Understanding WPA Supplicant

Chaitanya Ganoo

08-03-2009
Introduction

• **Q: What is WPA Supplicant?**
  - It is the IEEE 802.1X/WPA component in the client stations.

• **Q: What is wpa_supplicant?**
  - It is an *implementation* of the WPA Supplicant component from the HostAP project.
  - Runs on Linux, BSD, MAC OS X and Windows.
  - In nature, it is a *daemon* program and acts as the backend component controlling wireless connection (in the client).

• **Q: What are the alternatives to wpa_supplicant?**
  - XSupplicant from the Open1X project.
  - Microsoft Windows XP+ and Mac OS X have built-in pre-compiled supplicants.
  - Aegis (now CSSC- Cisco Secure Services Client) by Cisco.
  - Odyssey by Juniper networks.
  - SecureW2 by SecureW2
Introduction

The bigger picture:

- HostAP project aims to transform linux boxes into clients, Access Points and Authentication Servers supporting IEEE 802.11i

- Code written by Jouni Malinen.

  1. hostAP Linux driver for Conexant (formerly Intersil) Prism chipsets.

  2. wpa_supplicant runs on clients which can typically be laptop computers or embedded systems.

  3. Hostapd is a user space daemon for APs and Authentication Servers.
Functionality

Q: What does wpa_supplicant exactly do?

1. Requests kernel driver to scan neighboring BSSes.
2. Selects a BSS based on its configuration.
3. Request kernel driver to associate with the chosen BSS.
4. If WPA-PSK: Uses PSK as MSK.
5. If WPA-Enterprise: Integrated IEEE 802.1X Supplicant completes EAP authentication with the Authentication server (proxied by the Authenticator) and through this MSK is received form the IEEE 802.1X supplicant.
6. Now wpa_supplicant completes 4-Way Handshake and Group Key Handshake with the authenticator.
7. Configures encryption keys for unicast and broadcast.
8. Transmission and reception of normal data packets ensues.
Structure of the source code

- `wpa_supplicant.c`
  - `wpa.c`
- `driver_*.c`
  - `driver.h`
- `eapol_sm.c`
  - `eap.c`
  - `eap_*.c`
- `config.c`
- Generic helper functions
  - Most notably: `eloop.c`
- Cryptographic functions
Structure of the source code

- `wpa_supplicant.c` Program Initialization, main() and event handling.
- `eapol_sm.c` IEEE 802.1X/EAPOL processing and state machines.
- `eap.c` EAP SM
- `eap_*.c` EAP methods
- `driver_*.c` Driver dependent implementations.
- `driver.h` The API for driver interface.
- `config.c` Parser for command line.
- `eloop.c` Event loop (registerable timeouts + 2 types of callbacks)
- `common {.c , .h}` Common helper functions.
- `defs.h` Definitions shared by multiple files.
- Cryptographic functions and some other wrappers.
Drivers

- `wpa_supplicant` currently supports the following drivers:

  - `hostap` (default) Host AP driver (Intersil Prism2/2.5/3).
  - `hermes` Agere Systems Inc. driver (Hermes-I/Hermes-II).
  - `madwifi` MADWIFI 802.11 support (Atheros, etc.).
  - `atmel` ATMEL AT76C5XXx (USB, PCMCIA).
  - `wext` Linux wireless extensions (generic).
  - `ndiswrapper` Linux ndiswrapper.
  - `broadcom` Broadcom driver.
  - `ipw` Intel ipw2100/2200 driver.
  - `wired` `wpa_supplicant` wired Ethernet driver
  - `bsd` BSD 802.11 support (Atheros, etc.).
  - `ndis` Windows NDIS driver.
A cursory examination of main() reveals that:

1. Parses the command line.
2. Initializes event loop.
3. Calls the configuration file parser.
4. “Sets” the driver.
5. Initializes the control interface.

6. Initializes EAPOL state machine.
7. Initializes RSNA Supplicant Key Management.
8. Registers driver event handler.
9. Registers L2 receive handler.
10. Registers handlers for certain signals.

11. **Goes and sits in eloop_run()**

12. Also contains a cleanup block which is activated in case certain error conditions are satisfied.
Before we start analyzing `main()` its worth noting the following:

- `wpa_supplicant` uses a single process model and an event loop to provide callback on events.

- Thus modules interact by reading and writing state which is stored in shared global data structures.
  
