.c`). Platform-specific code paths and deeper parsing logic, however, remained untouched, highlighting the need for a more diverse seed corpus and extended testing period.

Fuzzing Without Seeds

To assess the impact of removing the initial seed corpus, the corpus-packaging steps in the build script were disabled, the fuzzer was rebuilt, and then run against an empty directory. The commands below illustrate how `fuzz_pcap` was invoked with no seeds:

```
git checkout -- projects/libpcap/build.sh
git apply ../oss-fuzz.diff
python3 infra/helper.py build_fuzzers libpcap
rm -rf build/out/libpcap/fuzz_pcap_seed_corpus.zip
mkdir -p build/out/corpusnoseeds
python3 infra/helper.py run_fuzzer libpcap fuzz_pcap --corpus-dir build/out/corpusnoseeds
```

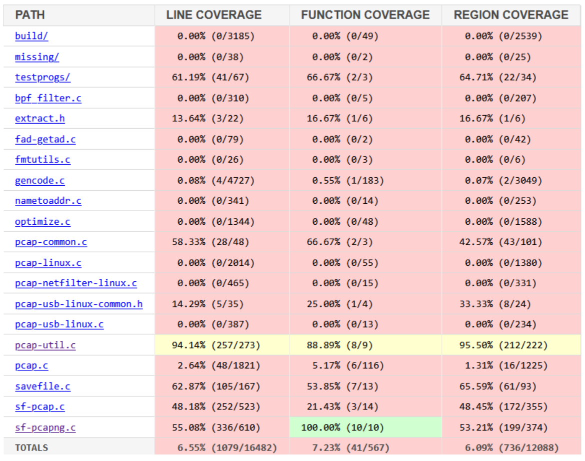
> **NOTE:** To ensure that the fuzzing campaign runs strictly without any predefined input corpus, we explicitly removed the default seed corpus archive using the command:
>
> ```
> rm -rf build/out/libpcap/fuzz_pcap_seed_corpus.zip
> ```

> While OSS-Fuzz's build system is supposed to respect changes in the `build.sh` script that exclude the seed corpus, we observed that in some cases—possibly due to build image inconsistencies or platform-specific differences (e.g., ARM builds versus x86_64)—the fuzzer still included the default corpus unintentionally. Manually deleting the archive is therefore a necessary precaution to guarantee a truly corpus-free run for comparative coverage analysis when reusing the same `OSS-Fuzz` directory across different fuzzing runs.

Patch Diff

The following diff shows exactly which lines in `projects/libpcap/build.sh` were commented out to prevent the seed corpus from being packaged and installed into the fuzzing image. This is the change that makes the "no-seeds" run possible:

```
diff --git a/projects/libpcap/build.sh b/projects/libpcap/build.sh
index 75463b343..122625987 100755
--- a/projects/libpcap/build.sh
+++ b/projects/libpcap/build.sh
@@ -32,13 +32,13 @@ done
 # export other associated stuff
 cd ..
-cp testprogs/fuzz/fuzz_*.options $OUT/
+#cp testprogs/fuzz/fuzz_*.options $OUT/
 # builds corpus
 cd $SRC/tcpdump/
-zip -r fuzz_pcap_seed_corpus.zip tests/
-cp fuzz_pcap_seed_corpus.zip $OUT/
+#zip -r fuzz_pcap_seed_corpus.zip tests/
+#cp fuzz_pcap_seed_corpus.zip $OUT/
 cd $SRC/libpcap/testprogs/BPF
 mkdir corpus
 ls *.txt | while read i; do tail -1 $i > corpus/$i; done
-zip -r fuzz_filter_seed_corpus.zip corpus/
-cp fuzz_filter_seed_corpus.zip $OUT/
+#zip -r fuzz_filter_seed_corpus.zip corpus/
+#cp fuzz_filter_seed_corpus.zip $OUT
```



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
build/	0.00% (0/3185)	0.00% (0/49)	0.00% (0/2539)
missing/	0.00% (0/38)	0.00% (0/2)	0.00% (0/25)
testprogs/	61.19% (41/67)	66.67% (2/3)	64.71% (22/34)
bpf_filter.c	0.00% (0/310)	0.00% (0/5)	0.00% (0/207)
extract.h	13.64% (3/22)	16.67% (1/6)	16.67% (1/6)
fad-getad.c	0.00% (0/79)	0.00% (0/2)	0.00% (0/42)
fmtutils.c	0.00% (0/26)	0.00% (0/3)	0.00% (0/6)
gencode.c	0.08% (4/4727)	0.55% (1/183)	0.07% (2/3049)
nametoaddr.c	0.00% (0/341)	0.00% (0/14)	0.00% (0/253)
optimize.c	0.00% (0/1344)	0.00% (0/48)	0.00% (0/1588)
pcap-common.c	58.33% (28/48)	66.67% (2/3)	42.57% (43/101)
pcap-linux.c	0.00% (0/2014)	0.00% (0/55)	0.00% (0/1380)
pcap-netfilter-linux.c	0.00% (0/465)	0.00% (0/15)	0.00% (0/331)
pcap-usb-linux-common.h	14.29% (5/35)	25.00% (1/4)	33.33% (8/24)
pcap-usb-linux.c	0.00% (0/387)	0.00% (0/13)	0.00% (0/234)
pcap-util.c	94.14% (257/273)	88.89% (8/9)	95.50% (212/222)
pcap.c	2.64% (48/1821)	5.17% (6/116)	1.31% (16/1225)
savefile.c	62.87% (105/167)	53.85% (7/13)	65.59% (61/93)
sf-pcap.c	48.18% (252/523)	21.43% (3/14)	48.45% (172/355)
sf-pcapng.c	55.08% (336/610)	100.00% (10/10)	53.21% (199/374)
TOTALS	6.55% (1079/16482)	7.23% (41/567)	6.09% (736/12088)

