P33 Building an Internet Router - Documentation Part 1

C. T. Bashford-Chuchla (ctb33)

Contents

1 Code Flow

Packets enter my code in the router_handle_packet (packet_info_t *pi) function. Of which packets seperate into two directions, one for ARP packets and the other for IP packets.

1.1 ARP Packets

All incoming ARP packets are sent through to handle_ARP_packet(packet_info_t *pi). All functions that deal with ARP packets are contained in the *arp.c* file.

1. handle_ARP_packet(packet_info_t *pi)
   This function deals with the processing of the incoming ARP packets. It is split into two, first part for requests and the second for replies. The request part first stores the mac address of the sender in the arp table, and replies by modifying the packet (changes the OP code to 2, swapping the sender

and target details, and sets the sender mac address). The reply part saves the sender mac address in the ARP table and then checks if there are any pending packets that need the ARP information just received (see section 2.2 in Design Decisions) and sends them if necessary.

2. handle_not_repsponding_to_arp (byte *payload, unsigned len)
   This function generates a not responding to arp packet with the given packet. Most of this function is achieved by calling generate_response_ICMP_packet(packet_info_t *pi, int type, int code) where type is 3 and code is 1.

3. generate_pending_ARP_thread()
   This function is constantly being run in the ARP thread (see section 2.2 in Design Decisions). It has two main functions, first of which is to send ARP packets that are in the pending_arp array and secondly to send not responding to arp packets(see above) when there is no response after 5 arp requests sent.

4. send_ARP_request(addr_ip_t ip, int num)
   This function sends an ARP request to the specified ip address parameter. It is used in the ARP thread by the generate_pending_ARP_thread() function.

5. router_add_arp_entry (router_t *router, addr_mac_t mac, addr_ip_t ip, bool dynamic)
   This function adds the arp entry given into the arp table. It also sends the same ARP entry to the hardware, if running in hardware mode.

6. router_delete_arp_entry (router_t *router, addr_ip_t ip)
   This funciton deletes an arp entry specified by the ip parameter. After deleteing the entry, it also shifts all entries after it up by one, so there are no gaps in the table. This is all reflected in the hardware ARP table, by sending the same ARP entries to the hardware.

7. router_delete_all_arp_entries (router_t *router, bool dynamic)
   This function deletes all ARP entries that are specified by the dynamic boolean parameter. This is achieved by deleting the entries and shifting succeeding entries in one loop through the arp entries. The hardware ARP table is updated by sending all the ARP entries to the hardware at the end.

8. router_find_arp_entry (router_t *router, addr_ip_t ip)
   This function finds the ARP entry in the ARP table specified by the ip parameter. The function returns a pointer to the ip, mac pair if found, otherwise returns NULL.

1.2 IP Packets

All incoming IP packets are sent through to handle_IPv4_packet(packet_info_t *pi). All ICMP packets are sent through to handle_ICMP_packet(packet_info_t *pi) (see section 1.2.1), TCP packets to handle_TCP_packe(packet_info_t *pi) (contained in *ip.c*) and PWOSPF packets to handle_PWOSPF_packet(packet_info_t *pi). All functions that deal with IP packets are contained in the *ip.c* file.

1. handle_IPv4_packet(packet_info_t *pi)
   This function is called with all incoming IP packets. It first of all checks that there are no options in the packet and then checks the checksum. If the packet has a destination of one the interfaces or the OSPF IP address then it is dealt by the router directly using the functions specified above, otherwise the TTL is decreased by one (and recompute checksum) and sends the packet towards its destination.

2. handle_TCP_packet(packet_info_t *pi)
   All TCP packets that are intended for one of the router's interfaces are called with this function. The

ethernet header is stripped off and the rest of the packet is sent to the sr_transport_input (uint8_t * packet) function.

3. calc_checksum(byte *header, int len)
   This function just computes the checksum of the header specified with the length given.

4. add_IPv4_header(uint8_t * payload, uint8_t proto, uint32_t src, uint32_t dest, int len)
   Ths function takes a packet and puts a IPv4 header on it with the specified protocol, source and destination IP. A new packet is malloc'ed with 20 bytes longer length and the function returns the new pointer to the packet.

