Linux IPsec Tutorial

Sowmini Varadhan (Oracle) & Paul Wouters (Redhat)

Netdev0x12, July 2018, Montreal, Canada

Agenda

- Background: brief introduction to IPsec and IKE terminology
- IPsec datapath walk-through: trace the life of a UDP packet for the transmit and receive path as it passes through the Linux kernel's network stack (Sowmini Varadhan)
- IPsec control plane walk-through: everything you wanted to know about the IKE control plane (Paul Wouters)

First, a review of IPsec/IKE terminology/definitions

What is IPsec?

- IP Security
- Suite of protocols for encryption (adding a “ESP” header) and Authentication (adding a “AUTH” header)
- Encryption parameters (e.g., key, algorithm) are determined from two databases:
  - Security Policy database (SPD)
  - Security Association database (SADB)

SPD and SADB

- SPD: Security Policy Database
  - **What** must be done: e.g., “for packets in 13.0.0.0/24, perform IPsec ESP processing”, “discard packets in 192.168.0.0/16”
  - Multiple transforms may be specified, e.g., “for packets from 12.0.0.1→12.0.0.2, apply ESP, then apply compression”
  - May need to create/lookup a Security Association to perform the action defined the SPD entry
- SADB: Security Association Database
  - **How** to apply the security transform(s): e.g., “for packets from 13.0.0.1 → 13.0.0.2, apply AES-GCM-256 with the key *0x1234..* and a 128 bit IV”
  - IKE (Internet Key Exchange) protocol for negotiating and establishing SADB parameters for the IPsec association

Example of SPD and SADB entries

- Policy for “Encrypt all UDP packets to 13.0.0.0/24 using ESP”

```
# ip x p   # to display SPD using /sbin/ip
  src 13.0.0.8/32 dst 13.0.0.0/24 proto udp
        dir out priority 2024 ptype main
        tmpl src 0.0.0.0 dst 0.0.0.0
                proto esp reqid 0 mode transport
```

- SA for 13.0.0.8 → 13.0.0.9 “acquired” from that Policy

```
# ip x s # to display SADB entries
  src 13.0.0.8 dst 13.0.0.9
        proto esp spi 0x9de792fc reqid 16397 mode transport
        replay-window 32
        aead rfc4106(gcm(aes)) 0x9831b3b3b7c7.. 64
        anti-replay context: seq 0x0, oseq 0x1, bitmap 0x0
        sel src 13.0.0.8/32 dst 13.0.0.9/32 proto udp
```

- We will come back to this example as we trace our packet path...

IPsec Transport vs Tunnel mode

- IPsec Transport mode: ESP/AH transforms apply to L4 (TCP or UDP) header and payload.
  - Protects L4 header
  - L3/routing information is not modified
  - Typically used for host-host IPsec
- IPsec Tunnel mode: IP packet is encapsulated inside another IP packet. The IPsec transforms are applied to the inner (original) IP packet.
  - Protects IP and TCP header of the original packet
  - Typically used for VPNs
  - Routing information MAY be modified
- Note that we say that we establish an “SWAN tunnel” even for transport mode- the “tunnel” in this case is just the terminology for an end-to-end IPsec association.

IPsec encryption with ESP

- Encrypts data (either TCP/UDP payload for transport mode, or IP packet for tunnel mode)
- Adds an ESP header with an “Security Parameter Index” (SPI) and sequence number
  - SPI uniquely identifies a “Security Assocation” (SA) for which the security parameters (keys, crypto algo etc) are defined. Thus SPI essentially identifies a flow for IPsec
  - Sequence number is used to protect against replay attacks
- Adds an ESP trailer which contains the “original protocol” of the data that was encrypted.

What does each transform look like?