Figure 2: HTML report overview showing total line coverage: **6.55%**

The full HTML coverage report is available at `part1/report/w_o_corpus/linux/report.html`.

Analysis

To evaluate the impact of having a seed corpus on code coverage, we conducted a comparative study of two source files in the `libpcap` project: `sf-pcapng.c` and `pcap-util.c`. For each file, we analyzed the HTML coverage reports generated under two distinct conditions: (1) fuzzing with a set of seed inputs and (2) fuzzing without any initial corpus. Our objective was to understand which specific parts of the code benefit from seeding, and why certain execution paths remain uncovered in the absence of input guidance.

sf-pcapng.c The source file `sf-pcapng.c` contains logic for parsing and validating `pcapng` file blocks. This format introduces structured metadata and multiple block types (e.g., Section Header Blocks, Interface Description Blocks). Coverage analysis reveals that the inclusion of seed inputs significantly influences which parts of the code are exercised.

When fuzzing without seeds, only a narrow subset of the parsing logic is executed. Specifically, the coverage is largely limited to initial block identification and format detection routines. Many conditionals and error paths are skipped, notably those dealing with malformed input, version mismatches, or incorrect block trailer lengths. This observation is expected: without a seed corpus, the fuzzer generates largely invalid binary input that often fails to pass even the first few sanity checks in the parsing pipeline.

By contrast, when seed inputs are supplied—particularly minimal valid pcapng files—the fuzzer is capable of progressing deeper into the parsing stages. For instance, functions that interpret options, timestamps, and per-interface metadata are triggered only when input resembles a structurally valid pcapng stream. As a result, blocks such as `process_idb_options`, `add_interface`, and timestamp scaling logic (`scale_type`, `scale_factor`) show marked increases in line coverage. Additionally, code paths that test for optional interface fields such as `IF_TSRESOL` or `IF_TSOFFSET` are not covered at all in the unseeded run, further illustrating the limitations of purely random input generation in reaching structured conditional logic.

This disparity indicates that the parser in `sf-pcapng.c` depends heavily on syntactic and semantic constraints that are non-trivial to satisfy without structured guidance. The presence of well-formed seed inputs thus serves as a critical enabler for coverage depth and exploration of edge-case branches.

pcap-util.c The source file `pcap-util.c` provides utility routines that support file manipulation and encoding/decoding operations, including endian swapping and string formatting for packet metadata. Unlike the deeply nested and structure-sensitive logic of `sf-pcapng.c`, the utilities in this file are comparatively shallow and compositional.

Interestingly, even without seed inputs, a relatively large portion of `pcap-util.c` is exercised during fuzzing. This can be attributed to the fact that the utility functions are often called at early stages of parsing or error formatting, and therefore do not require syntactically valid input to be triggered. For example, functions such as `pcap_fmt_errmsg_for_errno` or byte-order manipulation macros are invoked even when the input causes early failure in the main parsing loop.

However, we observe that certain utility functions—particularly those related to timestamp formatting or numerical resolution decoding—remain uncovered in the unseeded setup. These functions rely on downstream parsing stages reaching specific block types (e.g., IDB, EPB) or fields (e.g., `if_tsresol`, `option_length`) that are structurally absent in random input. With a seed corpus, these paths are successfully triggered, resulting in improved overall coverage and better exercise of the numeric formatting logic.

Thus, while `pcap-util.c` exhibits greater resilience to input randomness than `sf-pcapng.c`, it still benefits from seeded execution, especially in exploring edge logic and fully parameterized code paths.

Discussion The coverage gap between seeded and unseeded fuzzing runs highlights a fundamental limitation of structure-agnostic input generation. Without syntactic and semantic scaffolding, fuzzers fail to trigger higher-level abstractions, particularly in file format parsers and data interpreters. Seed inputs act as "scaffolds" that not only guide the fuzzer past initial validity checks but also open up opportunities for exercising conditional logic and error handling pathways that would otherwise remain dormant.

Both the coverage profile of `sf-pcapng.c` and the relatively more robust coverage of `pcap-util.c` underscore the importance of combining fuzzing with domain-specific input seeding. While utility code can often be exercised without strict input conformance, format parsers require initial guidance to expose the full space of possible execution paths.

2 Part2

2.1 Uncovered Region 1: `pcap_activate_linux`

A major uncovered region in our fuzzing campaign is the function `pcap_activate_linux`, defined in the Linux-specific file `pcap-linux.c`. This function is responsible for activating live capture on a specified network interface and performs extensive system-level setup, including socket creation, interface validation, ioctl configuration, eventfd allocation, and memory-mapped ring buffer initialization.

