Kernel TLS offload

Kernel TLS operation

Linux kernel provides TLS connection offload infrastructure. Once a TCP connection is in `ESTABLISHED` state user space can enable the TLS Upper Layer Protocol (ULP) and install the cryptographic connection state. For details regarding the user-facing interface refer to the TLS documentation in :ref:`Documentation/networking/tls.rst <kernel_tls>`.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst`, line 10); *backlink***
>
> Unknown interpreted text role "ref".

`ktls` can operate in three modes:

- Software crypto mode (`TLS_SW`) - CPU handles the cryptography. In most basic cases only crypto operations synchronous with the CPU can be used, but depending on calling context CPU may utilize asynchronous crypto accelerators. The use of accelerators introduces extra latency on socket reads (decryption only starts when a read syscall is made) and additional I/O load on the system.
- Packet-based NIC offload mode (`TLS_HW`) - the NIC handles crypto on a packet by packet basis, provided the packets arrive in order. This mode integrates best with the kernel stack and is described in detail in the remaining part of this document (`ethtool` flags `tls-hw-tx-offload` and `tls-hw-rx-offload`).
- Full TCP NIC offload mode (`TLS_HW_RECORD`) - mode of operation where NIC driver and firmware replace the kernel networking stack with its own TCP handling, it is not usable in production environments making use of the Linux networking stack for example any firewalling abilities or QoS and packet scheduling (`ethtool` flag `tls-hw-record`).

The operation mode is selected automatically based on device configuration, offload opt-in or opt-out on per-connection basis is not currently supported.

TX

At a high level user write requests are turned into a scatter list, the TLS ULP intercepts them, inserts record framing, performs encryption (in `TLS_SW` mode) and then hands the modified scatter list to the TCP layer. From this point on the TCP stack proceeds as normal.

In `TLS_HW` mode the encryption is not performed in the TLS ULP. Instead packets reach a device driver, the driver will mark the packets for crypto offload based on the socket the packet is attached to, and send them to the device for encryption and transmission.

RX

On the receive side if the device handled decryption and authentication successfully, the driver will set the decrypted bit in the associated :c:type:`struct sk_buff <sk_buff>`. The packets reach the TCP stack and are handled normally. `ktls` is informed when data is queued to the socket and the `strparser` mechanism is used to delineate the records. Upon read request, records are retrieved from the socket and passed to decryption routine. If device decrypted all the segments of the record the decryption is skipped, otherwise software path handles decryption.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst`, line 54); *backlink***
>
> Unknown interpreted text role "c:type".

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst`, line 63)**
>
> Unknown directive type "kernel-figure".
>
> ```
> .. kernel-figure::  tls-offload-layers.svg
>    :alt:        TLS offload layers
>    :align:      center
>    :figwidth:   28em
> ```

```
        Layers of Kernel TLS stack
```

Device configuration

During driver initialization device sets the `NETIF_F_HW_TLS_RX` and `NETIF_F_HW_TLS_TX` features and installs its :c:type:`struct tlsdev_ops <tlsdev_ops>` pointer in the :c:member:`tlsdev_ops` member of the :c:type:`struct net_device <net_device>`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 73); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 73); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 73); *backlink*

Unknown interpreted text role "c:type".

When TLS cryptographic connection state is installed on a `ktls` socket (note that it is done twice, once for RX and once for TX direction, and the two are completely independent), the kernel checks if the underlying network device is offload-capable and attempts the offload. In case offload fails the connection is handled entirely in software using the same mechanism as if the offload was never tried.

Offload request is performed via the :c:member:`tls_dev_add` callback of :c:type:`struct tlsdev_ops <tlsdev_ops>`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 86); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 86); *backlink*

Unknown interpreted text role "c:type".

```
int (*tls_dev_add)(struct net_device *netdev, struct sock *sk,
                   enum tls_offload_ctx_dir direction,
                   struct tls_crypto_info *crypto_info,
                   u32 start_offload_tcp_sn);
```

`direction` indicates whether the cryptographic information is for the received or transmitted packets. Driver uses the `sk` parameter to retrieve the connection 5-tuple and socket family (IPv4 vs IPv6). Cryptographic information in `crypto_info` includes the key, iv, salt as well as TLS record sequence number. `start_offload_tcp_sn` indicates which TCP sequence number corresponds to the beginning of the record with sequence number from `crypto_info`. The driver can add its state at the end of kernel structures (see :c:member:`driver_state` members in `include/net/tls.h`) to avoid additional allocations and pointer dereferences.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 96); *backlink*

Unknown interpreted text role "c:member".

TX

After TX state is installed, the stack guarantees that the first segment of the stream will start exactly at the `start_offload_tcp_sn` sequence number, simplifying TCP sequence number matching.