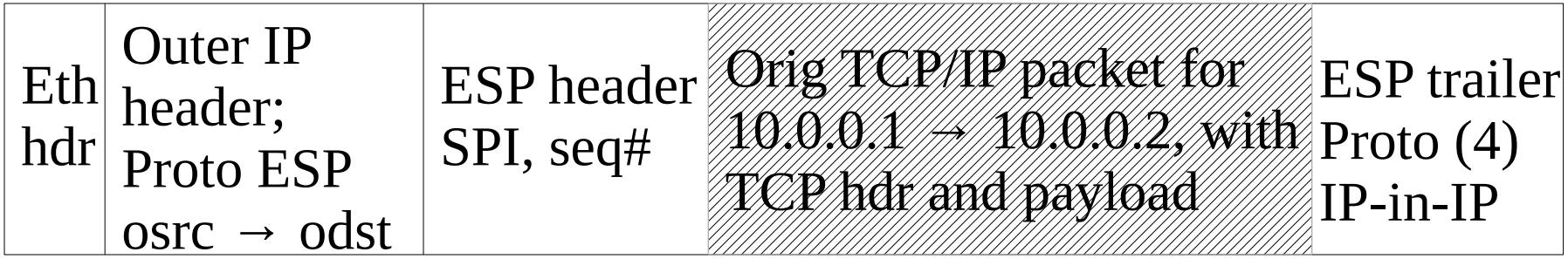
If we start with the following packet in clear (unencrypted packet):

Eth header	IP header; proto TCP 10.0.0.1 → 10.0.0.2	TCP hdr	TCP Payload

Effect of IPsec transforms

IPsec transport-mode encaps (ESP only)

IPsec tunnel mode. The outer osrc/odst are determined by VPN config. They would be the 10.0.0.1 and 10.0.0.2 if no VPN gw is used.



