NVIDIA TLS Offload Guide
Table of contents

Introduction

TLS Handshake

kTLS

HW-offloading kTLS

kTLS Offload Flow in High Level

Resync and Error Handling

Prerequisites

Checking Hardware Support for Crypto Acceleration

Kernel Requirements

Configurations and Useful Commands

TLS Setup

Finding NVIDIA Interfaces

Configuring TLS Offload

Configuring OVS Bridge on BlueField

Common Use Cases

OpenSSL

Nginx

Prerequisites

Configuration

Stopping Nginx

Wrk – Client
Using Wrk

Testing Offload via OpenSSL

TLS Testing Setup

Adding Certificate and Key

Running Server Side

Running Client Side

Testing kTLS

Optimizations over kTLS

XLIO

Performance Tuning Options

Additional Reading
This guide provides an overview and configuration steps of TLS hardware offloading via kernel-TLS, using hardware capabilities of NVIDIA® BlueField® DPU.

**Introduction**

Transport layer security (TLS) is a cryptographic protocol designed to provide communications security over a computer network. The protocol is widely used in applications such as email, instant messaging, and voice over IP (VoIP), but its use in securing HTTPS remains the most publicly visible.

The TLS protocol aims primarily to provide cryptography, including privacy (confidentiality), integrity, and authenticity using certificates, between two or more communicating computer applications. It runs in the application layer and is itself composed of two layers: the TLS record and the TLS handshake protocols.

TLS works over TCP and consists of 3 phases:

1. **Handshake** – establishment of a connection
2. **Application** – sending and receiving encrypted packets
3. **Termination** – connection termination

**TLS Handshake**

In the handshake phase, the client and server decide on which cipher suites they will use, and exchange keys and certificates according to the following flow:

1. **Client hello**, provides the server at a minimum with the following:
   - A key exchange algorithm, to determine how symmetric keys are exchanged
   - An authentication or digital signature algorithm, which dictates how server authentication and client authentication (if required) are implemented
   - A bulk encryption cipher, which is used to encrypt the data
   - A hash/MAC (message authentication code) function, which determines how data integrity checks are carried out
The version of the protocol it understands

- The cipher suites it is capable of working with

- A unique random number, which is important to guard against replay attacks

2. Server hello:

- Selects a cipher suite

- Generates its own random number

- Assigns a session ID to the TLS connection

- Sends enough information to complete a key exchange—most often, this means sending a certificate including an RSA public key

3. Client:

- Responsible for completing the key exchange using the information the server provided

At this point, the connection is secured, both sides have agreed on an encryption algorithm, a MAC algorithm, and respective keys.

**kTLS**

The Linux kernel provides TLS offload infrastructure. kTLS (kernel TLS) offloads TLS handling from the user-space to the kernel-space.

kTLS has 3 modes of operation:

- **SW** – all operation is handled in kernel (i.e., handshake, encryption, decryption)

- **HW-offload** (the focus of this guide) – handshake and error handling are performed in software. Packets are encrypted/decrypted in hardware. In this case, there is an additional offload from the kernel to the hardware.

- **HW-record** – all operations are handled by the hardware (driver and firmware) including the handshake. It also handles its own TCP session. This option is currently
not supported.

Note

It is important to understand that Rx (receiving) and Tx (sending) can have two separate modes. For example, Rx can be dealt in SW mode but Tx in HW-offload mode (i.e., the hardware will only encrypt but not decrypt).

HW-offloading kTLS

In general, the TLS HW-offload performs best and provides optimal value on longer lived sessions, with relatively large packets. Scaling in terms of concurrent connections and connections per second is use-case dependent (e.g., the amount of active concurrent connections from the overall open concurrent connections is material).

It is necessary to learn the following terms before proceeding:

- The transport interface send (TIS) object is responsible for performing all transport-related operations of the transmit side. Messages from Send Queues (SQs) get segmented and transmitted by the TIS including all transport required implications. For example, in the case of a large send offload, the TIS is responsible for the segmentation. The NVIDIA® ConnectX® hardware uses a TIS object to save and access the TLS crypto information and state of an offloaded Tx kTLS connection.