TX offload being fully initialized does not imply that all segments passing through the driver and which belong to the offloaded socket will be after the expected sequence number and will have kernel record information. In particular, already encrypted data may have been queued to the socket before installing the connection state in the kernel.

RX

In RX direction local networking stack has little control over the segmentation, so the initial records' TCP sequence number may be anywhere inside the segment.

Normal operation

At the minimum the device maintains the following state for each connection, in each direction:

- crypto secrets (key, iv, salt)
- crypto processing state (partial blocks, partial authentication tag, etc.)
- record metadata (sequence number, processing offset and length)
- expected TCP sequence number

There are no guarantees on record length or record segmentation. In particular segments may start at any point of a record and contain any number of records. Assuming segments are received in order, the device should be able to perform crypto operations and authentication regardless of segmentation. For this to be possible device has to keep small amount of segment-to-segment state. This includes at least:

- partial headers (if a segment carried only a part of the TLS header)
- partial data block
- partial authentication tag (all data had been seen but part of the authentication tag has to be written or read from the subsequent segment)

Record reassembly is not necessary for TLS offload. If the packets arrive in order the device should be able to handle them separately and make forward progress.

TX

The kernel stack performs record framing reserving space for the authentication tag and populating all other TLS header and tailer fields.

Both the device and the driver maintain expected TCP sequence numbers due to the possibility of retransmissions and the lack of software fallback once the packet reaches the device. For segments passed in order, the driver marks the packets with a connection identifier (note that a 5-tuple lookup is insufficient to identify packets requiring HW offload, see the :ref:`5tuple_problems` section) and hands them to the device. The device identifies the packet as requiring TLS handling and confirms the sequence number matches its expectation. The device performs encryption and authentication of the record data. It replaces the authentication tag and TCP checksum with correct values.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst`, **line 159**); *backlink*
>
> Unknown interpreted text role "ref".

RX

Before a packet is DMAed to the host (but after NIC's embedded switching and packet transformation functions) the device validates the Layer 4 checksum and performs a 5-tuple lookup to find any TLS connection the packet may belong to (technically a 4-tuple lookup is sufficient - IP addresses and TCP port numbers, as the protocol is always TCP). If connection is matched device confirms if the TCP sequence number is the expected one and proceeds to TLS handling (record delineation, decryption, authentication for each record in the packet). The device leaves the record framing unmodified, the stack takes care of record decapsulation. Device indicates successful handling of TLS offload in the per-packet context (descriptor) passed to the host.

Upon reception of a TLS offloaded packet, the driver sets the :c:member:`decrypted` mark in :c:type:`struct sk_buff <sk_buff>` corresponding to the segment. Networking stack makes sure decrypted and non-decrypted segments do not get coalesced (e.g. by GRO or socket layer) and takes care of partial decryption.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst`, **line 185**); *backlink*
>
> Unknown interpreted text role "c:member".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 185); *backlink*

Unknown interpreted text role "c:type".

Resync handling

In presence of packet drops or network packet reordering, the device may lose synchronization with the TLS stream, and require a resync with the kernel's TCP stack.

Note that resync is only attempted for connections which were successfully added to the device table and are in TLS_HW mode. For example, if the table was full when cryptographic state was installed in the kernel, such connection will never get offloaded. Therefore the resync request does not carry any cryptographic connection state.

TX

Segments transmitted from an offloaded socket can get out of sync in similar ways to the receive side-retransmissions - local drops are possible, though network reorders are not. There are currently two mechanisms for dealing with out of order segments.

Crypto state rebuilding

Whenever an out of order segment is transmitted the driver provides the device with enough information to perform cryptographic operations. This means most likely that the part of the record preceding the current segment has to be passed to the device as part of the packet context, together with its TCP sequence number and TLS record number. The device can then initialize its crypto state, process and discard the preceding data (to be able to insert the authentication tag) and move onto handling the actual packet.

In this mode depending on the implementation the driver can either ask for a continuation with the crypto state and the new sequence number (next expected segment is the one after the out of order one), or continue with the previous stream state - assuming that the out of order segment was just a retransmission. The former is simpler, and does not require retransmission detection therefore it is the recommended method until such time it is proven inefficient.

Next record sync

Whenever an out of order segment is detected the driver requests that the `ktls` software fallback code encrypt it. If the segment's sequence number is lower than expected the driver assumes retransmission and doesn't change device state. If the segment is in the future, it may imply a local drop, the driver asks the stack to sync the device to the next record state and falls back to software.

Resync request is indicated with:

```
void tls_offload_tx_resync_request(struct sock *sk, u32 got_seq, u32 exp_seq)
```

Until resync is complete driver should not access its expected TCP sequence number (as it will be updated from a different context). Following helper should be used to test if resync is complete:

```
bool tls_offload_tx_resync_pending(struct sock *sk)
```

Next time `ktls` pushes a record it will first send its TCP sequence number and TLS record number to the driver. Stack will also make sure that the new record will start on a segment boundary (like it does when the connection is initially added).