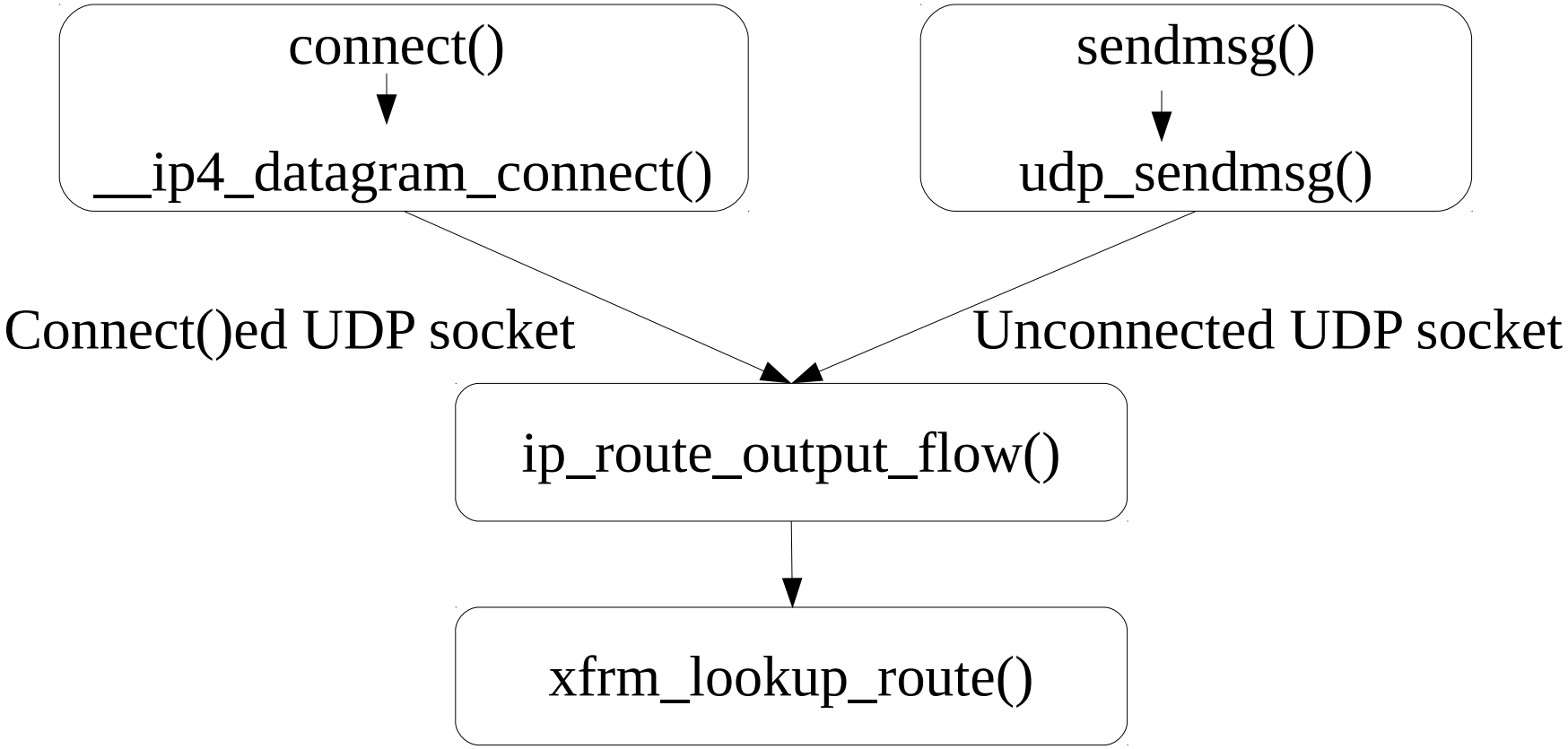
IKE: Internet Key Exchange

- Dynamically negotiates and establishes the security parameters for each IPsec tunnel, typically from user-space
  - Implemented in the “pluto” daemon from the *swan packages on Linux
- IKEv2: IETF RFC 7296
- Paul W will tell us more about IKE ….

Linux IPsec datapath for the transmit side

- Assume that a SPD entry has been setup for a flow from 13.0.0.8 → 13.0.0.0/24
- Follow the life of a UDP packet sent from 13.0.0.8 down the stack to see how IPsec gets applied.
  - Based on 4.16 source code
- This example will show on-demand SA establishment, and assumes that the SPD entry has been set up using some IKE implementation
- Will not cover IPsec offload

Route lookup for UDP transmit path



ip_route_output_flow() looks up the routing table via fib_lookup() to find the path to the destination (aka "dst_orig" in the code).
Then we look for any IPsec SPDs via xfrm_lookup_route(). Note that "XFRM" stands for IPsec transform

Finding the SA from the destination cache entry (dst_entry) in xfrm_lookup_route()

- For clear traffic, the dst_entry returned is the same as dst_orig

```
{
 dev = 0xffff8837ed520000,  /* net_device */
 ops = 0xffffffff821cda80 <ipv4_dst_ops>,
 _metrics = 18446744071592359009,
 expires = 0,
 xfrm = 0x0,
 input = 0xffffffff81679000 <dst_discard>,
 output = 0xffffffff816c2670 <ip_output>,
  :
```

