Wpa_supplicant workflow analysis

21 September 2024 12:49

Wpa_supplicant workflow analysis

wpa_supplicant official website: https://w1.fi/wpa_supplicant/
wpa_supplicant source code download official website address: https://w1.fi/releases/

This article is based on the version: V2.6

1. Initialization

wpa_supplicant/main.c

In main(), four things are done.
Let's take a look at the source code first, and then analyze them one by one.

```
int main(int argc, char *argv[])
{

 ... //Loop to receive incoming parameters

for (;;) {
c = getopt(argc, argv, "b:Bc:C:D:de:f:g:G:hi:I:KLMm:No:O:p:P:qsTtuvW");
if (c < 0) break;
switch (c)
{

case 'b': iface->bridge_ifname = optarg ; break;
case 'B': params.daemonize++;break;

case 'c': iface->confname = optarg; break;
case 'C': iface->ctrl_interface = optarg; break;

case 'D': iface->driver = optarg; break;
case 'd':
#ifdef CONFIG_NO_STDOUT_DEBUG
printf("Debugging disabled with " "CONFIG_NO_STDOUT_DEBUG=y build time " "option.\n");
goto out;
#else /* CONFIG_NO_STDOUT_DEBUG */
params.wpa_debug_level--; break;
#endif /* CONFIG_NO_STDOUT_DEBUG */

case 'e': params.entropy_file = optarg; break;

#ifdef CONFIG_DEBUG_FILE
case ' f': params.wpa_debug_file_path = optarg; break;
#endif /* CONFIG_DEBUG_FILE */

case 'g': params.ctrl_interface = optarg; break;
case 'G': params.ctrl_interface_group = optarg; break;

case 'h': usage(); exitcode = 0; goto out;
case 'i': iface->ifname = optarg; break;
case 'I': iface->confanother = optarg; break;

case 'K': params.wpa_debug_show_keys++; break;
case 'L': license(); exitcode = 0; goto out;

#ifdef CONFIG_P2P
case 'm': params.conf_p2p_dev = optarg; break;
#endif /* CONFIG_P2 P */

case 'o': params.override_driver = optarg; break;
case 'O': params.override_ctrl_interface = optarg; break;

case 'p': iface->driver_param = optarg; break;
case 'P': os_free(params.pid_file); params.pid_file = os_rel2abs_path(optarg); break;

case 'q': params.wpa_debug_level++; break;

#ifdef CONFIG_DEBUG_SYSLOG
case 's': params.wpa_debug_syslog++; break;
#endif /* CONFIG_DEBUG_SYSLOG */

#ifdef CONFIG_DEBUG_LINUX_TRACING
case 'T': params.wpa_debug_tracing++ ; break;
#endif /* CONFIG_DEBUG_LINUX_TRACING */

case 't': params.wpa_debug_timestamp++; break;

#ifdef CONFIG_DBUS
case 'u': params.dbus_ctrl_interface = 1; break;
#endif /* CONFIG_DBUS */

case 'v': printf("%s\n", wpa_supplicant_version); exitcode = 0; goto out;

case 'W': params.wait_for_monitor++; break;

#ifdef CONFIG_MATCH_IFACE
case 'M': params.match_iface_count++;
iface = os_realloc_array(params.match_ifaces, params.match_iface_count, sizeof(struct wpa_interface)) ;
if (!iface) goto out; params.match_ifaces = iface; iface = &params.match_ifaces[params.match_iface_count - 1];
os_memset(iface, 0, sizeof(*iface)); break;
#endif /* CONFIG_MATCH_IFACE */
case 'N': iface_count++;
iface = os_realloc_array(ifaces, iface_count, sizeof(struct wpa_interface));
if (iface == NULL) goto out; ifaces = iface; iface = &ifaces[iface_count - 1];
os_memset(iface, 0, sizeof(*iface));break; default: usage(); exitcode = 0; goto out; } }
exitcode = 0; //Start initialization with the passed parameters global = wpa_supplicant_init(&params);
if (global == NULL) { wpa_printf(MSG_ERROR, "Failed to initialize wpa_supplicant"); exitcode = -1; goto out; }
else
{
wpa_printf(MSG_INFO, "Successfully initialized "
"wpa_supplicant");
}
......
//Call wpa_supplicant_add_iface to add the control interface. Note that global here is returned by wpa_supplicant_init()
wpa_s = wpa_supplicant_add_iface(global, &ifaces[i], NULL);
if (wpa_s == NULL) {
exitcode = -1;
break;
}
}
#ifdef CONFIG_MATCH_IFACE
if (exitcode == 0)
exitcode = wpa_supplicant_init_match(global);
#endif /* CONFIG_MATCH_IFACE */
//If the control window is added successfully, call wpa_supplicant_run() to run the wpa_supplicant event main loop
if (exitcode == 0)
exitcode = wpa_supplicant_run(global);
......
}
```