RX

A small amount of RX reorder events may not require a full resynchronization. In particular the device should not lose synchronization when record boundary can be recovered:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 268)

Unknown directive type "kernel-figure".

```
.. kernel-figure::  tls-offload-reorder-good.svg
   :alt:        reorder of non-header segment
   :align:      center

   Reorder of non-header segment
```

Green segments are successfully decrypted, blue ones are passed as received on wire, red stripes mark start of new records.

In above case segment 1 is received and decrypted successfully. Segment 2 was dropped so 3 arrives out of order. The device knows the next record starts inside 3, based on record length in segment 1. Segment 3 is passed untouched, because due to lack of

data from segment 2 the remainder of the previous record inside segment 3 cannot be handled. The device can, however, collect the authentication algorithm's state and partial block from the new record in segment 3 and when 4 and 5 arrive continue decryption. Finally when 2 arrives it's completely outside of expected window of the device so it's passed as is without special handling. `ktls` software fallback handles the decryption of record spanning segments 1, 2 and 3. The device did not get out of sync, even though two segments did not get decrypted.

Kernel synchronization may be necessary if the lost segment contained a record header and arrived after the next record header has already passed:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 293)**
>
> Unknown directive type "kernel-figure".
>
> ```
> .. kernel-figure::  tls-offload-reorder-bad.svg
>    :alt:        reorder of header segment
>    :align:      center
>
>    Reorder of segment with a TLS header
> ```

In this example segment 2 gets dropped, and it contains a record header. Device can only detect that segment 4 also contains a TLS header if it knows the length of the previous record from segment 2. In this case the device will lose synchronization with the stream.

Stream scan resynchronization

When the device gets out of sync and the stream reaches TCP sequence numbers more than a max size record past the expected TCP sequence number, the device starts scanning for a known header pattern. For example for TLS 1.2 and TLS 1.3 subsequent bytes of value `0x03 0x03` occur in the SSL/TLS version field of the header. Once pattern is matched the device continues attempting parsing headers at expected locations (based on the length fields at guessed locations). Whenever the expected location does not contain a valid header the scan is restarted.

When the header is matched the device sends a confirmation request to the kernel, asking if the guessed location is correct (if a TLS record really starts there), and which record sequence number the given header had. The kernel confirms the guessed location was correct and tells the device the record sequence number. Meanwhile, the device had been parsing and counting all records since the just-confirmed one, it adds the number of records it had seen to the record number provided by the kernel. At this point the device is in sync and can resume decryption at next segment boundary.

In a pathological case the device may latch onto a sequence of matching headers and never hear back from the kernel (there is no negative confirmation from the kernel). The implementation may choose to periodically restart scan. Given how unlikely falsely-matching stream is, however, periodic restart is not deemed necessary.

Special care has to be taken if the confirmation request is passed asynchronously to the packet stream and record may get processed by the kernel before the confirmation request.

Stack-driven resynchronization

The driver may also request the stack to perform resynchronization whenever it sees the records are no longer getting decrypted. If the connection is configured in this mode the stack automatically schedules resynchronization after it has received two completely encrypted records.

The stack waits for the socket to drain and informs the device about the next expected record number and its TCP sequence number. If the records continue to be received fully encrypted stack retries the synchronization with an exponential back off (first after 2 encrypted records, then after 4 records, after 8, after 16... up until every 128 records).

Error handling

TX

Packets may be redirected or rerouted by the stack to a different device than the selected TLS offload device. The stack will handle such condition using the :c:func:`sk_validate_xmit_skb` helper (TLS offload code installs :c:func:`tls_validate_xmit_skb` at this hook). Offload maintains information about all records until the data is fully acknowledged, so if skbs reach the wrong device they can be handled by software fallback.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 359); *backlink***
>
> Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 359); *backlink*

Unknown interpreted text role "c:func".

Any device TLS offload handling error on the transmission side must result in the packet being dropped. For example if a packet got out of order due to a bug in the stack or the device, reached the device and can't be encrypted such packet must be dropped.

RX

If the device encounters any problems with TLS offload on the receive side it should pass the packet to the host's networking stack as it was received on the wire.

For example authentication failure for any record in the segment should result in passing the unmodified packet to the software fallback. This means packets should not be modified "in place". Splitting segments to handle partial decryption is not advised. In other words either all records in the packet had been handled successfully and authenticated or the packet has to be passed to the host's stack as it was on the wire (recovering original packet in the driver if device provides precise error is sufficient).

The Linux networking stack does not provide a way of reporting per-packet decryption and authentication errors, packets with errors must simply not have the :c:member:`decrypted` mark set.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\networking\(linux-master)(Documentation)(networking)tls-offload.rst, line 387); *backlink*

Unknown interpreted text role "c:member".