- The transport interface receive (TIR) object is responsible for performing all transport-related operations on the receive side. TIR performs the packet processing and reassembly and is also responsible for demultiplexing the packets into different receive queues (RQs).

- Both TIS and TIR hold the data encryption key (DEK).

kTLS Offload Flow in High Level
1. Establishes a TLS connection with remote host (server or client) by handling a TLS handshake by kernel on current host.

2. Initializes the following state for each connection, Rx and Tx:
   - Crypto secrets (e.g., public key)
   - Crypto processing state
   - Record metadata (e.g., record sequence number, offset)
   - Expected TCP sequence number

**Tx flow:**

1. Packets belonging to device offloaded sockets arrive to the kernel and it does not encrypt them.

2. Kernel performs record framing and marks the packet with a connection identifier.

3. Kernel sends packets to the device driver for offloading.

4. Device checks that the sequence number matches the state in the TIS and performs encryption and authentication.

**Rx flow:**

1. When the connection is created, a HW steering rule is added to steer packets to their respective TIR.

2. Device receives the packet then validates and checks that sequence number of TCP matches the state in the TIR.

3. Performs decryption and authentication, and indicates in the CQE (completion queue entry).

### Note

The following flow does not include resync and errors.
4. Kernel understands that the packet is already decrypted so it does not decrypt it itself and passes it on to the user-space.

**Resync and Error Handling**

When the sequence number does not match expectations or if any other error occurs, the hardware gives control back to the SW which handles the problem.

See more about kTLS modes, resync, and error handling in the [Linux Kernel documentation](#).

**Prerequisites**

All commands in this section should be performed on host (not on BlueField) unless stated otherwise.

**Checking Hardware Support for Crypto Acceleration**

To check if the BlueField or ConnectX have crypto acceleration, run the following command from host:

```bash
host> mst start # turn on mst driver
host> flint -d <device under /dev/mst/ directory> dc | grep Crypto
```

The output should include **Crypto Enabled**. For example:

```bash
host> flint -d /dev/mst/mt41686_piconf0 dc | grep Crypto
....
;;Description = NVIDIA BlueField-2 E-Series Eng. sample DPU; 200GbE single-port QSFP56; PCIe Gen4 x16; Secure Boot Disabled; Crypto Enabled; 16GB on-board DDR; 1GbE OOB management
....
```
Kernel Requirements

- Operating system must be either:
  - FreeBSD 13.0+
  - A Linux distribution built on Linux kernel version 5.3 or later for Tx support and version 5.9 or later for Rx support. We recommend using the latest version when possible for the best available optimizations.

**Note**

TIS Pool optimization is added to Linux kernel version 6.0. Instead of creating TIS per new connection, unused TIS from previous connection, will be recycled. This will improve Tx connection rate. No further installations required beyond installing the kernel itself.

- Check the current kernel version on the host. Run:

  ```
  host> uname -r
  ```

- The kernel must be configured to support TLS by setting the options `TLSDEVICE` and `MLX5_TLS` to `y`. To check if TLS is configured, run:

  ```
  host> cat /boot/config-$(`uname -r`) | grep TLS
  ```

  **Example output:**

  ```
  host> cat /boot/config-5.4.0-121-generic | grep TLS
  ...
  CONFIG_TLS_DEVICE=y
  ```
If the current kernel does not support one of the options, you can change the configurations and recompile, or build a new kernel.

**Note**

Follow the build instructions provided with the kernel provider.

Schematic flow for building a Linux kernel:

1. Enter the Linux kernel directory downloaded (usually in /usr/src):

   ```
   host> make menuconfig # Set TLSDEVICE=y and MLX5_TLS=y in options. Setting location in the menu can be found by pressing '/' and typing 'setting'.
   host> make -j <num-of-cores> && make -j <num-of-cores> modules_install && make -j <num of cores> install
   ```

2. Update the grub to the new configured kernel then reboot.

**Configurations and Useful Commands**

**TLS Setup**
Finding NVIDIA Interfaces

```
host> mst start  # if mst driver is not loaded.
host> mst status -v
```

NVIDIA's netdev interfaces are found be under the `NET` column.