1. Parse the relevant parameters of init.rc
   (in this case, it is the operation of wpa_supplicant when the device starts the wpa_supplicant process)

    or

   user input (such as when we manually start the wpa_supplicant process).
   The general format is as follows:
   ```
    wpa_supplicant/system/bin/wpa_supplicant -D wext -I wlan0 -c /data/misc/wifi/wpa_supplicant.conf
      -D: indicates the type of driver to use, generally the default is wext. Now most of them use nl80211.
       -i: indicates the network interface name
       -c: indicates the configuration file name
   ```
   From the source code, we can see that in main, wpa_supplicant will fill the two structures iface and param according to the parameters passed in.

2. Call wpa_supplicant_init() to perform some initialization operations.
   The data pointer returned by wpa_supplicant_init() can be used to add and remove network interfaces.

3. Call wpa_supplicant_add_iface() to add a control interface.
   Create a wpa_supplicant structure.
   Call wpa_supplicant_init_iface() to do some work to initialize the interface.

4. Call wpa_supplicant_run() to run the wpa_supplicant event main loop.
   After configuring the eloop loop, call eloop_run() to run the eloop loop.
   Then wpa_supplicant handles commands from the upper layer and events from the lower layer in this loop.

2. How wpa_supplicant receives and responds to upper-level commands

```
void eloop_run(void)
{

#ifdef CONFIG_ELOOP_POLL int num_poll_fds; int timeout_ms = 0;
#endif /* CONFIG_ELOOP_POLL */

#ifdef CONFIG_ELOOP_SELECT fd_set *rfds, *wfds, *efds; struct timeval _tv;
#endif /* CONFIG_ELOOP_SELECT */

#ifdef CONFIG_ELOOP_EPOLL int timeout_ms = -1;
#endif /* CONFIG_ELOOP_EPOLL */

#ifdef CONFIG_ELOOP_KQUEUE
struct timespec ts;
#endif /* CONFIG_ELOOP_KQUEUE */ int res; struct os_reltime tv, now;

#ifdef CONFIG_ELOOP_SELECT
rfds = os_malloc(sizeof(*rfds));
wfds = os_malloc(sizeof(*wfds));
efds = os_malloc(sizeof(*efds));
if (rfds == NULL || wfds = = NULL || efds == NULL) goto out;
#endif /* CONFIG_ELOOP_SELECT */

while (!eloop.terminate && (!dl_list_empty(&eloop.timeout) || eloop.readers.count > 0 || eloop.writers.count > 0 || eloop.exceptions.count > 0)) {
struct eloop_timeout *timeout;

if (eloop.pending_terminate) {
 /* * This may happen in some corner cases where a signal * is received during a blocking operation. We need to * process the pending signals and exit if requested to * avoid hitting the SIGALRM limit if the blocking * operation took more than two seconds. */
eloop_process_pending_signals();
if (eloop.terminate) break;
}
timeout = dl_list_first(&eloop.timeout, struct eloop_timeout, list);
if (timeout) { os_get_reltime(&now);

if (os_reltime_before(&now, &timeout->time))
os_reltime_sub(&timeout->time, &now, &tv);
Else
tv. sec = tv.usec = 0;
```

```
#if defined(CONFIG_ELOOP_POLL) || defined(CONFIG_ELOOP_EPOLL)
timeout_ms = tv.sec * 1000 + tv.usec / 1000;
#endif /* defined(CONFIG_ELOOP_POLL) || defined(CONFIG_ELOOP_EPOLL) */

#ifdef CONFIG_ELOOP_SELECT _tv.tv_sec = tv.sec; _tv.tv_usec = tv.usec;
#endif /* CONFIG_ELOOP_SELECT */

#ifdef CONFIG_ELOOP_KQUEUE ts.tv_sec = tv.sec ; ts.tv_nsec = tv.usec * 1000L;
#endif /* CONFIG_ELOOP_KQUEUE */ }

#ifdef CONFIG_ELOOP_POLL num_poll_fds = eloop_sock_table_set_fds( &eloop.readers, &eloop.writers, &eloop.exceptions, eloop.pollfds, eloop.pollfds_map, eloop.max_pollfd_map);