Despite its central role in initializing packet capture on Linux systems, `pcap_activate_linux` had **0% function, line, and region coverage** in our fuzzing report.

Significance

- It is the primary entry point for enabling live packet capture on Linux and is used by widely deployed tools like `tcpdump`, `Wireshark`, and other network monitoring frameworks.
- The function interacts with raw sockets, system devices (e.g., `/sys/class/net/`), and kernel ring buffers, which makes it highly sensitive to edge cases, misconfigurations, and platform-specific behavior.
- Vulnerabilities in this logic, such as unchecked return values or misused syscalls, could lead to denial-of-service, information disclosure, or incorrect capture setup. In fact, a real-world double-free issue was previously identified and patched in this function's error-handling paths[2].

Why It Was Not Covered

- The default fuzzing harness, `fuzz_pcap`, targets parsing of `.pcap` files and does not invoke live interface activation.
- `pcap_activate_linux` requires privileged system access to create and configure raw sockets, bind interfaces, and set low-level capture modes, which represent operations that are blocked in OSS-Fuzz's sandboxed container environment.
- The function expects a valid and fully initialized `pcap_t` structure and a real network interface name, making it difficult to reach through standard fuzzing inputs.

Takeaway The complete absence of coverage in such a fundamental and security-relevant function illustrates a major limitation of generic fuzzing setups: system-level logic often requires tailored harnesses, mock environments, or elevated permissions to be meaningfully tested.

2.2 Uncovered Region 2: pcap_findalldevs_ex

An interesting uncovered region that is part of the native API is the functionpcap_findalldevs_ex (Listing 1), which is the entry-point API into libpcap's device-discovery functionality. This function handles several roles—scanning a directory for capture files, listing local network interfaces, or enumerating devices on a remote RPCAP-compatible server. An Remote Capture () server is simply a `rpcapd` daemon running on a remote host; it implements libpcap's RPCAP protocol, allowing clients to authenticate, list available interfaces, and capture packets over the network.

```c
int pcap_findalldevs_ex(const char *source,
    struct pcap_rmtauth *auth _USED_FOR_REMOTE,
    pcap_if_t **alldevs,
    char *errbuf);
```

Listing 1: Signature of `pcap_findalldevs_ex`

If `source` begins with `file://`, the remainder is treated as a filesystem path. If it names a directory, libpcap scans that directory (non-recursively) for files with known savefile extensions (e.g. `.pcap`, `.pcapng`) and returns each as a separate "capture file" device.

If `source` begins with `rpcap://[username:password@]host[:port]/devicename` (where the parts inside the square brackets are optional), libpcap uses the `auth` struct (which carries the authentication type, username, and password) to connect to the remote `rpcapd` daemon and then enumerates its interfaces; if instead `source` is just `rpcap://devicename`, it bypasses RPCAP entirely and opens `devicename` on localhost. If no recognized prefix appears, `source` is assumed to be the name of a local network interface and attempts to match it by enumerating the available interfaces.

On success, `alldevs` is set to a linked list of `pcap_if_t` records (each with fields for `name`, `description`, addresses, and flags). On failure (unsupported mode, unreachable host, nonexistent path), the function returns −1 and writes a human-readable error into `errbuf`.

This function is directly used by Wireshark: it defines the wrapper `ws_pcap_findalldevs_ex`, which is called in the snippet below to enable remote packet capture [3]:

```c
#ifdef HAVE_PCAP_REMOTE
GList *
get\_interface\_list\_findalldevs_ex(const char *hostname, const char *port,
              int auth_type, const char *username,
              const char *passwd, int *err, char **err_str)
{
// ...existing code...
    auth.type = auth_type;
    auth.username = g_strdup(username);
    auth.password = g_strdup(passwd);

    if (ws_pcap_findalldevs_ex(source, &auth, &alldevs, errbuf) == -1) {
        *err = CANT_GET_INTERFACE_LIST;
        if (strcmp(errbuf, "not supported") == 0) {
            g_strlcpy(errbuf, "Remote capture not supported",
                    PCAP_ERRBUF_SIZE);
        }
    }
```

Listing 2: Call to `ws_pcap_findalldevs_ex` within `get_interface_list_findalldevs_ex` in `capture-pcap-util.c`

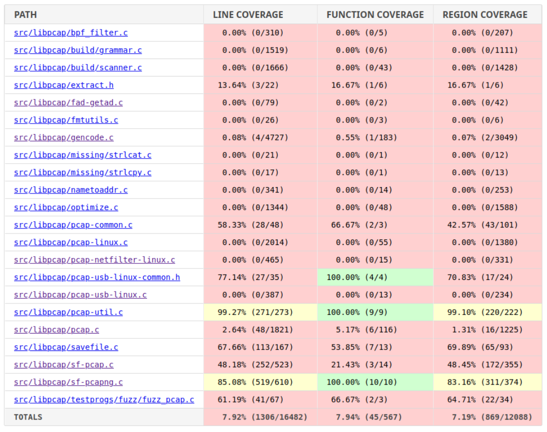
Both Wireshark and `tcpdump` (maintained by the same team as `libpcap`) directly rely on `pcap_findalldevs_ex` for local and remote enumeration, underscoring its importance in thoroughly testing it.