For example:

```
host> mst status -v
....
DEVICE_TYPE MST PCI RDMA NET NUMA
BlueField2(rev:0) /dev/mst/mt41686_pciconf0.1 b1:00.1 mlx5_1 net-ens5f1 1
BlueField2(rev:0) /dev/mst/mt41686_pciconf0 b1:00.0 mlx5_0 net-ens5f0 1
```

In this example, the interfaces `ens5f1` and `ens5f0` are NVIDIA's netdev interfaces.
Configuring TLS Offload

- To check if the offload option is on or off, run:

  ```
  host> ethtool -k $iface | grep tls
  ```

  Example output:

  ```
  tls-hw-tx-offload: on
  tls-hw-rx-offload: off
  tls-hw-record: off [fixed]
  ```

  **Note**

  `tls-hw-record` is not required for the device as kTLS does not support "HW Record" mode.

- To turn Tx offload on or off:

  ```
  host> ethtool -K $iface tls-hw-tx-offload <on | off>
  ```

- To turn Rx offload on or off:

  ```
  host> ethtool -K $iface tls-hw-rx-offload <on | off>
  ```
Configuring OVS Bridge on BlueField

When the host is connected to a BlueField device, an OVS bridge must be configured on the BlueField so traffic passes bidirectionally from host to uplink. If no OVS bridge is configured, the host is isolated from the network (see diagram above).

Note

On BlueField image version 3.7.0 or higher the default OVS configuration can be used without additional modifications.

To configure the OVS bridge on BlueField, run the following commands on BlueField:

dpu> for br in $(ovs-vsctl list-br); do ovs-vsctl del-br $br; done # erasing existing bridges
dpu> ovs-vsctl add-br ovs-br0 && ovs-vsctl add-port ovs-br0 p0 && ovs-vsctl add-port ovs-br0 pf0hpf
dpu> ovs-vsctl add-br ovs-br1 && ovs-vsctl add-port ovs-br1 p1 && ovs-vsctl add-port ovs-br1 pf1hpf
dpu> ovs-vsctl set Open_vSwitch . other_config:hw-offload=true && systemctl restart openvswitch-switch

Where p0/p1 are the uplink interfaces and pf0hpf/pf1hpf are the interfaces facing the host.

Common Use Cases

OpenSSL

OpenSSL is an all-around cryptography library that offers open-source application of the TLS protocol. It is the main library for using kTLS and other applications since Nginx depends on it as their base library.

Note
kTLS is supported only in OpenSSL version 3.0.0 or higher, and only on the supported kernel versions. The supported OpenSSL version is available for download from distro packages, or it can be downloaded and compiled from the OpenSSL GitHub.

⚠️ **Warning**

Many modules depend on OpenSSL. Changing the default version may cause problems. Adding `--prefix=/var/tmp/ssl --openssldir=/var/tmp/ssl` in the `./Configure` command below may prevent the built OpenSSL from becoming the default one used by the system. Make sure the directory of the OpenSSL you build manually is not located in any paths listed in the PATH environment variable.

1. Check the version of the default OpenSSL:

   ```
   host> openssl version
   ```

2. Follow OpenSSL installation instructions from OpenSSL's supplied guides. During the configuration process, make sure to set the `enable-ktls` option before building it by running it from within the OpenSSL directory (works in version 3.0 and higher). For example:

   ```
   host> ./Configure linux-$({uname -p}) enable-ktls --prefix=/var/tmp/ssl --openssldir=/var/tmp/ssl
   # Add "threads" as well for multithread support
   ```
3. Check if kTLS is enabled in OpenSSL by running the following command from within the OpenSSL directory, and check whether \textit{kts} is listed under \textit{Enabled features}:

```
host> perl configdata.pm --dump | less
```