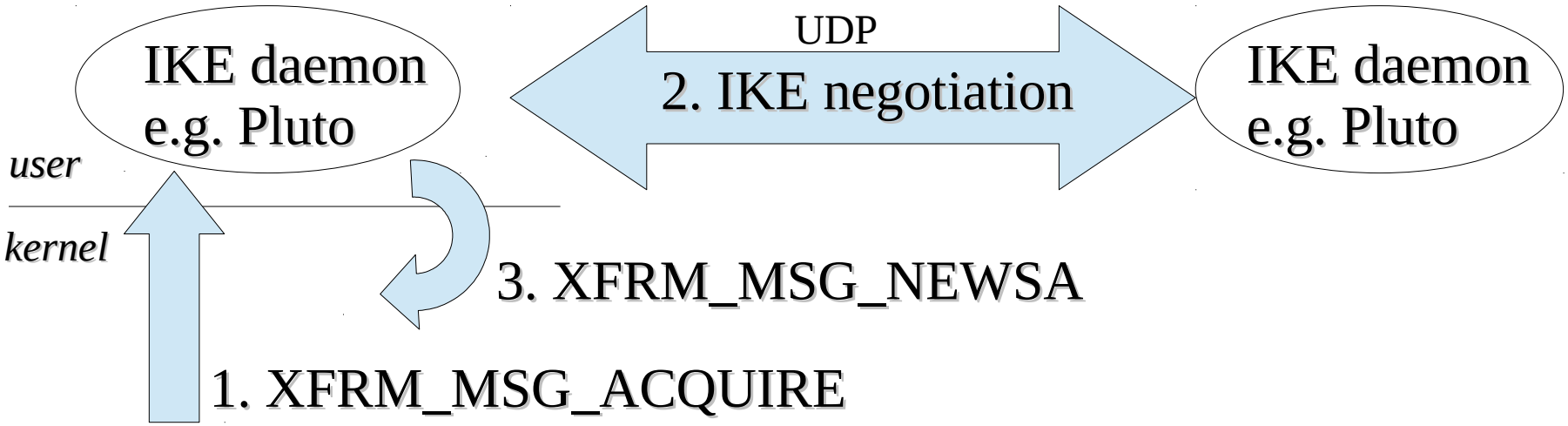
- IPsec traffic: If there is an applicable SPD and SADB entry for this flow, the SA will be returned in the `xfrm`
- "`xfrm_state`" is the Linux kernel data-structure that tracks SA's

```
{/* dst_entry from xfrm_lookup_route() */
dev = 0xffff8837ed4c0000,
ops = 0xffffffff821bb2c0 <init_net+4608>,
_metrics = 18446612372530779936,
expires = 0,
xfrm = 0xffff8837f1be5400, /* xfrm_state */
input = 0xffffffff81718a40 <dst_discard>,
output = 0xffffffff81717ff0 <xfrm4_output>,
 :
```

“On Demand” SADB creation

- The Security Association is tracked in a `xfrm_state` structure in the Linux kernel.
- The IKE daemon can be set up so that SA’s are created “on-demand”, i.e., establish the IKE tunnel and create the SA when traffic is triggered for the flow specified by the SPD.
  - This is common practice, to avoid creating tunnels when there is no traffic going through them.
- In this case, `xfrm_lookup_route`() will find a matching SPD entry, but no SA
- The kernel has to make an upcall to the IKE daemon to trigger SA creation for the packet.

On-demand SA creation via ACQUIRE




```
xfrm_send_acquire()
    :
xfrm_tmpl_resolve_one()
    :
xfrm_lookup_route()
```

Dispatch packet based on dst_entry returned from routing table lookup

- After the SADB lookup returns with success (SA was needed, we found one), the →`output` indirection in the `dst_entry` (aka dst_output()) points at xfrm4_output() for our UDP packet
- Any applicable Netfilter hooks may be applied in xfrm4_output. If these are successful (packet passes filters) __xfrm4_output() gets called.
- Now we need to actually apply the IPsec transform defined in the xfrm_state
  - let's see what the `xfrm_state` contains..

xfrm_state for the 13.0.0.8 → 13.0.0.9 SA

daddr = 13.0.0.9 spi = 0xfc92e79d proto = 50 (ESP)	xfrm_id
daddr=13.0.0.9 saddr=13.0.0.8 proto = 17 (UDP) :	xfrm_selector
alg_name=”rfc4106(gcm(aes))” alg_key_len=288 alg_icv_len=64 alg_key[]=0x9831b3b3b7..	xfrm_algo_aead
reqid=16397 ealgo=18 /* SADB_X_EALG_AES_GCM_ICV8 */ saddr=13.0.0.8 header_len = 16 trailer_len = 13 :	props
input = xfrm4_transport_input output = xfrm4_transport_output afinfo = xfrm4_state_afinfo :	xfrm_mode[1]

