WHITEPAPER (/CATEGORY/WHITEPAPER)

Emulating WLAN in Linux - part II: mac80211_hwsim

Par Geoffrey Le Gourrierec (http://linuxembedded.fr/liste-articles-auteur/152) le 29 janvier 2021 (/2021/01/emulating-wlan-in-linux-part-ii-mac80211hwsim) • pas de commentaire (/2021/01/emulating-wlan-in-linux-part-ii-mac80211hwsim#comments)

Let's continue exploring the WLAN stack in Linux. In the first part of this series (https://www.linuxembedded.fr/2020/05/emulating-wlan-in-linux-part-i-the-80211-stack), we took a quick glance at what exactly a WLAN "interface" is in Linux, and studied the kernel modules handling them for us in userspace. We now have **sufficient knowledge to understand how the "virtual interface" mac80211_hswim works.** We shall do this using **practical use cases.**

All the examples given can be reproduced using the configuration files provided in the "Sources" paragraph. You may need to compile a kernel to run the tracing commands used in "mac8211_hwsim: a practical study"; I also provide a customized Buildroot [1] you can clone to build a QEMU image equipped with all the tools needed.

Quick recap of part I

Remember that a physical WLAN (802.11) network adapter is represented using two objects in the kernel:

- The **wiphy** represents the physical, autonomous 802.11-capable transceiver
- The **virtual interface** is both the standard Linux "network interface" on top of the wiphy, and also the manifestation of a specific mode for it

"Modes", as we discovered, determine what we can do with our interface: connecting to an access point is completely different from providing an access point, and no interface can do both at the same time. Juggling with modes is also part of the **netlink-supported API** that every interface has to fit into; this API is quite large and represents the modern way to communicate between kernel and userspace for everything related to WLAN.

Going below the API, we approached the **three kernel modules** making the magic work:

- **nl80211,** which offers the actual netlink communication.
- **cfg80211,** which is both a set of high-level hooks making the back-end of nl80211, and an important piece of code on its own. It handles common WLAN topics like describing adapter capabilities, following regulatory domains, and more generally all the glue that is not part of the standard, but that the industry considers generic functionality.
- **mac80211,** also called the *soft MAC module,* which offers software-based implementations of some (or all) of the protocol state machine, depending on the hardware's capabilities. We acknowledged that many modern devices handle quite a lot in hardware for performance reasons, but that usually a bit of code was delegated here, making mac80211 the real "driver" of your typical adapter.

Now that our memory has been refreshed, let's dive deeper into the code !

mac80211_hwsim: using it

Parameters

Just like every Linux kernel module, **mac80211_hwsim exposes a number of parameters** that we should consider when loading the module. Due to the testing/debugging use case behind mac80211_hwsim, we should always load it at runtime anyway, not statically link it in our kernel.

The most notable parameter is "*radios*", which selects the **number of physical (simulated) adapters** we want to spawn. **Each will live independently.** If we're trying to set up some kind of "ghost" client connexion to an access point, we only need one. If we wish to simulate trafic between some client stations and an access point, we may spawn two or more. The application-layer logic that we wish to test will be none the wiser.

Other parameters revolve around implementation details (e.g. "*paged_rx*" selects paged skbuff objects for reception) or protocol features (e.g. "*support_p2p_device*" will add P2P to available modes). Let's keep to "*radios*" for this article. We'll use it soon to **demonstrate a small simulated WLAN network.**

Usual manipulations

Let's set up a single simulated device, and launch common commands from a shell. *[Please note all following commands will require elevated privileges]* We will need a single wiphy:

```
$ modprobe mac80211_hwsim radios=1
mac80211_hwsim: initializing netlink
$ iw dev
phy#0
        Interface wlan0
                ifindex 4
                wdev 0x1
                addr 02:00:00:00:00:00
                type managed
                txpower 0.00 dBm
```

Alright, we got ourselves a new WLAN interface. Actually, we got a wiphy and one "regular" interface on top of it. Let's see what the wiphy can do:

```
$ iw phy phy0 info
```

The output is quite long, so here is a pastebin for iw phy output. (https://pastebin.com/XLvb1eAh) As you can see, **it looks just like a regular interface.** We've got access to **both 2.4GHz and 5GHz frequency bands,** and the **majority of modes** are available. That's one of the advantages of simulated HW: no limitations. :) We can manipulate it as usual, too. Let us say we need to inspect WLAN traffic from this interface; if you recall from the previous article, we need *monitor* mode for this.

```
# We need an active interface to change its channel
$ ip link set dev wlan0 up

$ iw dev wlan0 set type monitor
$ iw dev wlan0 set channel 8
$ iw dev
phy#0
        Interface wlan0
                ifindex 4
                wdev 0x1
                addr 02:00:00:00:00:00
                type monitor
                # We switched from channel 1 -> 8
                channel 8 (2447 MHz), width: 20 MHz (no HT), center1: 2447 MHz
                txpower 20.00 dBm
```

Now, of course with our setup we won't see any frame at all. This might raise the question: *but will we see any frame, anyway ?* Is the simulated adapter able to "see" through the real WLAN traffic ? The answer is no; **the simulated network is an enclosed space.** Which is actually another advantage from the tester point of view: complete control over the network environment. We could spawn 4 devices and make them talk to each other on channel 8, and our "monitoring interface" will see all that traffic, and only that.