  - `wpa_supplicant` is the most important data structure. It is defined in `wpa_supplicant_i.h`
  - `wpa_s` is an instance of this `wpa_supplicant` structure.
  - This is initialized in `main()` in `wpa_supplicant.c`

- We also note that as far as the event loop is concerned, 3 types of ‘events’ can trigger a callback:
  
  - Registered Timeouts
  - Received Packet
  - Signal
Data Structures

- `eloop.c` is an example of such an implementation of the event loop and uses `select()` and sockets.
  - `eloop_data` is the crucial data structure here. It is defined in `eloop.c`
  - `eloop` is an instance of this `eloop_data` structure.
  - 3 important structures contained within it:
    - `eloop_sock`. Instantiated as readers.
    - `eloop_timeout`. Instantiated as timeout.
    - `eloop_signal`. Instantiated as signals.
  - Also contains a counter to maintain a count on the number of readers as well as the number of signals.
Analyzing main() 1-4

• main():
  
  – Infinite ‘for’ loop for parsing command line (BcDdehiLqvw)
  
  – eloop_init(&wpa_s):
    
    Initializes eloop to zero and after that populates the user_data field with the contents referenced by wpa_s.
  
  – Call config file parser: read_lradxt_config():
    
    This has been defined in driver_lradxt.c and by default reads the ‘/usr/local/lrdxt/lrdxt_config.new’ file.
    Parameters scanned by the above parser are:
    
    • WLAN SSID
    • SEC MODE
    • SEC_RSN_PARAMS
    • MS_BASE_MAC
    • AP_BASE_MAC
    • AP HOST DIST LIST
  
  – wpa_supplicant_set_driver(&wpa_s,driver)
    
    • The driver field in in wpa_s is populated with the address of wpa_driver_*_ops.
    • (in our case: wpa_driver_lradxt_ops)
  
• Note: **wpa_driver_lradxt_ops** is an instance of **struct wpa_driver_ops**. This wpa_driver_ops structure defines the API that each driver interface needs to implement for core wpa_supplicant code. All driver specific functionality is captured in this wrapper.
Analyzing main()

- wpa_supplicant_ctrl_iface_init(&wpa_s)
  
  - This init call effectively creates a datagram socket of the PF_UNIX domain called ‘s’.
  
  - This is then assigned to the control interface socket in wpa_s.
  
  - Lastly this is registered with the event loop through a eloop_register_read_sock() call.

- The use of this is that now wpa_ctrl.c can be used as small C library by external programs like wpa_cli

- Some typical control interface commands are:
  
  - PING
  - MIB
  - STATUS

Note:

wpa_ctrl_request () defined in wpa_ctrl.c is mostly used to send commands to the wpa_supplicant. wpa_s has a structure wpa_config which contains a field ‘ctrl_iface’. This is a directory for UNIX domain sockets.
Quick look at EAP

• **Q: What is EAP?**
  Extensible Authentication Protocol (EAP) is an authentication framework defined in RFC 3748.

• **Q: How is it related to wpa_supplicant?**
  wpa_supplicant uses a separate and independent code module for EAP peer implementation.

• **Q: What is the relationship of 802.1X with EAP?**
  EAP was developed for dial-up connections. Hence no link-layer protocols.
  802.1X provides a set of context (such as port and supplicant), state machines between the various layers, and the EAP over LAN (EAPOL) protocol.

• **Q: What is EAPOL?**
  EAPOL is a standard for passing EAP over a wired or wireless LAN. This is referred to as the ‘lower layer’ in the EAP Peer implementation.

• 2 RFCs are referred to:
  
  – RFC 2284: PPP EAP
  – RFC 4137: State Machines for EAP Peer and Authenticator
4. Peer State Machine

This follows is a diagram of the EAP peer state machine. Also included is an explanation of the primitives and procedures referenced in the diagram, as well as a clarification of notation.

Figure 3: EAP Peer State Machine
EAP Peer SM

Dissecting the EAP Peer State Machine (SM):

• Basic 4 (+2) EAP Types as mandated by section 3.0 of RFC 2284.
• EapType enum defined in eap.h actually has 9 possible values to account for new and proprietary EAP types like LEAP, SIM etc.

• 13 possible states in the EAP Peer SM according to Fig. 3 in RFC 4137.
  ✓ This is reflected in the EAP_state enum within the structure eap_sm in eap.h

• methodState is an important bidirectional interface between Peer SM and Methods.
  ✓ It has 5 possible values and is implemented as an enum within eap_sm in eap.h

✓ Other important short term and long term local variables as indicated in RFC 4137 are defined in either eap.h or eapol_sm.h
Analyzing main()

(If EAPOL is NOT disabled)

- `eapol_sm_init(ctx)`
  - `wpa_s` contains the EAPOL state machine **structure called as eapol_sm** within it.
  - `eapol` is an instance of this `eapol_sm` structure defined in `wpa_supplicant_i.h`.
  - `sm` is an instance of `eapol_sm` structure as defined in the `eapol_sm_init` definition in `eapol_sm.c`.
    - EAP state machine is initialized through `eap_sm_init(sm)`.
    - The `eap_sm` structure which is returned is put into `sm->eap`.
    - EAPOL state machine is initialized through the `eapol_sm_step(sm)`.

