IPsec encapsulation over TCP

Sabrina Dubroca sd@queasysnail.net

Red Hat

netdev 0x13, 2019-03-22

Introduction

ESP and IKE

- Components of the IPsec protocol suite

  ESP **Encapsulating Security Payload**
  AH Authentication Header
  IKE **Internet Key Exchange**

- Packet formats

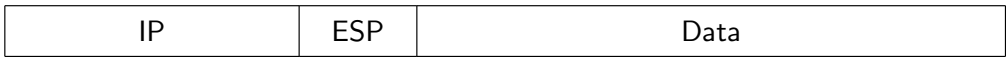

Figure: ESP packet format

Figure: IKE over UDP port 500 packet format

UDP encapsulation

- workaround for NAT: NAT needs ports to demultiplex inside/outside

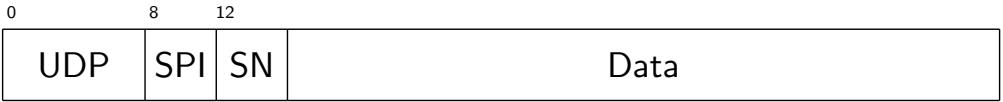

Figure: ESP over UDP packet format

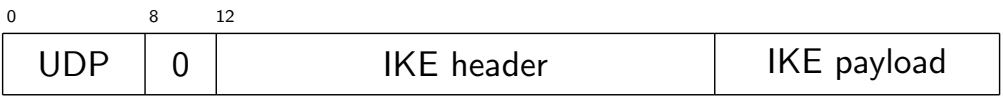

Figure: IKE over UDP port 4500 packet format

- RX node uses the first 32 bits of UDP payload to differentiate ESP/IKE

More middlebox trouble

Some middleboxes don't let anything other than TCP pass.

RFC 8229: TCP Encapsulation of IKE and ESP Packets

RFC 8229: packet formats

TCP stream, composed of concatenated messages

Figure: ESP over TCP message format

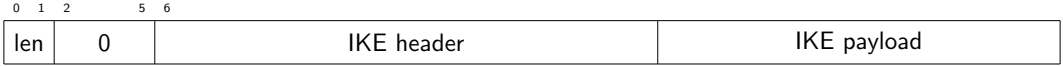

Figure: IKE over TCP message format

RFC 8229: sender and receiver operations

- sender
  - adds a prefix to each message
    - for all messages: length (2 bytes)
    - for IKE messages: non-ESP marker (like in UDP encapsulation)
  - concatenates all messages within a TCP stream
- receiver
  - parses TCP stream to extract messages
  - differentiates ESP from IKE (SPI/non-ESP marker)
  - handles each message

Linux implementation

XFRM

- infrastructure underneath linux kernel's IPsec implementation
- applies transformations on packets

Upper Layer Protocol (ULP)

- infrastracture to implement in-kernel protocols that live on top of TCP
  - for example, TLS
- `init` callback triggered by the `TCP_ULP` setsockopt

```
#define ULPNAME <...>
setsockopt(sock, SOL_TCP, TCP_ULP,
           ULPNAME, sizeof(ULPNAME));
```

- `init` callback can change some of the socket's operations
  - replace operations with protocol-specific actions
  - for example, sendmsg/recvmsg, close

Stream Parser (strp)

- framework to parse messages out of a TCP stream
- main callbacks
  - parse_msg returns the length of the next message in the stream
  - rcv_msg processes the next message
- strp's receive function
  - triggered by arrival of new data on the TCP socket
  - calls those operations

RFC 8229 implementation

- uses ULP
  - initializes a streamparser on the TCP socket
  - redefines socket operations
- streamparser
  - extracts messages, either IKE or ESP
  - passes messages to userspace or XFRM

RFC 8229 implementation: RX handling

- data arrives on TCP encap socket → `__strp_rcv()` → `parse_msg` + `rcv_msg`
- `parse_msg`
  - reads length field
- `rcv_msg`
  - reads SPI/non-ESP marker
  - continues processing as ESP or IKE
- ESP messages → XFRM → decrypt/verify
- IKE messages → userspace queue → `recv()` (via custom `recvmsg` op)

RFC 8229 implementation: TX handling

- IKE daemon → `send()` → custom `sendmsg` op → add length prefix → enqueue to TCP
- data packets → XFRM → add length prefix → enqueue to TCP

RFC 8229 implementation: TX handling: interleaving of messages

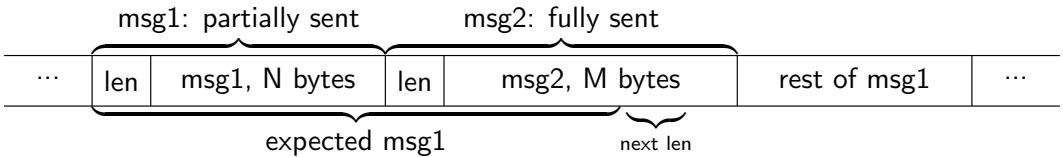

Figure: Interleaving problem between IKE and ESP

- avoid interleaving messages over TCP
  - temporary slot in front of the TCP socket
  - keep that partially-sent message in the slot

Userspace view

Userspace API: creating XFRM states

- almost identical to UDP encap: `s/espinudp/espintcp/`

```
CPORT=<local port of the client socket>
SPORT=4500
CADDR=<IP address of the client>
SADDR=<IP address of the server>

ip xfrm state add src $CADDR dst $SADDR proto esp spi $SPI1  \
                  aead 'rfc4106(gcm(aes))' $KEY $ICVLEN      \
                  mode transport sel src $CADDR dst $SADDR   \
                  encap espintcp $CPORT $SPORT 0.0.0.0
ip xfrm state add src $SADDR dst $CADDR proto esp spi $SPI2  \
                  aead 'rfc4106(gcm(aes))' $KEY $ICVLEN      \
                  mode transport sel src $SADDR dst $CADDR   \
                  encap espintcp $SPORT $CPORT 0.0.0.0
```

Userspace API: client program

```
sock = socket(AF_INET, SOCK_STREAM, 0);

struct xfrm_userpolicy_info policy = {
        .action = XFRM_POLICY_ALLOW,
        .sel.family = AF_INET,
};

policy.dir = XFRM_POLICY_OUT;
setsockopt(sock, IPPROTO_IP, IP_XFRM_POLICY, &policy, sizeof(policy));
policy.dir = XFRM_POLICY_IN;
setsockopt(sock, IPPROTO_IP, IP_XFRM_POLICY, &policy, sizeof(policy));
connect(sock, ...);
send(sock, "IKETCP", 6, 0);
setsockopt(sock, SOL_TCP, TCP_ULP, "espintcp", sizeof("espintcp"));
```

Socket behavior

- TCP socket, but behaves like a UDP socket once encapsulation is enabled
- length prefix added/removed by kernel
- non-ESP marker is userspace's responsibility (identical to ESPINUDP encap)
- `"IKETCP"` prefix
  - written sent by userspace client before enabling encapsulation
  - read by userspace server (after accept()) before enabling encapsulation (otherwise, close connection)
- one `send()` = 1 IKE message
  - `MSG_MORE` not implemented
- one `recv()` = 1 IKE message
  - recv buffer smaller than actual message ⇒ partial read, rest of the message dropped
  - `MSG_PEEK` returns the first N bytes of the message at the head of the receive queue

Conclusion

Remaining work

- Possible starvation issue between ESP and IKE
- Testing with IKE daemon
- Upstreaming