The way this works is, even though our device cannot actually reach into the wireless spectrum, mac80211_hwsim maintains network packets queues like hardware would. Thus, **packet monitoring** and **packet injection** (both features provided by mac80211) work as expected, giving us as much control over the network as any device with a comprehensive driver implementation would.

Simple access point & station demo

We have looked at the link-layer configuration with the usual *iw* commands. The next step is to make the network go "live".

We would like to see an access point set up and broadcasting, and a single client station associating with it. We'll use WPA2 encryption with a passphrase. The tools we're going to use are the common ones in the Linux landscape:

- For setting up the access point, ***hostapd*** will implement the 802.11 state machine
  - Once we have a link-level access point, we'll use ***dhcpd*** to provide a DHCP server
- For setting up the station connection, ***wpa_supplicant*** will perform the 802.11 association handshake
  - After we're hooked, ***dhclient*** will get us an IP address

[French-speaking readers interested in a deeper understanding of wpa_supplicant and managed-mode tools should definitely check out Jean-Charles Bronner's article [2] on the topic.]

First, we'll need two distinct wiphy devices. Let's remove the previous one, and create another one (device creation/destruction are tied to module insertion/removal):

```
$ modprobe -r mac80211_hwsim
$ modprobe mac80211_hwsim radios=2
mac80211_hwsim: initializing netlink
```

The client station will need to be in "managed" mode, which is the default one. The access point will need, well, "access point" mode, which *hostapd* is smart enough to set by itself.

hostapd and *dhcpd* will need configuration files [3][4]. I will not delve into the details; both should be clear enough.

Once we launch *hostapd*, the access point will happily start broadcasting "beacons" to make itself known to the (simulated) world. We can prepare IP address assignation in parallel, so our access point becomes a real little gateway.

```
# Launch in background, write logs to file instead of stdout
$ hostapd -B -f hostapd.log -i wlan0 hostapd.conf

# Set IP @ first (otherwise dhcpd will frown and just exit)
$ ip addr add 192.168.42.1/24 dev wlan0
# Launch in background is implicit
$ dhcpd -cf dhcpd.conf wlan0
```

We're done on the access point side. Now, for the client, *wpa_supplicant* also requires a small configuration file [5]. Please note the configuration should match the one of the access point: SSID and encryption method (here, WPA2).

```
# Launch in background, write logs to file instead of stdout
$ wpa_supplicant -B -c wpa_supplicant.conf -f wpa_supplicant.log -i wlan1

# Get ourselves a fresh IP @, and go into background once the lease has been obtained
$ dhclient -4 wlan1
# Note: you may get the following warning message; do not worry, it does not mean
# you could not get an IP @
# cat: can't open '/etc/resolv.conf.*': No such file or directory
```

When *dhclient* gives back control, we can assume things went well: it means we got an IP address for wlan1. *[Note: you probably won't be able to ping one interface from the other at this point, because default routes created by Linux won't tell to respond from wlan0 if wlan0 got pinged from wlan1, and vice-versa. We'll focus on link-layer for this article; putting IP addresses on top of our network was merely for demonstration purposes.]*

```
# We got 192.168.42.2
$ ip addr show wlan1
7: wlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq qlen 1000
    link/ether 02:00:00:00:01:00 brd ff:ff:ff:ff:ff:ff
    inet 192.168.42.2/24 brd 192.168.42.255 scope global wlan1
       valid_lft forever preferred_lft forever
    inet6 fe80::ff:fe00:100/64 scope link
       valid_lft forever preferred_lft forever
```