5. handle_no_route_to_host (packet_info_t *pi)
   This function is called when there is no route to the host in the routing table. A ICMP packet is sent to the source specifying this. Most of this function is achieved by calling generate_response_ICMP_packet(packet_info_t *pi, int type, int code) where type is 3 and code is 0.

1.2.1 ICMP Packets

All incoming ICMP packets are sent through to handle_ICMP_packet(packet_info_t *pi). All functions that deal with ICMP packets are contained in the *my_icmp.c* file.

1. handle_ICMP_packet(packet_info_t *pi)
   This function is called with all incoming ICMP packets. The checksum for the ICMP packet is calculated first of all. Then if the packet is an echo request, the type is changed to 0 and the checksum is recalculated. Otherwise, if the packet is a echo reply then handle_ping_reply (packet_info_t *pi) is called (see below). All echo request packets (after type is set to 0) are sent in the tail part of the handle_IPv4_packet (packet_info_t *pi) function after this packet returns.

2. generate_response_ICMP_packet(packet_info_t *pi, int type, int code)
   This function creates a ICMP packet with IPv4 header with the specified type and code. The function creates a new packet and sets it in the packet_info_t structure. The destination is computed by the source of the inputted packet.

3. void send_ping(router_t *router, addr_ip_t dest_ip, addr_ip_t src_ip, uint16_t id, uint16_t count)
   This function sends a ping to the specified destination IP address. This function is called only by cli_ping_request (router_t * rtr, int fd, addr_ip_t ip) in —cli_ping.c. A IPv4 is added to the ping packet in this function and is sent to the correct interface.

4. handle_ping_reply (packet_info_t *pi)
   This function deals with an incoming ping reply packet. This is called by handle_ICMP_packet(packet_info_t *pi). The packet is sent to the cli via the function cli_ping_handle_reply (addr_ip_t ip, uint16_t seq).

1.2.2 PWOSPF Packets

All incoming PWOSPF packets are sent through to handle_PWOSPF_packet(packet_info_t *pi). All functions that deal with PWOSPF packets are contained in the *routing.c* file. This includes computing the routing table.

1. handle_PWOSPF_packet(packet_info_t *pi)
   This function is called with all incoming PWOSPF packets. First of all, there is a check to see when the router is using the OSPF protocol currently. If it is not then all packets are dropped. If the OSPF

protocol is running then the version value is checked that it is 2, the authentication is checked that it is 0, the checksum is checked and finally that the area id in the packet is the same as the router's area id. If any of these checks fail then the packet is dropped. The packet goes into two paths, one for HELLO packets and the other for Link State Update packets. For the HELLO packets, the subnet_mask and helloint are checked against the interface the packet came in on. If there is a mismatch the packet is dropped. The neighbours are added to the neighbor linked list, if they are not already on it. Otherwise, the last time recieved counter in the corresponding neighbor entry is updated and so is the link state update for the neighbor in the link state database for the current router. Lastly, if a new neighbor was added to the linked list, then a link state is added to the database for the current router. Now, if the packet is a Link State Update, the router id is checked that is for this router. If there is no database entry for this router, then one is created with the specified link state adverts and the routing table is recomputed. Otherwise, the sequence number is checked that it is different to the previous and that the contents is different to the last link state update recieved from this router. If either check fails then the packet is dropped. First of all the link state adverts are compared with the current link state adverts in the database for the source router. If there is no change then the last time recieved counter are updated and it finishes. If there is a change in the link states adverts in the database or the link state packet has come from a router not in the database then the new advertisements link states are added (with replacement) and the routing table is updated by calling update_routing_table (). Finally, if the packet hasn't been dropped, the packet's TTL is decreased by one (with recomputing the checksum) and forwarded to all neighbors.

2. print_database ()
   Ths function prints out the database of stored link state updates for all routes.

3. update_routing_table ()
   This function first finds all subnets in the link state database and calculates a distance to each subnet. For all reachable subnets a routing entry is added. This is explained in more detail in the Design Decision section 2.3.