res = poll(eloop.pollfds, num_poll_fds, timeout ? timeout_ms : -1);
#endif /* CONFIG_ELOOP_POLL */

#ifdef CONFIG_ELOOP_SELECT eloop_sock_table_set_fds(&eloop.readers, rfds); eloop_sock_table _set_fds(&eloop. writers, wfds); eloop_sock_table_set_fds(&eloop.exceptions, efds); res = select(eloop.max_sock + 1, rfds, wfds, efds,
timeout ? &_tv : NULL);
#endif /* CONFIG_ELOOP_SELECT */

#ifdef CONFIG_ELOOP_EPOLL
if (eloop.count == 0) { res = 0; }
else { res = epoll_wait(eloop.epollfd, eloop.epoll_events, eloop.count, timeout_ms); }
#endif /* CONFIG_ELOOP_EPOLL */

#ifdef CONFIG_ELOOP_KQUEUE
if ( eloop.count == 0) { res = 0; }
else { res = kevent(eloop.kqueuefd, NULL, 0, eloop.kqueue_events, eloop.kqueue_nevents, timeout ? &ts : NULL); }
#endif /* CONFIG_ELOOP_KQUEUE */

if (res < 0 && errno != EINTR && errno != 0) {
wpa_printf(MSG_ERROR, "eloop: %s: %s",

#ifdef CONFIG_ELOOP_POLL "poll"
#endif /* CONFIG_ELOOP_POLL */

#ifdef CONFIG_ELOOP_SELECT "select"
#endif /* CONFIG_ELOOP_SELECT */

#ifdef CONFIG_ELOOP_EPOLL "epoll"
#endif / * CONFIG_ELOOP_EPOLL */

#ifdef CONFIG_ELOOP_KQUEUE "kqueue"
#endif /* CONFIG_ELOOP_EKQUEUE */, strerror(errno)); goto out;

}

eloop.readers.changed = 0; eloop.writers.changed = 0; eloop.exceptions.changed = 0; eloop_process_pending_signals();
/* check if some registered timeouts have occurred */
timeout = dl_list_first(&eloop.timeout, struct eloop_timeout, list);

if (timeout) { os_get_reltime(&now);

if (!os_reltime_before(&now, &timeout- >time))
{
void *eloop_data = timeout->eloop_data;
void *user_data = timeout->user_data;
eloop_timeout_handler handler = timeout->handler;
eloop_remove_timeout(timeout);
handler(eloop_data, user_data); } }
if (res <= 0) continue; if (eloop.readers.changed || eloop.writers.changed || eloop.exceptions.changed)
{
/* * Sockets may have been closed and reopened with the
* same FD in the signal or timeout handlers, so we
* must skip the previous results and check again
* whether any of the currently registered sockets have
* events.
*/
continue;
}

#ifdef CONFIG_ELOOP_POLL
eloop_sock_table_dispatch(&eloop.readers, &eloop. writers,
&eloop.exceptions, eloop.pollfds_map,
eloop.max_pollfd_map);
#endif /* CONFIG_ELOOP_POLL */

/*
*Call the select system call to poll several types of events: whether the relevant fd is readable, writable, * whether an exception occurs, time out.
*If the relevant event occurs, eloop_sock_table_dispatch() will handle it as follows*, it will call the relevant handler
*/
#ifdef CONFIG_ELOOP_SELECT
eloop_sock_table_dispatch(&eloop.readers, rfds);
eloop_sock_table_dispatch(&eloop.writers, wfds); eloop_sock_table_dispatch(&eloop.exceptions, efds);
#endif /* CONFIG_ELOOP_SELECT */
#ifdef CONFIG_ELOOP_EPOLLeloop_sock_table_dispatch(eloop.epoll_events, res);
#endif /* CONFIG_ELOOP_EPOLL */
#ifdef CONFIG_ELOOP_KQUEUE
eloop_sock_table_dispatch(eloop.kqueue_events, res);
#endif /* CONFIG_ELOOP_KQUEUE */ } eloop.terminate = 0; out: #ifdef CONFIG_ELOOP_SELECT os_free(rfds); os_free(wfds); os_free(efds);
#endif /* CONFIG_ELOOP_SELECT */

return;
}
```

3. Registering the handler() callback function
Take the read event as an example:
Different types of event sources will write their own handler functions, such as udp, Unix general, etc.