I recommend you try to reproduce this, and take a look at the logs: they express the 802.11 handshake. Here is what I got (cut for clarity):

```
###########
### hostapd.log
###########

Configuration file: hostapd.conf
rfkill: Cannot open RFKILL control device
Using interface wlan0 with hwaddr 02:00:00:00:00:00 and ssid "SmileECS"
wlan0: interface state UNINITIALIZED->ENABLED
wlan0: AP-ENABLED
wlan0: interface state ENABLED->DISABLED
wlan0: AP-DISABLED
wlan0: CTRL-EVENT-TERMINATING
nl80211: deinit ifname=wlan0 disabled_11b_rates=0
Configuration file: hostapd.conf
rfkill: Cannot open RFKILL control device
Using interface wlan0 with hwaddr 02:00:00:00:00:00 and ssid "SmileECS"
wlan0: interface state UNINITIALIZED->ENABLED
wlan0: AP-ENABLED
wlan0: STA 02:00:00:00:01:00 IEEE 802.11: authenticated
wlan0: STA 02:00:00:00:01:00 IEEE 802.11: associated (aid 1)
wlan0: AP-STA-CONNECTED 02:00:00:00:01:00
wlan0: STA 02:00:00:00:01:00 WPA: pairwise key handshake completed (RSN)
wlan0: AP-STA-DISCONNECTED 02:00:00:00:01:00
wlan0: STA 02:00:00:00:01:00 IEEE 802.11: authenticated
wlan0: STA 02:00:00:00:01:00 IEEE 802.11: associated (aid 1)
wlan0: AP-STA-CONNECTED 02:00:00:00:01:00
wlan0: STA 02:00:00:00:01:00 WPA: pairwise key handshake completed (RSN)
wlan0: AP-STA-POLL-OK 02:00:00:00:01:00
wlan0: AP-STA-POLL-OK 02:00:00:00:01:00
wlan0: AP-STA-POLL-OK 02:00:00:00:01:00
wlan0: AP-STA-POLL-OK 02:00:00:00:01:00
wlan0: AP-STA-DISCONNECTED 02:00:00:00:01:00

###############
### wpa_supplicant.log
###############

Successfully initialized wpa_supplicant
rfkill: Cannot open RFKILL control device
wlan1: SME: Trying to authenticate with 02:00:00:00:00:00 (SSID='SmileECS' freq=2437 MHz)
wlan1: Trying to associate with 02:00:00:00:00:00 (SSID='SmileECS' freq=2437 MHz)
wlan1: Associated with 02:00:00:00:00:00
wlan1: CTRL-EVENT-SUBNET-STATUS-UPDATE status=0
wlan1: WPA: Key negotiation completed with 02:00:00:00:00:00 [PTK=CCMP GTK=CCMP]
wlan1: CTRL-EVENT-CONNECTED - Connection to 02:00:00:00:00:00 completed [id=0 id_str=]
wlan1: CTRL-EVENT-DISCONNECTED bssid=02:00:00:00:00:00 reason=3 locally_generated=1
nl80211: deinit ifname=wlan1 disabled_11b_rates=0
wlan1: CTRL-EVENT-TERMINATING
Successfully initialized wpa_supplicant
rfkill: Cannot open RFKILL control device
wlan1: SME: Trying to authenticate with 02:00:00:00:00:00 (SSID='SmileECS' freq=2437 MHz)
wlan1: Trying to associate with 02:00:00:00:00:00 (SSID='SmileECS' freq=2437 MHz)
```

```
wlan1: Associated with 02:00:00:00:00:00
wlan1: CTRL-EVENT-SUBNET-STATUS-UPDATE status=0
wlan1: WPA: Key negotiation completed with 02:00:00:00:00:00 [PTK=CCMP GTK=CCMP]
wlan1: CTRL-EVENT-CONNECTED - Connection to 02:00:00:00:00:00 completed [id=0 id_str=]
wlan1: CTRL-EVENT-DISCONNECTED bssid=02:00:00:00:00:00 reason=3
```

At this point one should be convinced we're just manipulating good ol' WLAN interfaces. In order to get a better grasp of how mac80211_hwsim interacts with those programs, we shall observe how small operations work.

mac80211_hwsim: a practical study

Now that we're convinced that these simulated adapters work just as well as the hardware they're imitating, let's take the chance to understand how the Linux 802.11 stack works with a couple of case studies. **We'll perform operations from userspace, and grab a magnifying glass** over the kernel to see **how a packet goes through the stack.**

> We are not trying to understand every line of code here. As such, you'll often find us skipping though minor blocks of code.

The "magnifying glass" shall be the **kernel tracing subsystem, *ftrace*.** After making sure our kernel has *ftrace* enabled, we'll make use of the *trace-cmd* user-friendly(ier) tool to record what's happening in the beast's entrails while we run specific commands. *[To be precise, we'll use the function tracer; ftrace enables other tracing methods, which we're not going to explore today.]* The *ftrace* system can be a bit daunting; for a nice, hands-on introduction to *trace-cmd*, I'd recommend this Red Hat tutorial [6]. For a more in-depth explanation, *ftrace* (and *trace-cmd*) author Steven Rostedt wrote an article on LWN [7] some time ago. It is still relevant, and well-written.

> While reading the following guided explanations, make sure to **grab a copy of the kernel code** and follow through it. Several links will point you directly to the relevant code block, in the Linux cgit Web service. But it's more entertaining to explore at will !

Case I: Changing WLAN channel

We already **picked a channel manually** before on an interface in monitor mode. How does it work ?

```
# Record every function call in the kernel, for the whole duration of the command
$ trace-cmd record -p function iw dev wlan1 set channel 8
  plugin 'function'
CPU0 data recorded at offset=0x182000
    208896 bytes in size
```

We got ourselves a quite large trace.dat (about 1.5MB), even though the command seemed nearly instantaneous to us ! A lot of stuff happened there, but take note that we did not filter events in any way. **Every function call was traced,** whether they're related to *iw* or not. **Let's be a bit smarter, and only look at functions containing "80211"** in their name. As kernel coding style goes, cfg80211 code contains mostly functions named "*cfg80211_foo()*" and "*cfg80211_baz()*", and so it goes for the other modules we're interested in: mac80211, nl80211, and mac80211_hswim.

```
$ trace-cmd record -p function -l '*80211*' iw dev wlan1 set channel 8
  plugin 'function'
CPU0 data recorded at offset=0x182000
    4096 bytes in size
```