4. send_HELLO_packet(interface_t *intf)
   This function is used to send HELLO packets to the specified interface. It computes the desired packet by the interface information and sends it if there is a neighbor connected.

5. send_LSU_packet(unsigned seq_no)
   This function is used to send link state update packets with the specified sequence number to interfaces with routers as neighbors. The function computes the link states adverts from the neighbors on each interface and puts them into a packet to be forwarded to each interface.

6. generate_HELLO_thread()
   This function is constantly being run in the HELLO thread (see section 2.3 in Design Decisions). It has three main functions, first of which is to send HELLO packets to each connected neighbor every helloint seconds, secondly to check for neighbor timeouts (and if there is to update the routing table and send a new link state update packet to all interfaces) and lastly to send link state update packets to all interfaces every lsuint seconds or if there a link state change has been detected.

7. router_add_route (router_t * router , addr_ip_t prefix , addr_ip_t next_hop , addr_ip_t subnet_mask, const char *intf_name , bool dynamic)
   This function adds the specified route to the routing table in the correct place (sorted by dynamic and then network mask), it does this by shifting the succeeding routes down by one. The same changes are also reflected in the hardware routing table.

8. router_find_route_entry (router_t *router , addr_ip_t dest , addr_ip_t gw, addr_ip_t mask, const char *intf_name)
   This function finds the entry in the routing table specified by the parameters. The function returns a pointer to the routing entry if found, otherwise returns NULL.

9. router_delete_route_entry (router_t *router , addr_ip_t dest , addr_ip_t gw, addr_ip_t mask, const char *intf_name)
   This function deletes the route in the routing table specified by the parameters. The function returns whether it is successful in finding a matching entry. All changes are reflected in the hardware routing table.

10. router_delete_all_route_entries (router_t *router , bool dynamic)
    This function deletes all routes in the routing table that matches the dynamic boolean parameter. All changes are reflected in the hardware routing table.

11. sr_read_routes_from_file (router_t * router , const char* filename)
    This function reads in the routing file and adds all routes as static in the routing table.

12. database_find_link (database_entry_t *database_entry , uint32_t router_id , uint32_t subnet_no)
    This function finds the link in the database entry specified by the parameters. The function returns a pointer to the link if found, otherwise returns NULL.

13. router_add_link_to_database_entry (router_t *router , database_entry_t *database_entry , link_t *link_to_add)
    This function adds the link to the database entry specified by the database_entry and link_to_add parameters respectively. If the link already exists in the database, then the last time is updated.

14. router_remove_link_from_database_entry (router_t *router , database_entry_t *database_entry , uint32_t router_id)
    This function removes the link in the database entry specified by the database_entry and router_id parameters respectively. The function returns whether the link was found in the database.

15. router_add_database_entry (router_t * router , uint32_t router_id , link_t link [], unsigned num_links, uint16_t seq_no , byte *packet, unsigned len)
    This function adds all the links in the specified link array to the database entry specified by the router id. If the links exceed 10 then an error is thrown. Each link is copied and thus the array parameter will not need to be kept in memory.

16. router_find_database_entry_position (router_t *router , uint32_t router_id)
    This function find the position of the database entry specified by the router id. This will return the integer in which the router id entry is in the database array.

1.3 Router

This section reflects the changes in *sr_router.c*. The main purpose of the functions added was to send packets and to setup the hardware interface table.

1. send_packet(byte *payload, uint32_t src , uint32_t dest , int len , bool is_arp_packet , bool is_hello_packet)
   This function sends a packet specified by payload to the destination specified. It first checks the destination is one of the interfaces or the OSPF IP address. If it is not, a no route to host packet is generated. Once passing this test, the packet is sent to the found interface by send_packet_intf (...) (see below).

2. send_packet_intf (interface_t *intf , byte *payload, uint32_t src , uint32_t dest , int len , bool is_arp_packet , bool is_hello_packet)
   This function sends the specified packet on the interface specified by the argument intf. The ethernet packet is added to the packet, in doing so it checks for an ARP entry, and sends the packet. If there is no ARP entry for the next hop, then the packet is queued and an ARP request is sent.