In the end, the following interfaces will be called:
-->eloop_register_read_sock()
-->eloop_register_sock()
-->eloop_get_sock_table() //Different types return different tables
-->eloop_sock_table_add_sock() //Add it to table->table[i].handler

Through the above process, the handler() callback function is first registered to the table. When the relevant event occurs, the relevant handler() is called to handle the event.

4. Analysis of wpa_supplicant scanning process

In the previous section, we talked about registering a handler() callback function. The handler() registration interface for a read event is as follows:

eloop_register_read_sock(priv->sock, wpa_supplicant_ctrl_iface_receive,wpa_s, priv);
This interface registers a wpa_supplicant_ctrl_iface_receive() handler.

The main task of wpa_supplicant_ctrl_iface_receive() is to monitor the socket and process the command from the related socket.

Let's look at the source code of this function:

```
static void wpa_supplicant_ctrl_iface_receive(int sock, void *eloop_ctx,void *sock_ctx)
{

struct wpa_supplicant *wpa_s = eloop_ctx;
struct ctrl_iface_priv *priv = sock_ctx; char buf[4096];

int res; struct sockaddr_storage from; socklen_t fromlen = sizeof(from);
char *reply = NULL, *reply_buf = NULL;
size_t reply_len = 0;
int new_attached = 0; //recvform() listens to the relevant socket
res = recvfrom(sock, buf, sizeof(buf) - 1, 0, (struct sockaddr *)&from,&fromlen);

if (res < 0)
{
wpa_printf(MSG_ERROR,"recvfrom(ctrl_iface): %s",strerror(errno)) return;
}

buf[res] = '\0';
if (os_strcmp(buf, "ATTACH") == 0)
{
if (wpa_supplicant_ctrl_iface_attach(&priv->ctrl_dst, &from,fromlen, 0))
reply_len = 1 ;
else
{
new_attached = 1; reply_len = 2;
}
}
else if (os_strcmp(buf, "DETACH") == 0)
{
if(wpa_supplicant_ctrl_iface_detach(&priv->ctrl_dst, &from,fromlen)) reply_len = 1; else reply_len = 2;
}
else if (os_strncmp(buf, "LEVEL", 6) == 0)
{
if (wpa_supplicant_ctrl_iface_level(priv, &from, fromlen,buf + 6))
reply_len = 1;
else
reply_len = 2;
}
else
{
/*
*Submit the relevant commands to wpa_supplicant_ctrl_iface_process() * for processing.
*/
reply_buf = wpa_supplicant_ctrl_iface_process(wpa_s, buf,
&reply_len);
reply = reply_buf;

....
/*
*Feedback the relevant processing results to the corresponding socket.
*/ if (reply) {
wpas_ctrl_sock_debug("ctrl_sock-sendto", sock, reply, reply_len);
if (sendto(sock, reply, reply_len, 0, (struct sockaddr *) &from, fromlen) < 0) {
int _errno = errno;
wpa_dbg(wpa_s, MSG_DEBUG, "ctrl_iface sendto failed: %d - %s", _errno, strerror(_errno));

if (_errno == ENOBUFS || _errno == EAGAIN) {
 /* * The socket send buffer could be full. This * may happen if client programs are not * receiving their pending messages. Close and * reopen the socket as a workaround to avoid * getting stuck being unable to send any new * responses. */
sock = wpas_ctrl_iface_reinit(wpa_s, priv);
if (sock < 0) {
wpa_dbg(wpa_s, MSG_DEBUG, "Failed to reinitialize ctrl_iface
}
}

if (new_attached) {
wpa_dbg(wpa_s, MSG_DEBUG, "Failed to send response to ATTACH - detac new_attached = 0;
supplicant_ctrl_iface_detach( &priv->ctrl_dst, &from, fromlen);
}

}
}

os_free(reply_buf);
if (new_attached) eapol_sm_notify_ctrl_attached(wpa_s->eapol);

}
```