The resulting file is not much smaller, but one should actually look at what *trace-cmd* says: "*4096 bytes in size*". We've eliminated *98% of noise* ! Your author assumes the file format has incompressible metadata, but has not looked into it. Anyway, it's **small enough to inspect easily:**

```
# Traces trimmed for clarity
$ trace-cmd report
  cpus=1
398.841150: function:             nl80211_pre_doit
398.841856: function:             nl80211_set_wiphy
398.841954: function:                __nl80211_set_channel
398.841984: function:                   nl80211_parse_chandef
398.842016: function:                      ieee80211_get_channel_khz
398.842045: function:                      cfg80211_chandef_create
398.842071: function:                      cfg80211_chandef_valid
398.842100: function:                      cfg80211_chandef_usable
398.842120: function:                         cfg80211_chandef_valid
398.842133: function:                         cfg80211_secondary_chans_ok
398.842161: function:                            ieee80211_get_channel_khz
398.842187: function:                   cfg80211_set_monitor_channel
398.842230: function:                      ieee80211_set_monitor_channel
398.842250: function:                         ieee80211_hw_config
398.842269: function:                            cfg80211_chandef_valid
398.842284: function:                            mac80211_hwsim_config
398.842327: function:             nl80211_post_doit
398.842810: function:             nl80211_netlink_notify
398.842832: function:                cfg80211_mlme_unregister_socket
398.842859: function:                cfg80211_release_pmsr
398.842877: function:                cfg80211_mlme_unregister_socket
398.842880: function:                cfg80211_release_pmsr
398.842893: function:             mac80211_hwsim_netlink_notify
```

1. **nl80211:** Recall *iw* uses the nl80211 user <-> kernel interface to do its magic. Hence, we pass first through the nl80211 module, which pipes our request down to the kernel using the netlink packet format.
    1. We arrive at *nl80211_set_wiphy()*. The kernel starts by getting the network interface (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/wireless/nl80211.c?h=v5.10#n3117) we want to fiddle with.
    2. Then, it finds the *NL80211_ATTR_WIPHY_FREQ* attribute (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/wireless/nl80211.c?h=v5.10#n3188) and understands we want to set our channel ("central frequency").
    3. *__nl80211_set_channel()* is the actual trigger for change. It starts by checking if the channel can be changed (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/wireless/nl80211.c?h=v5.10#n3011) directly (which we could not do ourselves if wlan1 was in "managed" mode for example).
    4. We're good, so *nl80211_parse_chandef()* is invoked to take the given frequency (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/wireless/nl80211.c?h=v5.10#n2895) (*nla_get_u32()* reminds us it's a 32-bit unsigned integer) and convert it

    (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/wireless/nl80211.c?h=v5.10#n2902) to a definition suitable to the kernel.
    5. After all this format handling, we jump into the configuration layer with *cfg80211_set_monitor_channel()* (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/wireless/nl80211.c?h=v5.10#n3052) to really change the channel.
2. **cfg80211:** We reached the layer that handles all the big levers a user might want to move. It does more than nl80211, and constructs complex operations from the set of low-level primitives the rest of the stack offers.
    1. If you search in the Linux source code, you may be surprised to find several vendor-specific "cfg80211.c" files. There is a "generic" part though: it lies in *net/wireless/* and we start in that generic part. (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/wireless/chan.c?h=v5.10#n1236)
    2. Our generic layer calls into the driver-specific channel setting function: (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/wireless/rdev-ops.h?h=v5.10#n429) *ops->set_monitor_channel(struct wiphy *, struct cfg80211_chan_def *)*
    3. Surprise, our "driver" is actually mac80211 ! (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/cfg.c?h=v5.10#n781) But why ? Remembering the role of mac80211, we realize that it is meant as a "soft" MAC driver. Had we used a driver doing everything in hardware (so-called "hard" MAC), we would have jumped directly to the driver code. We now understand the MAC layer handling in Linux is very flexible; that is how most driver writers actually delegate some functionality to mac80211, making compromises as they see fit.
    4. After making sure the required channel is different (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/cfg.c?h=v5.10#n788) from the current one (I made sure so before recording those traces !), we reach *ieee80211_hw_config()* (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/main.c?h=v5.10#n168) which is kind of a meeting point for making sure config changes are performed in a synchronous way between hardware and us. The actual driver code is invoked, (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/main.c?h=v5.10#n181) using an inlined wrapper. (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/driver-ops.h?h=v5.10#n145) Finally, we're in mac80211_hwsim !
3. **mac80211_hwsim:** This will actually be a short trip, compared to previous steps.

    1. A big chunk of code checks if the interface was actively scanning the network, (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n1848) jumping from channel to channel. If it was, we register the end the period spent on the previous channel, update our active channel, (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n1858) and register the start of a new period. *[Scanning is a user-requestable background activity. For instance, if* iw phy *shows you "register_beacons" in the supported command set, you can schedule yours. This "command set" is no magic: it is based on nl80211, too.]*
    2. If we were not scanning, we just update our active channel. (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n1869)
    3. The last bit is not intuitive for a monitor mode interface, but if we were beaconing, we need to stop the existing beacon timer and prepare a new one (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n1873) for this channel. Recall beacons are always sent on a given period, hence the timer always starts anew with *data->beacon_int* microseconds.
4. **nl80211:** Back here, to notify our userspace client that the operation was completed. Netlink provides asynchronous responses; this manifests with *nl80211_netlink_notify()*. (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/wireless/nl80211.c?h=v5.10#n17548) Nothing very exciting here, but note a mac80211_hwsim hook is called at the very end; meaning it is able to plug in some of its specifics if it wants to. We'll reflect on this in the second case study.