However, OSS-Fuzz's existing libpcap harness never calls the special source-string logic in `pcap_findalldevs_ex`. It fuzzes savefile parsing (by feeding fuzzed buffers into `pcap_open_offline`) and BPF compilation, but never calls `pcap_findalldevs_ex` with a `file://` or `rpcap://...` source string. As a result, all of that code—scanning a folder for `.pcap` files, enumerating local interfaces via `rpcap://`, RPCAP URL parsing, performing the RPCAP handshake, and remote device listing—remains unreachable under the current fuzzing setup. Moreover, even if the current harnesses were to directly supply fuzzed input as the source string to `pcap_findalldevs_ex`, the existing seed corpus critically lacks valid seeds formatted with file:// or rpcap:// prefixes. Without such targeted inputs, it is unlikely that the fuzzer will sufficiently probe the function's logic for handling directory scanning or remote interface enumeration.

Recent CVEs highlight why this region must be fuzzed. CVE-2024-8006 reports that, when remote support is enabled and a nonexistent path is given, a NULL-pointer dereference occurs in `pcap_findalldevs_ex` [4]. CVE-2019-15164 describes a server-side request forgery in the RPCAP daemon's URL parser when it treats a malicious capture-source string as a request [5]. These bugs show both the important of thoroughly testing this code and the complexity involed —effective fuzzing requires simulating directory layouts, platform-specific interface names, and even a mock RPCAP server to cover every branch.

At roughly 150 LOC (in just `findalldevs_ex` and not all the functions called), this single entry point embodies the highest untested complexity in libpcap and contains enough distinct logic to justify its own fuzzing harness [6].

3 Part3



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
src/libpcap/bpf_filter.c	0.00% (0/310)	0.00% (0/5)	0.00% (0/207)
src/libpcap/build/grammar.c	0.00% (0/1519)	0.00% (0/6)	0.00% (0/1111)
src/libpcap/build/scanner.c	0.00% (0/1666)	0.00% (0/43)	0.00% (0/1428)
src/libpcap/extract.h	13.64% (3/22)	16.67% (1/6)	16.67% (1/6)
src/libpcap/fad-getad.c	0.00% (0/79)	0.00% (0/2)	0.00% (0/42)
src/libpcap/fmtutils.c	0.00% (0/26)	0.00% (0/3)	0.00% (0/6)
src/libpcap/gencode.c	0.08% (4/4727)	0.55% (1/183)	0.07% (2/3049)
src/libpcap/missing/strlcat.c	0.00% (0/21)	0.00% (0/1)	0.00% (0/12)
src/libpcap/missing/strlcpy.c	0.00% (0/17)	0.00% (0/1)	0.00% (0/13)
src/libpcap/nametoaddr.c	0.00% (0/341)	0.00% (0/14)	0.00% (0/253)
src/libpcap/optimize.c	0.00% (0/1344)	0.00% (0/48)	0.00% (0/1588)
src/libpcap/pcap-common.c	58.33% (28/48)	66.67% (2/3)	42.57% (43/101)
src/libpcap/pcap-linux.c	0.00% (0/2014)	0.00% (0/55)	0.00% (0/1380)
src/libpcap/pcap-netfilter-linux.c	0.00% (0/465)	0.00% (0/15)	0.00% (0/331)
src/libpcap/pcap-usb-linux-common.h	77.14% (27/35)	100.00% (4/4)	70.83% (17/24)
src/libpcap/pcap-usb-linux.c	0.00% (0/387)	0.00% (0/13)	0.00% (0/234)
src/libpcap/pcap-util.c	99.27% (271/273)	100.00% (9/9)	99.10% (220/222)
src/libpcap/pcap.c	2.64% (48/1821)	5.17% (6/116)	1.31% (16/1225)
src/libpcap/savefile.c	67.66% (113/167)	53.85% (7/13)	69.89% (65/93)
src/libpcap/sf-pcap.c	48.18% (252/523)	21.43% (3/14)	48.45% (172/355)
src/libpcap/sf-pcapng.c	85.08% (519/610)	100.00% (10/10)	83.16% (311/374)
src/libpcap/testprogs/fuzz/fuzz_pcap.c	61.19% (41/67)	66.67% (2/3)	64.71% (22/34)
TOTALS	7.92% (1306/16482)	7.94% (45/567)	7.19% (869/12088)

Figure 3: Baseline coverage report for the pcap harness after three 4-hour fuzzing runs.

In Figure 3, the weaknesses of this harness are clearly shown. It primarily targets the parsing of `.pcap` and `.pcapng` packets, leaving most of the remaining code untouched.

The full HTML coverage report is available at the following path in the project repository: `part3/coverage_noimprove/coverage_improve2/report.html`.