wpa_supplicant_ctrl_iface_process() - The wpa_supplicant_ctrl_iface_process() interface will determine the type of command, such as "SCAN"

```
...
else if(os_strcmp(buf,"SCAN") == 0)
if(!wpa_s->scan_ongoing){
wpa_s->scan_req = 2;
wpa_supplicant_req_scan(wpa_s,0,0);
}else
wpa_printf(MSG_DEBUG,"Ongoing Scan acti
...
```

wpa_ctrl_scan() processes the scan command
wpa_supplicant_req_scan arranges to scan nearby APs
wpa_supplicant calls the driver's ioctl interface through the '-D' parameter mentioned earlier. Here you can see some of the driver interfaces under wpa_supplicant

```
const struct wpa_driver_ops *const wpa_drivers[] = {

#ifdef CONFIG_DRIVER_NL80211
&wpa_driver_nl80211_ops,
#endif /* CONFIG_DRIVER_NL80211 */

#ifdef CONFIG_DRIVER_WEXT
&wpa_driver_wext_ops,
#endif /* CONFIG_DRIVER_WEXT */

#ifdef CONFIG_DRIVER_HOSTAP
& wpa_driver_hostap_ops,
#endif /* CONFIG_DRIVER_HOSTAP */

#ifdef CONFIG_DRIVER_BSD
&wpa_driver_bsd_ops,
#endif /* CONFIG_DRIVER_BSD */

#ifdef CONFIG_DRIVER_OPENBSD
&wpa_driver_openbsd_ops,
#endif /* CONFIG_DRIVER_OPENBSD */

#ifdef CONFIG_DRIVER_NDIS
&wpa_driver_ndis_ops,
#endif /* CONFIG_DRIVER_NDIS */

#ifdef CONFIG_DRIVER_WIRED
&wpa_driver_wired_ops,
#endif /* CONFIG_DRIVER_WIRED */

#ifdef CONFIG_DRIVER_MACSEC_QCA
&wpa_driver _macsec_qca_ops,
#endif /* CONFIG_DRIVER_MACSEC_QCA */

#ifdef CONFIG_DRIVER_ROBOSWITCH
&wpa_driver_roboswitch_ops,
#endif /* CONFIG_DRIVER_ROBOSWITCH */

#ifdef CONFIG_DRIVER_ATHEROS
&wpa_driver_atheros_ops,
#endif /* CONFIG_DRIVER_ATHEROS */

#ifdef CONFIG_DRIVER_NONE
&wpa_driver_none_ops,
#endif /* CONFIG_DRIVER_NONE */NULL
};

//Take wext as an example to see the related interfaces it supports

const struct wpa_driver_ops wpa_driver_wext_ops = {
.name = "wext", .desc = "Linux wireless extensions (generic)" ,
.get_bssid = wpa_driver_wext_get_bssid,
.get_ssid = wpa_driver_wext_get_ssid,
.set_key = wpa_driver_wext_set_key,
.set_countermeasures = wpa_driver_wext_set_countermeasures,
.scan2 = wpa_driver_wext_scan,
.get_scan_results2 = wpa_driver_wext_get_scan_results,
.deauthenticate = wpa_driver_wext_deauthenticate,
.associate = wpa_driver_wext_associate,
.init = wpa_driver_wext_init,
.deinit = wpa_driver_wext_deinit,
.add_pmkid = wpa_driver_wext_add_pmkid,
. remove_pmkid = wpa_driver_wext_remove_pmkid,
.flush_pmkid = wpa_driver_wext_flush_pmkid,
.get_capa = wpa_driver_wext_get_capa,
.set_operstate = wpa_driver_wext_set_operstate,
.get_radio_name = wext_get_radio_name,
.signal_poll = wpa_driver _wext_signal_poll,
.status = wpa_driver_wext_status,
};
```

wpa_supplicant_scan —> wpa_supplicant_trigger_scan requests the driver to scan

```
int wpa_supplicant_trigger_scan(struct wpa_supplicant *wpa_s, struct wpa_driver_scan_params *params)
{

struct wpa_driver_scan_params *ctx;

if (wpa_s->scan_work) {
wpa_dbg(wpa_s, MSG_IN FO, "Reject scan trigger since one is already pending "); return -1;
}

ctx = wpa_scan_clone_params(params);

if (!ctx || radio_add_work(wpa_s, 0, "scan", 0, wpas_trigger_scan_cb, ctx) < 0)
{
wpa_scan_free_params(ctx);
wpa_msg(wpa_s, MSG_INFO, WPA_EVENT_SCAN_FAILED "ret=-1");
return -1;
}
return 0;
}
radio_add_work–>wpas_trigger_scan_cb(callback function)–>wpa_drv_scan
```