Such a long journey it was, just to change a 32-bit number...

Case II: Injecting a forged frame

The previous example allowed us to inspect the configuration layer (cfg80211) from the eye of mac80211_hwsim. The network layer (mac80211) remained discreet; packet injection sounds like a neat way to probe into it. **We'll send a single 802.11 frame and "follow it".**

We'll rely on *scapy* [8] to forge and send our little packet. *Scapy* is both a very powerful packet manipulation library and a companion command-line tool, that goes far beyond IEEE 802.11 as protocols go. I can't recommend it enough for quick hacks and even test scripts; it's a great tool. Now, we don't really care about the contents nor the nature of the packet; we just want to exercise the code path used for transmitting frames.

```
#! /usr/bin/python3

from scapy.all import *

pkt = Dot11(addr1='02:00:00:00:00:00', addr2='02:00:00:00:01:00') / Dot11Deauth(reason=8)
print("Sending the following packet:")
pkt.show()
sendp(pkt, count=1, iface='wlan1')
```

The kernel is asked to **send a *deauthentication* frame** to MAC address *02:00:00:00:00:00*, using *02:00:00:00:01:00* as the source. A deauthentication frame "cuts" a WLAN association between a station and an access point; it is usually sent by the latter to the former, because something impairs the radio connection and the link cannot be maintained. The actual reason is encoded in an integer.

The source MAC address does not matter: what counts is the interface we're asking to write from (wlan1). Packet forging means we can write pretty much any value here, which is very convenient, and made possible by the flexibility of mac80211.

Let's send the packet and get new traces.

```
$ trace-cmd record -p function -l '*80211*' ./send_deauth_pkt.py
  plugin 'function'
Sending the following packet:
###[ 802.11 ]###
  subtype   = Deauthentification
  type      = Management
  proto     = 0
  FCfield   =
  ID        = 0
  addr1     = 02:00:00:00:00:00 (RA=DA)
  addr2     = 02:00:00:00:01:00 (TA=SA)
  addr3     = 00:00:00:00:00:00 (BSSID/STA)
  SC        = 0
###[ 802.11 Deauthentication ]###
     reason    = disas-ST-leaving

device wlan1 entered promiscuous mode
.
Sent 1 packets.
device wlan1 left promiscuous mode
CPU0 data recorded at offset=0x182000
    4096 bytes in size
```

```
# Traces trimmed for clarity
$ trace-cmd report
  cpus=1
5148.720771: function:             ieee80211_get_stats64
5148.723061: function:             ieee80211_set_multicast_list
5148.723175: function:                ieee80211_queue_work
5148.733070: function:             ieee80211_reconfig_filter
5148.733171: function:                ieee80211_configure_filter
5148.733357: function:                   mac80211_hwsim_configure_filter
5148.797108: function:             ieee80211_monitor_select_queue
5148.797173: function:                ieee80211_parse_tx_radiotap
5148.797211: function:                   ieee80211_radiotap_iterator_init
5148.797292: function:                   ieee80211_radiotap_iterator_next
5148.797406: function:                ieee80211_hdrlen
5148.797450: function:                ieee80211_select_queue_80211
5148.798233: function:             ieee80211_monitor_start_xmit
5148.798276: function:                ieee80211_parse_tx_radiotap
5148.798277: function:                   ieee80211_radiotap_iterator_init
5148.798279: function:                   ieee80211_radiotap_iterator_next
5148.798309: function:                ieee80211_hdrlen
5148.798383: function:                cfg80211_reg_can_beacon
5148.798393: function:                   _cfg80211_reg_can_beacon
5148.798422: function:                      cfg80211_chandef_dfs_required
5148.798436: function:                         cfg80211_chandef_valid
5148.798567: function:                      cfg80211_chandef_usable
5148.798580: function:                         cfg80211_chandef_valid
5148.798644: function:                         cfg80211_secondary_chans_ok
5148.798668: function:                            ieee80211_get_channel_khz
5148.798861: function:                ieee80211_xmit
5148.798911: function:                   ieee80211_skb_resize
5148.798971: function:                   ieee80211_set_qos_hdr
5148.798994: function:                   ieee80211_tx
5148.799017: function:                      ieee80211_tx_prepare
5148.799432: function:             ieee80211_tx_h_select_key
5148.799468: function:             ieee80211_tx_h_rate_ctrl
5148.799875: function:                      ieee80211_queue_skb
5148.799920: function:             ieee80211_tx_h_michael_mic_add
5148.799984: function:             ieee80211_tx_h_encrypt
5148.800317: function:             ieee80211_frame_duration
5148.800388: function:                      __ieee80211_tx.constprop.0
5148.800448: function:                         ieee80211_tx_frags
5148.800534: function:                            mac80211_hwsim_tx
5148.800606: function:                               mac80211_hwsim_monitor_rx.isra.0
5148.800673: function:                               mac80211_hwsim_tx_frame_no_nl.isra.0
5148.801102: function:                                  mac80211_hwsim_addr_match
5148.801127: function:                                     ieee80211_iterate_active_interfaces_atomic
5148.801208: function:             mac80211_hwsim_addr_iter
5148.801279: function:                                  ieee80211_rx_irqsafe
5148.801417: function:                               ieee80211_tx_status_irqsafe
5148.801827: function:             ieee80211_tasklet_handler
5148.801906: function:                ieee80211_rx_napi
5148.801923: function:                   ieee80211_rx_list
```

```
5148.802109: function:                      ieee80211_rx_radiotap_hdrlen.isra.0
5148.802385: function:                      ieee80211_add_rx_radiotap_header
5148.802611: function:                         ieee80211_calculate_rx_timestamp
5148.802693: function:                            cfg80211_calculate_bitrate
5148.803420: function:                      ieee80211_clean_skb
5148.803499: function:                      ieee80211_hdrlen
5148.803701: function:             ieee80211_tasklet_handler
5148.803724: function:                ieee80211_tx_status
5148.803876: function:                   ieee80211_tx_status_ext
5148.804006: function:                      ieee80211_report_used_skb
5148.804098: function:                      ieee80211_tx_monitor
5148.811952: function:             ieee80211_get_stats64
5148.812046: function:             ieee80211_set_multicast_list
5148.812053: function:                ieee80211_queue_work
5148.816706: function:             ieee80211_reconfig_filter
5148.816716: function:                ieee80211_configure_filter
5148.816723: function:                   mac80211_hwsim_configure_filter
```