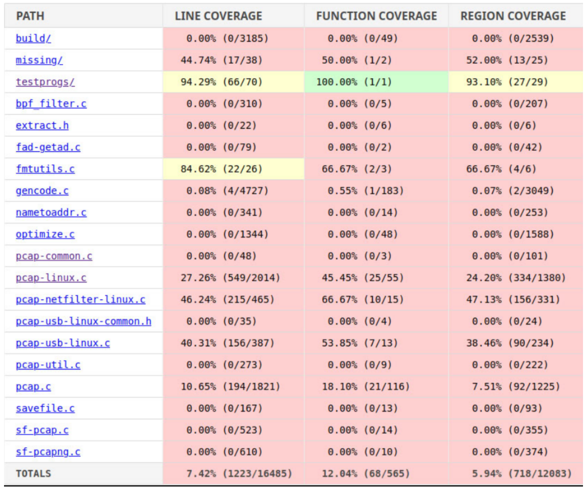
3.1 Uncovered Region 1 Improvement : pcap_activate

Challenge As previously mentioned, testing platform-dependent functionalities, such as `pcap_activate _linux`, relies on live USB and network interfaces. These functions interact directly with system-level resources like raw sockets, hardware interfaces, and kernel buffers. Hardware-dependent interactions pose a significant barrier to fuzzing, since many initialization routines cannot be executed without a real interface, resulting in substantial coverage gaps.

Harness Design To bypass hardware and privilege constraints, the harness constructs and configures a `pcap_t` handle entirely from fuzz input, without relying on real network devices. It begins by reading the first byte to determine the length of a synthetic interface name, copies the next bytes into a null-terminated string, and calls `pcap_create(interface_name, errbuf)`. It then consumes successive bytes to set capture parameters—snaplen, promiscuous mode, timeout, buffer size, immediate mode, and packet direction—via the corresponding `pcap_set_*` APIs. Finally, it invokes `pcap_activate`, which exercises validation, setup, and error-handling logic. Since no real interfaces are available beyond loopback, the activation either fails gracefully or traverses fallback paths, and on success the harness calls `pcap_stats`, `pcap_snapshot`, and `pcap_datalink` before closing the handle. This design maximizes coverage of Linux-specific initialization code while avoiding direct hardware interaction.

Due to the completely new fuzzing target—interface activation rather than packet parsing—the original PCAP seed corpus was unusable. Instead, we built a fresh seed set made up entirely of plausible interface names. All seeds were crafted solely to supply a valid interface name as the first input, ignoring all subsequent configuration bytes and relying on the fuzzer to generate those on its own.

Results With this new harness, we can see a jump in function coverage from 7.23% to 12.04%. Not the whole `pcap-linux.c` was fuzzed, as we mainly focused on the `pcap` generation, and less about other function that are oriented towards socket or network information.



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
build/	0.00% (0/3185)	0.00% (0/49)	0.00% (0/2539)
missing/	44.74% (17/38)	50.00% (1/2)	52.00% (13/25)
testprogs/	94.29% (66/70)	100.00% (1/1)	93.10% (27/29)
bpf_filter.c	0.00% (0/310)	0.00% (0/5)	0.00% (0/207)
extract.h	0.00% (0/22)	0.00% (0/6)	0.00% (0/6)
fad-getad.c	0.00% (0/79)	0.00% (0/2)	0.00% (0/42)
fmtutils.c	84.62% (22/26)	66.67% (2/3)	66.67% (4/6)
gencode.c	0.08% (4/4727)	0.55% (1/183)	0.07% (2/3049)
nametoaddr.c	0.00% (0/341)	0.00% (0/14)	0.00% (0/253)
optimize.c	0.00% (0/1344)	0.00% (0/48)	0.00% (0/1588)
pcap-common.c	0.00% (0/48)	0.00% (0/3)	0.00% (0/101)
pcap-linux.c	27.26% (549/2014)	45.45% (25/55)	24.20% (334/1380)
pcap-netfilter-linux.c	46.24% (215/465)	66.67% (10/15)	47.13% (156/331)
pcap-usb-linux-common.h	0.00% (0/35)	0.00% (0/4)	0.00% (0/24)
pcap-usb-linux.c	40.31% (156/387)	53.85% (7/13)	38.46% (90/234)
pcap-util.c	0.00% (0/273)	0.00% (0/9)	0.00% (0/222)
pcap.c	10.65% (194/1821)	18.10% (21/116)	7.51% (92/1225)
savefile.c	0.00% (0/167)	0.00% (0/13)	0.00% (0/93)
sf-pcap.c	0.00% (0/523)	0.00% (0/14)	0.00% (0/355)
sf-pcapng.c	0.00% (0/610)	0.00% (0/10)	0.00% (0/374)
TOTALS	7.42% (1223/16485)	12.04% (68/565)	5.94% (718/12083)

Figure 4: Coverage report for the `fuzz_linuxactivate` harness targeting Region 1 after three independent 4-hour fuzzing runs.

Future Work A few different strategies could be employed to improve coverage of this region. The first and most costly would be to implement mock interfaces that behave identically to real hardware, complete with raw socket semantics. With such mocks in place, the fuzzing harness could drive `pcap_activate_linux` through every initialization path and error condition.