3. setup_interface_registers (router_t * router , int intf_num)
   This function sends the interface mac address of the specified interface to the hardware interface table.

2 Design Decisions

2.1 Packets

In most cases when responding to a packet, the code will modify the incoming packet in order to responding to the packet. An exception to this rule is when generating response ICMP packets, as this will often need to be longer than the incoming packet. The decision was chosen because meant less memory need to be created and that less packet fields would need to be generated as they can be reused. This also means, that a lot of the code has a tail. In other words, in the handle_IPv4_packet () function the packet is sent in the tail of the function after the handle_ICMP_packet() finishes.

2.2 ARP Protocol

In order to deal with pending ARP requests, a seperate thread is used. The thread constantly runs the generate_pending_ARP_thread() function. The approach uses the thread to check for packets in the pending ARP queue and send up to five ARP requests if necessary, and more over generate a not responding to ARP request response. This was chosen over starting a seperate thread for each pending ARP request, as running too many threads caused some threads to not get enough CPU time to send the packets in a timely fashion. In order to keep the threads safe, locks were used when ever modifying the pending arp table.

2.3 Routing Protocol

2.3.1 Data Structures

The routing protcol required two data structures to be used, the neighbors from which a HELLO packet are recieved and the link state advertisments recieved. For the neighbors, a linked list on each interface is used, where all neighbors that send HELLO packets are recorded. Note that interfaces are also added to the neighbor linked list, this helps when calculating the link state packets, as it can just generate a link state advertisement for each neighbor in the neighbor linked lists on each interface. For the link state advertisements an array on the router_t struct is used. The array contains an entry for each router that sent link state advertisements, in each entry of the database array, there is a subarray containing an array of advertised links for the router. Thus, effectively a two dimension array, where the first dimension is number of routers in the topology and the second dimension is the number of links of each router. Note the link state advertisement database contains an entry for the router that the code is running on (i.e. the current router), this makes running Dijkstra's algorithm easier. These links on the database entry for current router are added for each interface and one for each neighbor that advertises a HELLO packet.

2.3.2 Computing Routing Table

This section describes how the code computes the routing table using the link state advertisements in the database. First of all, all distinct subnets are found by looking through the database of link state updates for unique subnets. These are what we are trying to find a route for, especially the interface in which we use to reach the subnet. To store what interface the route should use, the first router corresponding to the route is recorded. In the algorithm, the second part computes the distance to all routers in the database, by use of Dijkstra's algorithm. The first router and distance for the route is recorded for each router in the database. The algorithm is an update version, where a best distance is updated at each iteration of the database. Thirdly, the distance to each router is used to compute the distance to each unique subnet computed in the first step. The effectively gives us the route with the lowest distance and the correponding first router is used to identify the correct interface. Now, the routes can be added to the database, this part of the algorithm uses the first routers for each subnet to find the correct neighbor for the router, and thus adds a route with the neighbor IP address as the next hop (gateway).

3 Advanced Feature: Security Gateway

3.1 Overview

My advanced feature builds upon the basic working router by adding tunnelling and security functionality. There are three stages of the tunnelling and security feature where each stage builds upon the previous. Firstly, there is IP in IP tunnelling, this involves adding an extra IP header to all packets that have a matching policy. Secondly, we have authentication via a simplified version of the ESP header with null encryption. Lastly, we include simple encryption in the ESP header. This is all configurable by adding policies to the router, which signify whether a packet should have security included and the level of security. Policies can be added via the command line interface of the router.

3.2 Hardware and Software

All the code needed to implement this advanced feature is done on the software side. All the security features on the packets are added and removed in the software. Additional entries in the hardware routing table are needed to send packets with a matching policy to the CPU. These additional entries are added when a policy is added; the route range matches the destination range of the policy and the output queue is one of the interfaces id (for simplicity sake the first interface is chosen).

3.3 Policies

The policies are used to signify whether a packet should have security features added on. Each policy requires two ranges of IP addresses, one for the source and one for the destination. These ranges are implemented using a prefix and mask. The fields tell the router that packets need security features when the packet comes from an address that is in the source range and is going to an address that is in the destination range. Each policy also includes two fields for the local endpoint address and remote endpoint address. These fields signify where the security link starts and end. Thus, security is only added to packets at a router when one of the router's interfaces matches the local endpoint address and is removed and/or checked when one of the router's interfaces matches the remote link address.