Again, a relatively small set of functions is obtained. Let's peek.

1. **No nl80211:** This time, note we're *not* passing through nl80211: we do not need to reach cfg80211.
2. **Off-topic:** The first two functions are kind of unrelated to our packet.
    1. *ieee80211_get_stats64()* just returns some information on the network device, (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/iface.c?h=v5.10#n710) probably asked by Scapy.
    2. *ieee80211_set_multicast_list()* bridges to the kernel "all multicast" logic. (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/iface.c?h=v5.10#n653) This is not related to IEEE 802.11 only, and configures an interface-specific *filter* for reception of incoming multicast frames, that takes all frames if the associated counter is greater than zero. After fiddling with the counter, it spawns a kernel worker (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/iface.c?h=v5.10#n673) for *reconfig_filter()*. By the way, we can see it got scheduled immediately. But let's not go further in that direction.
3. **mac80211:** The meat of the subject.
    1. The kernel starts by selecting a HW queue for transmission, (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/iface.c?h=v5.10#n726) in *ieee80211_monitor_select_queue()*. Actually, it does more than this, because it parses the RadioTap header first, (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/iface.c?h=v5.10#n742) and drops our frame if the header is not acceptable. *RadioTap* [9] is an intermediate protocol carrying metadata on a radio link, and is implicitly used by pretty much all WLAN devices out there. Originally, it was made to carry information upper layers would not provide (the header is located *before* the 802.11 header in a frame), but Linux also uses some

of its fields for implementation-specific needs, like waiting for an ACK or not depending on *"TX flags (https://www.radiotap.org/fields/TX%20flags.html)*".

2. The actual queue selection happens in *ieee80211_select_queue_80211()*. Our simulated device always has five queues, (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n3103) as implemented in the virtual wiphy creation function. Hence, we pass the first check (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/wme.c?h=v5.10#n123) that considers always using the first queue for devices with a limited number of them. 802.11 embeds the notion of quality of service with "access categories"; a QoS-compatible device should then have at least *IEEE80211_NUM_ACS* queues. (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/ieee80211.h?h=v5.10#n250) mac80211_hwsim can do everything, I tell you :) Our frame is a management frame, (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/wme.c?h=v5.10#n128) not a data frame, so we automatically pick a queue with high priority (7).
3. Next step: transmitting the frame. *ieee80211_monitor_start_xmit()* (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n2223) is the door to sending our bytes "over the air". We parse again the RadioTap header (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n2240) but from the transmission perspective. As mentioned before, some fields are used also for transmission behaviour by Linux; we have an empty RadioTap header, so we're not interested in that.
4. Going forward, mac80211 starts preparing the skbuff structure (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n2247) for the soon-to-be-sent packet. It starts by resetting lengths of MAC/network/transport headers, because we're at the bottom of the stack and we do not know what's coming next.
5. We're still in code not specific to our network device, so Linux guesses the correct interface by MAC address: (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n2293) *ether_addr_equal(tmp_sdata->vif.addr /* current iface */, hdr->addr2 /* src addr field in 802.11 header */)*
6. A small jump to cfg80211 occurs to take into account regulatory requirements: (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n2334) injection won't be authorized if the *channel* requires proper protection. *[This*

can happen e.g. in 5GHz band, where some channels collide with radar, as mentioned in the comments.]