1: see backup slides for additional details

Dispatch packet based on xfrm_mode

- The outer_mode for a transport mode SA is the `xfrm4_transport_mode`. Relevant indirections are

```
input = 0xffffffffa0a23120 <xfrm4_transport_input>,
output = 0xffffffffa0a23080 <xfrm4_transport_output>,
afinfo = 0xffffffff821d1680 <xfrm4_state_afinfo>,
```

- __xfrm4_output invokes the afinfo→output_finish(), which is xfrm4_output_finish
- xfrm_output_finish() calls xfrm_output(), from where multiple IPsec perf features may be invoked.

xfrm_output() and beyond..

- If IPsec h/w offload can be applied (outgoing interfaces supports it, packet will not need ip fragmentation etc), we may set up state needed for offload in the packet (e.g., pointers to the xfrm_state for the NIC)
- If the packet is eligible for GSO the actual IPsec transform is deferred (will be done after GSO has segmented the packet)
- For small/non-GSO packets, we would call xfrm_output2→xfrm_output_resume() to do the IPsec transform

Applying the IPsec transform in xfrm_output_resume()

- Call xfrm_output_one() to do the following using info in the xfrm_state "x"
  - xfrm_skb_check_space: Is there space for ESP header?
  - x→outer_mode→output() (xfrm4_transport_output) adds the ESP encapsulation header
  - xfrm_state_check_expire(): did the key expire?
  - x→repl→overflow(): replay protection check (xfrm_replay_overflow)
  - Either offload the crypto or call x→type→output() (esp_output) to encrypt
- If all is still well, skb_dst(skb)→ops→local_out() aka __ip_local_out() is called

Sending the packet through IP

- The dst_output() is now ip_output, packet is encrypted.
- ip_output() → ip_finish_output() sends the IPv4 packet on its way
  - May be fragmented, in which case each fragment is sent with IP proto 50

Linux IPsec: UDP receive side path

- Two cases possible based on whether or not we have GRO enabled
- What is GRO?
  - Try to coalesce small packets in the same flow into a large packet before sending it up the stack. Builds on the principles described for clear packets in https://lwn.net/Articles/358910/

Case 1: no IPsec GRO, setting up the Destination Cache Entry

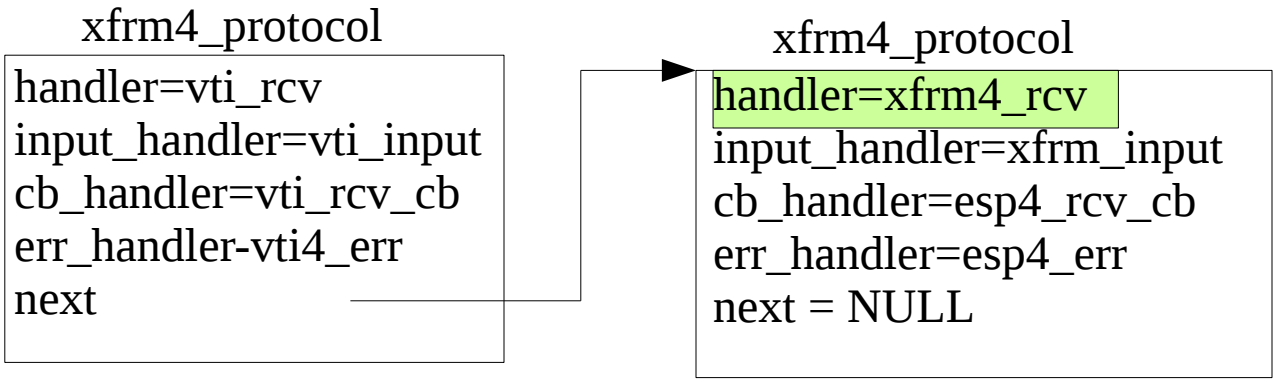
- Most drivers first try to deliver the packet through the GRO callback path, via napi_gro_receive()
- napi_gro_receive() calls dev_gro_receive()
  - In our example, the skb→protocol for the packet is the ethernet type 0x800, so in dev_gro_receive() we try to call inet_gro_receive()
  - But inet_gro_receive() will not find an entry for inet_offloads[IPPROTO_ESP]
  - So we fall back to the non-GRO path..
- Non-GRO path: napi_gro_receive() ends up in netif_receive_skb_core→ip_rcv→ip_rcv_finish
- We set up the dst_entry via ip_route_input_noref()
  - The dst_input is set to ip_local_deliver()