Furthermore, there are 2 extra fields that signify what level of security is needed. The first is the authentication secret and the second is the encryption rotation number. The authentication secret is what is used to add and check authentication on the packet. The encryption rotation number signifies how much shifting takes place in the encryption. If the authentication string and encryption are empty then only IP tunnelling takes place, otherwise depending on whether each field is non-empty then that security feature is used.

For functioning secure communication the same policy needs to be added to the local router and the remote router. Furthermore, a policy only signifies a one directional link, for a bidirectional link two policies are needed on both routers. Policies are stored in the router object as an array of policies.

3.3.1 Command Line Interface

The command line interface has been expanded to allow the addition and deletion of policies to the router. These are done via the "ip policy add" and "ip policy del" commands. There is also a command to remove all policies via "ip policy purge". The add command requires parameters specifying the source and destination range and the endpoints, as well as the optional authentication or encryption values. The source and destination range can be entered either via two separate IP addresses (the prefix address and mask) or via the CIDR notation. The delete command has the same required parameters as the add command.

Examples:
policy add 10.0.1.2/32 10.0.5.2/32 10.0.1.1 10.0.5.1 secretphrase 13
policy del 10.0.1.12 255.255.255.255 10.0.5.2 255.255.255.255 10.0.1.1 10.0.5.1

3.4 IP to IP Tunnelling

IP tunnelling takes place when a matching policy is found for a packet as describes above. It involves adding an extra IP header inbetween the Ethernet header and the original IP header. This encapsulates the original IP packet, some details of this can be seen at RFC 1853. There are a few fields of the extra IP header to note. First of all, the source and destination fields are set to the local and remote tunnel endpoint addresses respectively of the matching policy. The protocol field is the other important field. It signifies whether it encapsulates an IP or ESP packet by use of the encapsulation or ESP protocol number (see next section for more details). It is also worth noting that packets that need to be encapsulated and then exceed the MTU value will not be sent successfully. This is a place for improvement, by possibly fragmenting the packet or sending and ICMP packet to the originator.

3.5 ESP Protocol

The ESP protocol is used when there is a matching policy for a packet that has a non-empty field for the authentication secret or encryption rotation number. Note that both of these fields can be non-empty or just one for the ESP protocol to be used. The full details for the ESP protocol can be seen at RFC 4303. I have simplified the ESP header to make the implementation easier. I have not made use of the security parameters index field nor the padding fields. I have not needed the security parameters index because I assume that there is only one matching policy for each tunnel (i.e. there cannot be two policies with matching local and the remote points). Therefore, I lookup the policy via the source and destination address fields of the encapsulated IP header. This saves having the routers having to agree on the SPI values. Furthermore, I have not used the padding fields for simplicity.

3.5.1 Authentication

Authentication is done via the integrity check value. At the originating end (when the ESP header is added), the authentication value is computed from the packet (minus the ICV field) and the authentication secret, and is set in the ICV field. Then at the remote endpoint (when the ESP header is removed), the authentication value is computed in the same way and then checked against the ICV field value. Packets are dropped when the the computed authentication value doesn't match the ICV value.

The algorithm used for the authentication is SHA-256 and was imported from http://svnweb.freebsd.org/base/head/sys/crypto/sha2/.

3.5.2 Encryption

Encryption is done by Caesar Cipher-like algorithm. The algorithm is purely for demonstration purposes and a more secure encryption algorithm should be used in practice, but should not require a lot of extra code. At the local endpoint (when the ESP header is added), the encryption is done before authentication, whereas at the remote link end this is done in reverse, authentication check before decryption. The encryption shifts each byte in the packet (minus the ESP header and ICV field) by the rotation number and mods by 256.

3.6 Code Flow

There are two main additions made to the basic working router code flow in the handle_IPv4_packet(packet_info_t *pi) function that makes the advanced feature possible. The first is one for use at the local link end when adding the extra IP and ESP header and the other is for use at the remote end when handling packets with encapsulation.