7. Now we're ready to transmit (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n1988) with *ieee80211_xmit()*. Code will start to worry about priorities, flags and other bizarre objects. Your author suggests skipping the code making room for encryption (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n1997) and handling mesh networks, (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n2019)which are too specific for our present study.
8. "Quality of Service" follows what we've seen previously. We're not injecting a *data* frame, so we actually drop out of *ieee80211_set_qos_hdr()* (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/wme.c?h=v5.10#n247) quickly. We got priority 7, though, and that still counts: people implementing traffic control on a Linux box look at this value to prioritize packets.
9. As the sta argument is NULL (because of this line) *ieee80211_tx_prepare()* falls into the simple unicast case, (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n1203) and get one or two flags set on the way, here (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n1231) and there. (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n1247) Our packet continues its glorious journey. *[The astute reader will note only* data *frames can be queued for later transmission.]*
10. Because of our filter, *invoke_tx_handlers_*()* do not appear in the list of calls. Rest assured they are: they're the reason behind the sudden appearance of seemingly unrelated *ieee80211_tx_h_*()* calls, albeit with a different indentation than our current one in the traces. They are mostly related to encryption, so let us ignore them safely.
11. Being in monitor mode, our interface causes skipping packet queueing again: (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n1610) we're very busy, please send it ASAP ! :)
12. We've gone through five point in this list talking about sending the frame, and we're still not there. But getting closer: we reach __ieee80211_tx() (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n1897). This function actually dequeues the packet queue (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n1721), and calls our driver code to send it off

    (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/mac80211/tx.c?h=v5.10#n1698) in *ieee80211_tx_frags()*. As you can see, the majority of the code is dedicated to queue handling: dropping previous packets still in the pipe ("*if*"), or taking the chance to fill in extra ones ("*else*"). Finally, our packet reaches the driver.

4. **mac80211_hwsim:** Entering the software simulation...
    1. Right off the bat, we grab channel context (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n1534) and check some magic values [useful for testing, ndlr].
    2. A little further, we reach *mac80211_hswim_monitor_rx()*. (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n940) But wait: why a reception function ? We're trying to send something, are we not ?
    3. Actually, this is where we forge our final packet: we create the actual RadioTap header (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n963) and activate the receive path of the *hwsim_mon* virtual device. (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n982) This device serves as an intermediate buffer for packet injection: later on, the received path of our (simulated) destination (*02:00:00:00:00:00*) will pump the packet right back from it !
    4. We're nearing the end of our tour. Some other tasks should still pique your interest...
    5. Firstly, notice mac80211_hwsim can send an extra netlink notification to userspace (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n1188) after "emitting" our frame. As a tester, I'd be glad to have such a mecanism: it allows me to write a nice testing program, tracking the WLAN state machine more than an opaque device would allow. In short, this can be conditioned (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n1591) by the use of the companion *wmediumd* [10] tool, which does exactly that job. You'll find references to it scattered across the code.
    6. If we were not configured to work with *wmediumd*, we track frame count (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n1594) in kernelspace and send immediately an ACK frame (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/mac80211_hwsim.c?h=v5.10#n992) if we were told to (remember the RadioTap flags)
5. **Off-topic:** To conclude this effort, we're mostly cleaning up and sending information back now. A lot of these small tasks are done asynchronously, hence the *tasklets* popping up one after another. Let's stop here...

And we're done. Now we have a pretty good idea of what the kernel does to send a WLAN packet with mac80211_hswim. At least on the transmission side, the mysteries of the Linux kernel 802.11 stack have been solved ! And for that, we can thank the contributors who built mac80211_hwsim, which makes for a perfect learning tool.

I hope you enjoyed this article; do not hesitate to drop a comment. Happy (WLAN) hacking !

Sources

[1] Buildroot sources for building a QEMU "playground" image, with mac80211_hwsim and tools used in this article: https://gitlab.com/clumsyape/buildroot/-/tree/mac80211_hwsim_demo (https://gitlab.com/clumsyape/buildroot/-/tree/mac80211_hwsim_demo)
[2] *"Présentation des wireless daemon sous Linux"*, Jean-Charles Bronner: https://www.linuxembedded.fr/2020/07/presentation-des-wireless-daemon-sous-linux (https://www.linuxembedded.fr/2020/07/presentation-des-wireless-daemon-sous-linux)
[3] Demo hostapd.conf: https://pastebin.com/ZH6FCBkM (https://pastebin.com/ZH6FCBkM)
[4] Demo dhcpd.conf: https://pastebin.com/UPt7wzaa (https://pastebin.com/UPt7wzaa)
[5] Demo wpa_supplicant.conf: https://pastebin.com/guKJvVqn (https://pastebin.com/guKJvVqn)[6] Red Hat tutorial on *trace-cmd*: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/latency_tracing_using_trace-cmd (https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/latency_tracing_using_trace-cmd)
[7] *"trace-cmd: A front-end for Ftrace"*, Steven Rostedt: https://lwn.net/Articles/410200/ (https://lwn.net/Articles/410200/)
[8] Scapy documentation: https://scapy.readthedocs.io/en/latest/ (https://scapy.readthedocs.io/en/latest/)
[9] RadioTap protocol common fields: https://www.radiotap.org/fields/defined (https://www.radiotap.org/fields/defined)
[10] *wmediumd*, a userspace testing tool working in tandem with mac80211_hswim: https://github.com/bcopeland/wmediumd (https://github.com/bcopeland/wmediumd)

Mots-clés : kernel (/tag/kernel) Linux (/tag/linux) network (/tag/network) wifi (/tag/wifi) wlan (/tag/wlan)

← Linux MMUless (/2020/12/linux-mmuless)

Laisser un commentaire

Votre adresse de messagerie ne sera pas publiée.