A packet should also not be handled by the TLS offload if it contains incorrect checksums.

Performance metrics

TLS offload can be characterized by the following basic metrics:

- max connection count
- connection installation rate
- connection installation latency
- total cryptographic performance

Note that each TCP connection requires a TLS session in both directions, the performance may be reported treating each direction separately.

Max connection count

The number of connections device can support can be exposed via `devlink resource` API.

Total cryptographic performance

Offload performance may depend on segment and record size.

Overload of the cryptographic subsystem of the device should not have significant performance impact on non-offloaded streams.

Statistics

Following minimum set of TLS-related statistics should be reported by the driver:

- `rx_tls_decrypted_packets` - number of successfully decrypted RX packets which were part of a TLS stream.
- `rx_tls_decrypted_bytes` - number of TLS payload bytes in RX packets which were successfully decrypted.
- `rx_tls_ctx` - number of TLS RX HW offload contexts added to device for decryption.
- `rx_tls_del` - number of TLS RX HW offload contexts deleted from device (connection has finished).
- `rx_tls_resync_req_pkt` - number of received TLS packets with a resync request.
- `rx_tls_resync_req_start` - number of times the TLS async resync request was started.
- `rx_tls_resync_req_end` - number of times the TLS async resync request properly ended with providing the HW tracked tcp-seq.
- `rx_tls_resync_req_skip` - number of times the TLS async resync request

procedure was started by not properly ended.

- `rx_tls_resync_res_ok` - number of times the TLS resync response call to the driver was successfully handled.
- `rx_tls_resync_res_skip` - number of times the TLS resync response call to the driver was terminated unsuccessfully.
- `rx_tls_err` - number of RX packets which were part of a TLS stream but were not decrypted due to unexpected error in the state machine.
- `tx_tls_encrypted_packets` - number of TX packets passed to the device for encryption of their TLS payload.
- `tx_tls_encrypted_bytes` - number of TLS payload bytes in TX packets passed to the device for encryption.
- `tx_tls_ctx` - number of TLS TX HW offload contexts added to device for encryption.
- `tx_tls_ooo` - number of TX packets which were part of a TLS stream but did not arrive in the expected order.
- `tx_tls_skip_no_sync_data` - number of TX packets which were part of a TLS stream and arrived out-of-order, but skipped the HW offload routine and went to the regular transmit flow as they were retransmissions of the connection handshake.
- `tx_tls_drop_no_sync_data` - number of TX packets which were part of a TLS stream dropped, because they arrived out of order and associated record could not be found.
- `tx_tls_drop_bypass_req` - number of TX packets which were part of a TLS stream dropped, because they contain both data that has been encrypted by software and data that expects hardware crypto offload.

Notable corner cases, exceptions and additional requirements

5-tuple matching limitations

The device can only recognize received packets based on the 5-tuple of the socket. Current `ktls` implementation will not offload sockets routed through software interfaces such as those used for tunneling or virtual networking. However, many packet transformations performed by the networking stack (most notably any BPF logic) do not require any intermediate software device, therefore a 5-tuple match may consistently miss at the device level. In such cases the device should still be able to perform TX offload (encryption) and should fallback cleanly to software decryption (RX).

Out of order

Introducing extra processing in NICs should not cause packets to be transmitted or received out of order, for example pure ACK packets should not be reordered with respect to data segments.

Ingress reorder

A device is permitted to perform packet reordering for consecutive TCP segments (i.e. placing packets in the correct order) but any form of additional buffering is disallowed.

Coexistence with standard networking offload features

Offloaded `ktls` sockets should support standard TCP stack features transparently. Enabling device TLS offload should not cause any difference in packets as seen on the wire.

Transport layer transparency

The device should not modify any packet headers for the purpose of the simplifying TLS offload.

The device should not depend on any packet headers beyond what is strictly necessary for TLS offload.

Segment drops

Dropping packets is acceptable only in the event of catastrophic system errors and should never be used as an error handling mechanism in cases arising from normal operation. In other words, reliance on TCP retransmissions to handle corner cases is not acceptable.

TLS device features

Drivers should ignore the changes to the TLS device feature flags. These flags will be acted upon accordingly by the core `ktls` code. TLS device feature flags only control adding of new TLS connection offloads, old connections will remain active after flags are cleared.

TLS encryption cannot be offloaded to devices without checksum calculation offload. Hence, TLS TX device feature flag requires TX csum offload being set. Disabling the latter implies clearing the former. Disabling TX checksum offload should not affect old connections, and drivers should make sure checksum calculation does not break for them. Similarly, device-offloaded TLS decryption implies doing RXCSUM. If the user does not want to enable RX csum offload, TLS RX device feature is disabled as well.