- `SM_STEP_RUN (SUPP_PAE)`
- `SM_STEP_RUN (KEY_RX)`
- `SM_STEP_RUN (SUPP_BAE)`

  - The above macros expand to a function call to the EAPOL state machine step function.
  - Refer to `eapol_sm.c` and RFC 4137.

- RSNA Supplicant key Mgmt: contain calls to `eapol_sm_step()` which registers timeouts.
Driver details

- All hardware and driver dependent functionality is in separate C files that implement defined wrapper functions.

- Driver wrappers need to implement whatever calls are used in the target operating system/driver for controlling wireless LAN devices.

- **A driver wrapper needs to implement some or all of the functions defined in driver.h**

- These functions are registered by filling struct `wpa_driver_ops` with function pointers.

- Hardware independent parts of wpa_supplicant will call these functions to control the driver/wlan card.
Driver details

- These functions are as follows: (refer to driver_lradxt.c for details)
  - get_bssid: Query the kernel driver for the current BSSID
  - get_ssid
  - set_wpa: Enable/disable wpa support
  - set_key: Configure encryption key using either NONE/WEP/TKIP/CCMP
  - events_init: Initialize processing of driver events
  - events_deinit
  - set_countermeasures: enable/disable TKIP countermeasures
  - set_drop_unencrypted: enable/disable unencrypted frame filtering
  - scan: Request the driver to initiate a scan
  - get_scan_results
  - deauthenticate: call to wpa_driver_lradxt_mlme
  - disassociate: call to wpa_driver_lradxt_mlme
  - associate: Set WPA support, frequency, SSID/BSSID and call to lradxt_ioctl
  - set_auth_alg: Open, Shared Key or LEAP

- In addition, support for driver events is required.

- The event callback function, wpa_suppliant_event(), and its parameters are documented in driver.h.
The events_init () call:

- Initializes event processing for kernel driver events like association, scan results, MICHAEL MIC failure etc.

- Though this we reach wpa_driver_lradxt_events_init() wherein we open a raw socket of the PF_NETLINK domain and NETLINK_ROUTE protocol.

- This socket is then registered with the event loop through eloop_register_read_socket() using the wpa_driver_lradxt_event_receive handler.

- Within wpa_driver_lradxt_event_receive() we do a couple of things:
  
  - Listen on the socket using recvfrom()
  - Examine the netlink message header and switch to RTM_NETLINK or RTM_DELLINK
  - Now we use wext and in particular wpa_driver_wext_event_wireless()
  - This contains a switch case on iwe.
  - iwe is an instance of iw_event which is a standard struct defined for wireless events in Linux.
  - The cases of SIOCGIWAP, SIOCGIWSCAN etc are covered.

wpa_supplicant_driver_init():

- Calls l2_packet_init() which registers the L2 receive handler.
- Also clears WPA keys, disables countermeasures etc.
One possible chain within `wpa_driver_lradxt_event_receive` is:

```
1. wpa_driver_lradxt_event_receive
2. wpa_driver_wext_event_rtm_newlink
3. wpa_driver_wext_event_wireless
4. wpa_supplicant_event
5. wpa_supplicant_select_config
6. set_wpa_values_for_this_mobile
```
Analyzing main() 10

• eloop_register_signal()
  – 2 with wpa_supplicant_terminate as function pointer argument
  – 1 with wpa_supplicant_reconfig
  – 1 with print_eap_status.
  – What happens effectively in this call is that the ‘signalled’ field of eloop as well as that of the particular signal is incremented
  – A signal() system call is made with the passed handler

• eloop_register_timeout()
  – call to print EAP statistics.
Finally we get to eloop_run():

As mentioned before, the event loop keeps waiting in a while loop until either:

- A particular condition (e.g. eloop.terminate==0) occurs.
- Timeouts.
- Signal is raised.

rfds is our FD_SET.

We initially clear it using FD_ZERO.

Now we add the file descriptor ‘elope.readers[i].sock’ to rfds. Here i loops over the number of
readers present i.e. from 0 to eloop.reader_count.

After this a call to select() is made with the address of rfds which contains sockets that we are
waiting to become readable.

select() returns the number of sockets that had ‘become active’ in the form of an integer.

In the concluding for loop we use the macro FD_ISSET to test which socket of which reader has
become active and also call the appropriate handler.
Questions