Commentaire

Nom

Adresse de messagerie

Site web

Laisser un commentaire

Catégories

Actualité (/category/actualite)
HowTo (/category/howto)
Interview (/category/interview)
Non classé (/category/non-classe)
Technologie (/category/technologie)
WhitePaper (/category/whitepaper)

Archives

Mars 2021 (/liste-articles/202103)
Janvier 2021 (/liste-articles/202101)
Décembre 2020 (/liste-articles/202012)
Octobre 2020 (/liste-articles/202010)
Septembre 2020 (/liste-articles/202009)

Août 2020 (/liste-articles/202008)
Juillet 2020 (/liste-articles/202007)
Juin 2020 (/liste-articles/202006)
Mai 2020 (/liste-articles/202005)
Avril 2020 (/liste-articles/202004)
Mars 2020 (/liste-articles/202003)
Février 2020 (/liste-articles/202002)
Janvier 2020 (/liste-articles/202001)
Décembre 2019 (/liste-articles/201912)
Novembre 2019 (/liste-articles/201911)
Octobre 2019 (/liste-articles/201910)
Septembre 2019 (/liste-articles/201909)
Août 2019 (/liste-articles/201908)
Juillet 2019 (/liste-articles/201907)
Juin 2019 (/liste-articles/201906)
Mai 2019 (/liste-articles/201905)
Avril 2019 (/liste-articles/201904)
Mars 2019 (/liste-articles/201903)
Février 2019 (/liste-articles/201902)
Décembre 2018 (/liste-articles/201812)
Novembre 2018 (/liste-articles/201811)
Octobre 2018 (/liste-articles/201810)
Septembre 2018 (/liste-articles/201809)
Août 2018 (/liste-articles/201808)
Juillet 2018 (/liste-articles/201807)
Juin 2018 (/liste-articles/201806)
Mai 2018 (/liste-articles/201805)
Avril 2018 (/liste-articles/201804)
Mars 2018 (/liste-articles/201803)
Février 2018 (/liste-articles/201802)
Janvier 2018 (/liste-articles/201801)
Décembre 2017 (/liste-articles/201712)
Novembre 2017 (/liste-articles/201711)
Octobre 2017 (/liste-articles/201710)
Septembre 2017 (/liste-articles/201709)
Août 2017 (/liste-articles/201708)
Juillet 2017 (/liste-articles/201707)
Juin 2017 (/liste-articles/201706)
Mai 2017 (/liste-articles/201705)
Avril 2017 (/liste-articles/201704)
Mars 2017 (/liste-articles/201703)
Février 2017 (/liste-articles/201702)
Janvier 2017 (/liste-articles/201701)
Décembre 2016 (/liste-articles/201612)
Novembre 2016 (/liste-articles/201611)

Octobre 2016 (/liste-articles/201610)
Septembre 2016 (/liste-articles/201609)
Août 2016 (/liste-articles/201608)
Juillet 2016 (/liste-articles/201607)
Mai 2016 (/liste-articles/201605)
Mars 2016 (/liste-articles/201603)
Février 2016 (/liste-articles/201602)
Décembre 2015 (/liste-articles/201512)
Octobre 2015 (/liste-articles/201510)
Août 2015 (/liste-articles/201508)
Juillet 2015 (/liste-articles/201507)
Juin 2015 (/liste-articles/201506)
Mai 2015 (/liste-articles/201505)
Avril 2015 (/liste-articles/201504)
Mars 2015 (/liste-articles/201503)
Février 2015 (/liste-articles/201502)
Janvier 2015 (/liste-articles/201501)
Décembre 2014 (/liste-articles/201412)
Novembre 2014 (/liste-articles/201411)
Octobre 2014 (/liste-articles/201410)
Septembre 2014 (/liste-articles/201409)
Août 2014 (/liste-articles/201408)
Juillet 2014 (/liste-articles/201407)
Juin 2014 (/liste-articles/201406)
Mai 2014 (/liste-articles/201405)
Avril 2014 (/liste-articles/201404)
Mars 2014 (/liste-articles/201403)
Février 2014 (/liste-articles/201402)
Janvier 2014 (/liste-articles/201401)
Décembre 2013 (/liste-articles/201312)
Novembre 2013 (/liste-articles/201311)
Octobre 2013 (/liste-articles/201310)
Septembre 2013 (/liste-articles/201309)
Août 2013 (/liste-articles/201308)
Juillet 2013 (/liste-articles/201307)
Juin 2013 (/liste-articles/201306)
Mai 2013 (/liste-articles/201305)
Avril 2013 (/liste-articles/201304)
Mars 2013 (/liste-articles/201303)
Février 2013 (/liste-articles/201302)
Janvier 2013 (/liste-articles/201301)
Septembre 2012 (/liste-articles/201209)
Avril 2012 (/liste-articles/201204)
Mars 2012 (/liste-articles/201203)
Janvier 2012 (/liste-articles/201201)

Décembre 2011 (/liste-articles/201112)
Octobre 2011 (/liste-articles/201110)
Août 2011 (/liste-articles/201108)
Juin 2011 (/liste-articles/201106)
Mai 2011 (/liste-articles/201105)
Mars 2011 (/liste-articles/201103)
Février 2011 (/liste-articles/201102)
Janvier 2011 (/liste-articles/201101)
Décembre 2010 (/liste-articles/201012)
Novembre 2010 (/liste-articles/201011)
Octobre 2010 (/liste-articles/201010)
Septembre 2010 (/liste-articles/201009)