Call the previously specified driver registered Function, for example, the driver specified by the -D parameter before, if wext is specified, then wpa_s->driver->scan2 will be called to call wpa_driver_wext_scan (this has been mentioned before, this structure wpa_driver_wext_ops will register this interface) through ioctl sends the scan command to the driver.

```
static inline int wpa_drv_scan(struct wpa_supplicant *wpa_s,struct wpa_driver_scan_params *params)
{
#ifdef CONFIG_TESTING_OPTIONSif (wpa_s->test_failure == WPAS_TEST_FAILURE_SCAN_TRIGGER)
return -EBUSY;
#endif /* CONFIG_TESTING_OPTIONS */
//Go back and select the driver registration Scan interface if (wpa_s->driver->scan2)
return wpa_s->driver->scan2(wpa_s->drv_priv, params);
return -1;
}
...
int wpa_driver_wext_scan(void *priv, struct wpa_driver_scan_params *params)
{
....//Omit irrelevant code//Send the scan request to the driver through the ioctl() interface, drv->ioctl_sock will be discussed below if (ioctl(drv->ioctl_sock, SIOCSIWSCAN, &iwr) < 0) {
wpa_printf(MSG_ERROR, "ioctl[SIOCSIWSCAN]: %s",
strerror(errno));
ret = -1;
}
....//Omit irrelevant code
}



//The ioctl() interface uses drv->ioctl_sock, which comes from wpa_driver_wext_init(). This function mainly performs operations on the wext driver interface.
void * wpa_driver_wext_init(void *ctx, const char *ifname)
{
struct wpa_driver_wext_data *drv;
struct netlink_config *cfg;
struct rfkill_config *rcfg;
char path[128];
struct stat buf;
drv = os_zalloc(sizeof(*drv));
if (drv == NULL)
return NULL;
drv->ctx = ctx;
os_strlcpy(drv->ifname, ifname, sizeof(drv->ifname));
os_snprintf(path, sizeof(path), "/sys/class/net/%s/phy80211", ifname);
if (stat(path, &buf) == 0) {
wpa_printf(MSG_DEBUG, "WEXT: cfg80211-based driver detected");
drv->cfg80211 = 1;
wext_get_phy_name(drv);
}
//We use drv->ioctl_sock, create and initialize it here
drv->ioctl_sock = socket(PF_INET, SOCK_DGRAM, 0);
if (drv->ioctl_sock < 0) {
wpa_printf(MSG_ERROR, "socket(PF_INET,SOCK_DGRAM): %s",strerror(errno));
goto err1;
}
cfg = os_zalloc(sizeof(*cfg));
if (cfg == NULL)
goto err1;
cfg->ctx = drv; cfg->newlink_cb = wpa_driver_wext_event_rtm_newlink; cfg->dellink_cb = wpa_driver_wext_event_rtm_dellink; drv->netlink = netlink_init(cfg); if (drv->netlink == NULL) { os_free(cfg); goto err2;} rcfg =
os_zalloc(sizeof( *rcfg)); if (rcfg == NULL) goto err3; rcfg->ctx = drv; os_strlcpy(rcfg->ifname, ifname, sizeof(rcfg->ifname)); rcfg->blocked_cb = wpa_driver_wext_rfkill_blocked; rcfg->unblocked_cb = wpa_driver_wext_rfkill_unblocked;
drv->rfkill = rfkill_init(rcfg); if (drv->rfkill == NULL) { wpa_printf(MSG_DEBUG, "WEXT: RFKILL status not available"); os_free(rcfg); } drv->mlme_sock = -1; if (wpa_driver_wext_finish_drv_init (drv) < 0) goto err3;
wpa_driver_wext_set_auth_param(drv, IW_AUTH_WPA_ENABLED, 1); return drv;
err3:
rfkill_deinit(drv->rfkill);
netlink_deinit(drv->netlink);
err2:
close(drv->ioctl_sock);
err1:
os_free(drv);
return NULL;
}
```

At this point, the work of wpa_supplicant has been completed. Next, this scan command will continue to be sent to the kernel and handed over to the driver for implementation.