A more realistic approach would be to refine the seed corpus so that each entry includes not only a valid interface name but also the exact initialization bytes consumed by the harness (snaplen, promiscuous flag, timeout, buffer size, immediate mode, direction, etc.). The current corpus omits these "magic" values, forcing the fuzzer to stumble across them by chance—which makes it extremely unlikely to drive many of the deeper initialization paths during short fuzzing campaigns.

Coverage report The HTML coverage report is available at the following path in the project repository: `part3/improve1/coverage_improve1/report`.

3.2 Uncovered Region 2 Improvement : pcap_findalldevs_ex

Challenge The existing OSS-Fuzz harnesses for libpcap (the "libpcap", "filter" and "both" targets) never invoke the device-enumeration code in `pcap_findalldevs_ex`, because that code normally iterates over real network interfaces or connects to an RPCAP server—operations that are impractical in a sandboxed fuzzing environment.

Harness Design To work around this, a new harness was written that restricts itself to two core behaviors within that API: filesystem directory scanning for valid `.pcap` and `.pcang` files and RPCAP URL parsing. The first data input byte into the harness selects one of three modes (Listing 3). If `mode_selector == 0` the harness constructs a string prefixed with `rpcap://` from the fuzz data and calls `pcap_findalldevs_ex`. Because the library is built without `ENABLE_REMOTE` (by default), this exercises only the URL-parsing logic (Listing 4) and always returns an error, without trying to connect to a remote server.

```c
int mode_selector = Data[0] % 3;
```

Listing 3: Choosing between rpcap and file

```c
case PCAP_SRC_IFREMOTE:
#ifdef ENABLE_REMOTE
   return (pcap_findalldevs_ex_remote(source, auth, alldevs, errbuf));
#else
   pcapint_strlcpy(errbuf, "Remote packet capture is not supported",
                PCAP_ERRBUF_SIZE);
   return (PCAP_ERROR);
#endif
```

Listing 4: RPCAP URL parsing stub in `pcap_findalldevs_ex`

If `mode_selector == 1`, the harness enters a `file://`-based mode: it creates a temporary directory, consumes fuzz bytes to write 0–6 synthetic `.pcap` files, then uses a few more byte to produce a NULL terminated filename suffix. It builds the URL in the format:

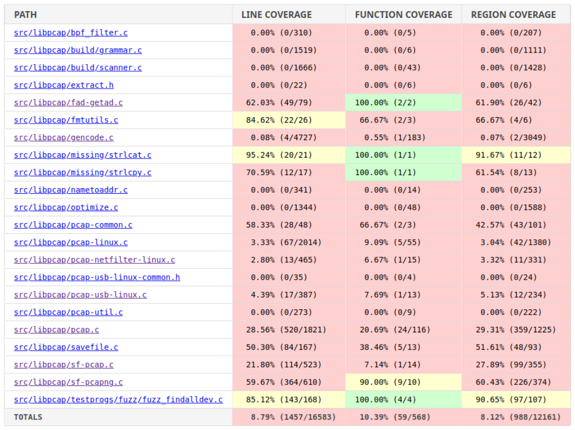
```
file://<temporary-directory>/<fuzzed-name>
```

and invokes `pcap_findalldevs_ex`, calling the filesystem-scanning and parsing logic. When it returns, the harness cleans up all files and the directory.

Finally, if `mode_selector == 2`, the harness appends the input data to `file://`, simulating malformed or non-existent paths. This reproduces the boundary conditions of CVE-2024-8006—calling the code with a non-existent directory—so that the function's error-handling and buffer routines are exercised without relying on real files or interfaces [4].

The original seed corpus from the `pcap` harness was augmented with well-formed examples drawn from the official documentation to improve the convergence toward realistic inputs. Specifically, five representative `file://...` entries and fifteen `rpcap://...` entries were added.

The `run.improve2.sh` script in `part3/improve2/` sets up OSS-Fuzz with the new harness and seed corpus under `oss-fuzz/projects/libpcap`, builds the Docker image and fuzzers, runs `fuzz_findalldev` for three four-hour sessions, and outputs an HTML coverage report to `part3/improve2/coverage_improve2/`.



PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
src/libpcap/bpf_filter.c	0.00% (0/310)	0.00% (0/5)	0.00% (0/207)
src/libpcap/build/grammar.c	0.00% (0/1519)	0.00% (0/6)	0.00% (0/1111)
src/libpcap/build/scanner.c	0.00% (0/1666)	0.00% (0/43)	0.00% (0/1428)
src/libpcap/extract.h	0.00% (0/22)	0.00% (0/6)	0.00% (0/6)
src/libpcap/fad-getad.c	62.03% (49/79)	100.00% (2/2)	61.90% (26/42)
src/libpcap/fmtutils.c	84.62% (22/26)	66.67% (2/3)	66.67% (4/6)
src/libpcap/gencode.c	0.08% (4/4727)	0.55% (1/183)	0.07% (2/3049)
src/libpcap/missing/strlcat.c	95.24% (20/21)	100.00% (1/1)	91.67% (11/12)
src/libpcap/missing/strlcpy.c	70.59% (12/17)	100.00% (1/1)	61.54% (8/13)
src/libpcap/nametoaddr.c	0.00% (0/341)	0.00% (0/14)	0.00% (0/253)
src/libpcap/optimize.c	0.00% (0/1344)	0.00% (0/48)	0.00% (0/1588)
src/libpcap/pcap-common.c	58.33% (28/48)	66.67% (2/3)	42.57% (43/101)
src/libpcap/pcap-linux.c	3.33% (67/2014)	9.09% (5/55)	3.04% (42/1380)
src/libpcap/pcap-netfilter-linux.c	2.80% (13/465)	6.67% (1/15)	3.32% (11/331)
src/libpcap/pcap-usb-linux-common.h	0.00% (0/35)	0.00% (0/4)	0.00% (0/24)
src/libpcap/pcap-usb-linux.c	4.39% (17/387)	7.69% (1/13)	5.13% (12/234)
src/libpcap/pcap-util.c	0.00% (0/273)	0.00% (0/9)	0.00% (0/222)
src/libpcap/pcap.c	28.56% (520/1821)	20.69% (24/116)	29.31% (359/1225)
src/libpcap/savefile.c	50.30% (84/167)	38.46% (5/13)	51.61% (48/93)
src/libpcap/sf-pcap.c	21.80% (114/523)	7.14% (1/14)	27.89% (99/355)
src/libpcap/sf-pcapng.c	59.67% (364/610)	90.00% (9/10)	60.43% (226/374)
src/libpcap/testprogs/fuzz/fuzz_findalldev.c	85.12% (143/168)	100.00% (4/4)	90.65% (97/107)
TOTALS	8.79% (1457/16583)	10.39% (59/568)	8.12% (988/12161)

Figure 5: Coverage report for the `fuzz_findalldev` harness targeting Region 2 after three independent 4-hour fuzzing runs.

Results Comparing the baseline coverage (Figure 3) with the results from our new harness (Figure 5) shows dramatic gains in the core library. On `pcap.c` itself, line coverage jumps from 2.64 % to 21.80 %, and function coverage increases by roughly 15 %. This improvement stems from using `pcap_findalldevs_ex` as an alternative entry point, which unlocks paths through directory scanning, RPCAP URL parsing, and related APIs that the original `fuzz_pcap` harness never reached.

The new harness exercises all non-error-handling paths in `pcap_findalldevs_ex`, and covers the majority of `pcap_parsesrcstr`, the routine responsible for parsing the source string. Additionally, it invokes the offline file-opening logic (via `pcap_open_offline`), further expanding coverage into file-loading mechanisms.

Future Work That said, even this new harness leaves important areas under-exercised—most notably the library's error-handling paths. Libpcap consistently reports all errors via the caller-supplied buffer, yet there is no coverage of those branches. For future work, it would be useful to steer the fuzzer toward inputs that deliberately trigger parse-failures (e.g. malformed filenames, permission errors, source buffer too big) in order to validate and harden every error case.

Another promising extension to this harness is to open each discovered "capture file" with `pcap_open_offline`. This enables fuzzing of the packet-iteration loop and BPF filter logic over both real and synthetic PCAP files—potentially uncovering deeply nested bugs that require specific packet sequences or filter configurations.

Coverage report The HTML coverage report is available at the following path in the project repository: `part3/improve2/coverage_improve2/report`.

4 Part4

4.1 Bug Triage

Fuzzing the Region 2 harness by passing raw input as a file path to `pcap_findalldevs_ex` in libpcap 1.10 triggers a buffer underflow whose causes are investigated further.

Proof-of-Concept

The following minimal program reproduces the crash when the `source` string is exactly `"file://"` or contains a leading NUL after the `"file://"` in the path:

```c
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_if_t *alldevs = NULL;
    errbuf[0] = '\0';
    const char *source = "file://\0buffer";

    int result = pcap_findalldevs_ex(source, NULL, &alldevs, errbuf);

    if (result == PCAP_ERROR) {
        fprintf(stderr, "pcap_findalldevs_ex failed.\n");
    }
    if (alldevs != NULL) {
        pcap_freealldevs(alldevs);
    }
```

Listing 5: POC for buffer underflow in `pcap_findalldevs_ex`

4.2 Root Cause Analysis

Execution Walkthrough For this crash to occur, two conditions must be met. First, the `source` string must begin with the prefix `"file://"` so that `pcap_findalldevs_ex` enters its FILE branch and attempts parsing the source as a directory. Second, the path component must be empty; the source is either exactly `"file://"` or begins with a NUL (e.g. `"file://\0aAfdasdlkj123"`). This leads to the following execution path:

```c
int pcap_findalldevs_ex(const char *source, struct pcap_rmtauth *auth _USED_FOR_REMOTE,
    pcap_if_t **alldevs, char *errbuf)
{
    int type;
    // ... setup and sanity checks ...big
        if (pcap_parsesrcstr(source, &type, NULL, NULL, NULL, errbuf) == -1)
            return (PCAP_ERROR);
    switch (type) {
        // ...existing code...
    case PCAP_SRC_FILE:
        // ...existing code...
```

The call to `pcap_parsesrcstr` parses the `source` string to determine its type (file, local, or remote). In this case, because the prefix is `file://`, execution enters the `PCAP_SRC_FILE` branch. The first action there is to invoke `pcapint_parsesrcstr_ex` a second time, with the following arguments:

```c
    pcapint_parsesrcstr_ex(source, type, NULL, host, port, name, NULL, errbuf))
```