The first addition to the code checks if the packet matches a sending policy (i.e. if encapsulation is needed). If a sending policy is found then an IP encapsulation header is added (via the add_IPv4_header() function) and then an optional ESP header with tail (via the add_ESP_packet() function) depending on the matching policy. The second addition is adding an additional case to the protocol switch, so that packets with either IP encapsulation and ESP protocol are sent to the handle_IP_ENCAP_packet(packet_info_t *pi) function.

There is also an additional parameter added to the add_IPv4_header() function called offset allowing the IP header to be added at any offset of the packet.

The rest of the additional code is within the *policy.c* file.

1. handle_IP_ENCAP_packet(packet_info_t *pi)
   This function is called with all incoming IP encapsulation packets. The function first checks if there is a matching receiving policy, otherwise just exits. If there is a matching receiving policy then the IP encapsulation header is removed and then the function checks that the policy matches the type of packet (i.e. IP encapsulation packet only when neither authentication secret nor encryption number found and IP encapsulation and ESP header when there is a authentication string and/or encryption number found in the policy). Next the function checks the authentication value and decrypts the packet (when appropriate). The ESP header is also removed if appropriate at the end.
2. add_ESP_packet(uint8_t *payload, unsigned offset , uint32_t spi , uint32_t seq_no, uint8_t pad_len , uint8_t next_hdr , char *secret , uint8_t encrypt_rot , int len)

This function takes a packet and puts an ESP header and tail on it at the specified offset with the specified SPI, sequence number, padding length, next header protocol, authentication secret and encryption rotation number. A new packet is malloced with 26 bytes longer length and the function returns the new pointer to the packet. Note SPI is set to 0 on all calls as it is not used in my implementation.

3. calc_sha256(uint8_t answer[16], uint8_t *payload, unsigned offset , unsigned len , char * secret)
   This function calculates the authentication value for the packet at the offest with the authentication secret. The result is set in the answer array.

4. copy_policy (policy_t *policy)
   This is a helper function to copy a policy (with secret string). The copied policy is returned.

5. free_policy (policy_t *policy)
   This is a helper function to free policies (with their secret string).

6. router_find_matching_policy_sending (router_t * router , addr_ip_t matching_src_ip , addr_ip_t matching_dest_ip)
   This function is called to check if there is a matching policy when sending a packet. A policy is found if the policy remote address is on one of the routers interfaces and the local and remote link end addresses match. A copy of the policy is returned.

7. router_find_matching_policy_receiving (router_t * router , /* addr_ip_t matching_src_ip , addr_ip_t matching_dest_ip ,*/ addr_ip_t matching_local_end , addr_ip_t matching_remote_end)
   This function is called to check if there is a matching policy when receiving a packet. A policy is found if the source IP address is in the policy source range, the destination IP address is in the policy destination range and the policy local endpoint address is on one of the routers interfaces. A copy of the policy is returned.

8. router_add_policy (router_t * router , addr_ip_t src_ip , addr_ip_t src_mask, addr_ip_t dest_ip , addr_ip_t dest_mask , addr_ip_t local_end , addr_ip_t remote_end, const char * secret , uint8_t encrypt_rot , uint32_t spi)
   This function is called from the command line interface when a policy is added. The function first adds the policy to the policy table and then a routing entry is added to the hardware routing table (see section 3.2 for more details). This is done by shifting all routes in the hardware routing table down by one and inserting it at the correct place.

9. router_delete_policy_entry (router_t *router , addr_ip_t src_ip , addr_ip_t src_mask, addr_ip_t dest_ip , addr_ip_t dest_mask, addr_ip_t local_end , addr_ip_t remote_end)
   This function is called from the command line interface when a policy is deleted. The function first deletes the policy from the policy table and then the corresponding routing entry for the policy is removed from the hardware routing table (see section 3.2 for more details). This is done by shifting all routes in the hardware routing table up by one after the deleted routing entry.

10. router_delete_all_policy (router_t *router)
    This function removes all policies from the policy array. It then removes all routing entries corresponding to a policy from the hardware table.