ip_local_deliver → XFRM

- ip_local_deliver → ip_local_deliver_finish looks up inet_protos[IPPROTO_ESP]
- The →handler() is xfrm4_esp_rcv(), so we drop into the XFRM code..

xfrm4_esp_rcv()

- xfrm4_esp_rcv goes through, and invokes, the list of handlers that were registered via xfrm4_protocol_register()
  - If there is a non-null handler that returns something other than EINVAL, the packet is considered “consumed”
- What’s in esp4_handlers? On my test machine, I have CONFIG_NET_IPVTI enabled, so I find:



Rx side IPsec processing: xfrm_input

- xfrm4_rcv → xfrm4_rcv_spi → xfrm_input
- Parse the packet for the SPI and find the SA
- Validates SA (check lifetime, replay protection..)
- If there was no h/w offload support, call esp_input and decrypt
- Reinject the decrypted packet to the IP stack via ip_local_deliver (the value in dst_input(skb))
  - When GRO is not available, this is done through a tasklet whose callback is xfrm_trans_reinject()

Case 2: IPsec GRO is enabled

- Basic idea behind GRO: coalesce packets in the same flow and try to send large packets up the stack
- For IPsec, first decrypt, then send the clear packet to the GRO callbacks for UDP/IP
- IPsec GRO is enabled when the esp4_offload modules is loaded in the kernel
  - Manually, by ‘modprobe esp4_offload’
  - By enabling crypto offload during SA addition

Callbacks provided by esp4_offload

- Loading the esp4_offload module adds the following callback to inet_offloads[50] for IPPROTO_ESP

```
static const struct net_offload esp4_offload = {
        .callbacks = {
                .gro_receive = esp4_gro_receive,
                .gso_segment = esp4_gso_segment,
        },
};
```

- So an incoming packet will now be processed in esp4_gro_receive() from inet_gro_receive()

esp4_gro_receive() processing

- Find the xfrm_state (SA), set up some GRO state in the sk_buff's secpath
- Call xfrm_input() for the Rx side IPsec processing
- Re-inject the packet via gro_cell_receive
  - NAPI framework is used for re-inject. The xfrm module has a dummy device, `xfrm_napi_dev`, that enables this.
- NAPI requeue will pick up the clear packet and pass it to inet_gro_receive() where we follow the UDP/IP path for GRO.

Backup Slides

Gory details about inner_mode, outer_mode, inner_mode_iaf in the SA

- The struct `xfrm_state` actually has 3 `xfrm_mode` structures:`inner_mode, outer_mode, inner_mode_iaf`
- When in tunnel mode, if the inner packet (being encrypted is IPv6 and the outer (VPN) header is IPv4 (or vice-versa: inner is IPv4, outer is IPv6) then the `inner_mode` != `outer_mode.`
  - The inner_mode corresponds to the `xfrm_mode` of the inner packet
  - In this case the inner_mode_iaf gives the `xfrm_mode` for the “other” IP version. The inner_mode_iaf is only used when x->sel.family is AF_UNSPEC. In this case we try to get the correct mode from the packet header information. Based on that, we use either inner_mode or inner_mode_iaf (see xfrm_prepare_input())
- In all other cases, the `inner_mode` == `outer_mode`
- `inner_mode_iaf` is always NULL for transport mode