When `source` is `"file://\0buffer"`, the `pcap_parse_source` function parses the scheme ("file") and the path (the portion after "://"). Because the path begins with a NUL character, an empty string is copied into the `name` buffer. This is the main culprit that will cause a read in the wrong location.

```c
/* Check that the filename is correct */
    stringlen = strlen(name);
    if (name[stringlen - 1] != ENDING_CHAR) {
        name[stringlen] = ENDING_CHAR;
        name[stringlen + 1] = 0;
        stringlen++;
    }
```

For a valid directory path, the string must end with a path separator (e.g. '\\' on Windows or '/' on Unix). However, when `name` is empty , `strlen(name)` returns 0. Consequently, the test

```c
if (name[stringlen - 1] != ENDING_CHAR)
```

accesses `name[-1]`, reading before the start of the buffer and causing a memory underflow.

Proposed Fix A minimal change to fix this issue is to verify that the length is greater than zero before indexing. An overflow check is unnecessary, as the function already performs a length validation at its start.

```c
    stringlen = strlen(name);
    if (stringlen > 0 && name[stringlen - 1] != ENDING_CHAR) {
        name[stringlen] = ENDING_CHAR;
        name[stringlen + 1] = 0;
        stringlen++;
    }
```

4.3 Exploitability Analysis

The vulnerability in `pcap_findalldevs_ex` stems from improper handling of empty strings. Specifically, the condition this code 4.2 results in a read of `name[-1]` when `name` is empty, resulting in Undefined Behavior (UB).

If the code is compiled with sanitizers like AddressSanitizer or UBSan, this out-of-bounds read is immediately detected, causing a crash and yielding a reliable Denial-of-Service (DoS) primitive.

Without sanitizers, the underflow reads the byte immediately before the `name` buffer on the stack. If that byte is not `ENDING_CHAR` (typically `"/"` on non-Windows systems), the code writes `ENDING_CHAR` into `name[0]`, turning `name` into `"/"`. Libpcap then treats `"/"` as the capture source and invokes `opendir()/readdir()/stat()` on its entries to look for ".pcap" or ".pcapng" files. In contexts where libpcap runs with elevated privileges, the underflow can force a scan of "/" for ".pcap"/".pcapng" files. Any matching filenames will be returned in the device list—even if the unprivileged user could not normally enumerate them—thus revealing the presence and exact names of privileged or hidden capture files. More importantly—assuming the sanitizers do not abort the process—that stray read fetches one arbitrary stack byte and immediately determines the subsequent directory-opening logic. Below is the relevant snippet:

```c
snprintf(path, sizeof(path), "%s", name);
pathlen = strlen(path);
unixdir = opendir(path);
if (unixdir == NULL) {
    snprintf(errbuf, PCAP_ERRBUF_SIZE,
        "Error when listing files in '%s': %s", path, pcap_strerror(errno));
    return (PCAP_ERROR);
}
```

Listing 6: Directory-opening logic in `pcap_findalldevs_ex` (lines 4751–4780 of `pcap.c`)

If `name[-1] != ENDING_CHAR` 4.2, then `name` remains the empty string, so `path` is `""`; as a result, `opendir("")` fails and logs an error showing an empty path 6. Conversely, if `name[-1] == ENDING_CHAR`, then `name` becomes `"/"`, so `path` is `"/"`; in that case, `opendir("/")` succeeds and silently iterates over the root directory without error. By observing whether an error is reported for the empty path versus no error (and a root-scan), an attacker recovers the single stack byte value leaked by the underflow.

Thus, in hardened builds, the bug is a consistent DoS. In relaxed environments, it may trigger unintended behavior or enable weak forms of memory inspection, but with limited security impact.

References

[1] the-tcpdump-group, “libpcap: the libpcap interface to various kernel packet capture mechanism,” https://github.com/the-tcpdump-group/libpcap, 2025, [Online; accessed 14-May-2025].

[2] mkarhumaa, “Bluetooth: fix non-blocking mode,” GitHub Pull Request #1002, the-tcpdump-group/libpcap, Mar. 2021. [Online]. Available: https://github.com/the-tcpdump-group/libpcap/pull/1002

[3] Wireshark Foundation, “Wireshark: Official code repository,” https://gitlab.com/wireshark/wireshark, 2018, accessed: 2025-05-06.

[4] Tcpdump Group, “Null pointer dereference in `pcap_findalldevs_ex()` when `opendir()` fails,” https://nvd.nist.gov/vuln/detail/CVE-2024-8006, Aug. 2024, accessed: 2025-05-06.

[5] ——, “rpcapd/daemon.c in libpcap before 1.9.1 allows SSRF because a URL may be provided as a capture source,” https://nvd.nist.gov/vuln/detail/CVE-2019-15164, Oct. 2019, accessed: 2025-05-06.

[6] OSS-Fuzz Introspector, “Inspector report for libpcap,” https://storage.googleapis.com/oss-fuzz-introspector/libpcap/inspector-report/20250505/fuzz_report.html, May 2025, accessed 6 May 2025.