If OpenSSL has been downloaded manually, the OpenSSL executable would be located in the 
\texttt{/<openssl-dir>/apps/} directory. For example, checking the version from within OpenSSL directory is done using the command \texttt{./apps/openssl version}.

\begin{itemize}
  \item \textbf{Note} \hfill \\
  Installing a new OpenSSL requires recompiling user tools that were configured over OpenSSL (e.g., Nginx).
\end{itemize}

\begin{itemize}
  \item \textbf{Note} \hfill \\
  In OpenSSL's master source code, there is a feature "Support for kTLS Zero-Copy sendfile() on Linux" (Zero-Copy commit). If the Zero-Copy option is set, \texttt{SSL\_sendfile()} uses the Zero-Copy TX mode which means that the data itself is not copied from the user space to Kernel space. This gives a performance boost when used with kTLS hardware offload. Be aware that invalid TLS records may be transmitted if the file is changed while being sent.
\end{itemize}

\section*{Nginx}

Nginx is a free and open-source software web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. Nginx can be configured to depend
on OpenSSL library and therefore Nginx could have the great advantages of TLS HW-offload on ConnectX-6 Dx, ConnectX-7 or the DPU.

Prerequisites

Refer to the OpenSSL section for setting OpenSSL.

Configuration

1. Install dependencies. For Ubuntu distribution, for example:

   host> apt install libpcre3 libpcre3-dev

2. Clone Nginx's repository and enter directory:

   host> git clone https://github.com/nginx/nginx.git && cd nginx

3. Configure Nginx components to support kTLS:

   host> ./auto/configure --with-openssl=/<insert_path_to_openssl_directory> --with-debug --with-http_ssl_module --with-openssl-opt="enable-ktls -DOPENSSL_LINUX_TLS -g3"

4. Build Nginx:

   host> make -j <num of cores> && sudo make -j <num-of-cores> install

Note
5. Add the following lines to the end of the `/usr/local/nginx/conf/nginx.conf` file (before the last closing bracket):

```nginx
server {
    listen 443 ssl default_server reuseport;
    server_name localhost;
    root /tmp/nginx/docs/html/;

    include /etc/nginx/default.d/*.conf;
    ssl_certificate /usr/local/nginx/conf/cert.pem;
    ssl_certificate_key /usr/local/nginx/conf/key.pem;
    ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256;
    ssl_protocols TLSv1.2;

    location / {
        index index.html;
    }

    error_page 404 /404.html;
    location = /40x.html {
    }

    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
    }
}
```

6. Notice that the key and certificate of the Nginx server should be located in `/usr/local/nginx/conf/`. Therefore, after creating a key and certificate (as mentioned in section "Adding Certificate and Key") they should be copied to the aforementioned directory:

```
host> cp key.pem /usr/local/nginx/conf/ && cp cert.pem /usr/local/nginx/conf/
```
7. To run Nginx:

```
host> cd nginx && objs/nginx
```

This command starts Nginx Server in the background.

**Stopping Nginx**

```
host> pkill nginx
```

**Wrk – Client**

A simple client for requesting Nginx's server is "wrk". It can be installed by running the following:

```
host> git clone https://github.com/wg/wrk.git && cd wrk/ && make -j <num-of-cores>
```

**Using Wrk**

The following is an example of using the wrk client to request the page index.html from the Nginx server in address 4.4.4.4 (run within wrk's directory):

```
host> taskset -c 0 ./wrk -t1 -c10 -d30s https://4.4.4.4:443/index.html
```
Testing the kTLS offload (with or without hardware offload) is in the same manner as mentioned in section "Testing kTLS". TBD

Testing Offload via OpenSSL

This chapter demonstrates how to test the kTLS hardware offload.

Note

Make sure to refer to section "OpenSSL" before proceeding.

TLS Testing Setup

For testing purposes, a server and a client are required. The testing section only tests a single setup of a host and BlueField-2 or a host ConnectX which will participate either as a server or as a client. Setting a back-to-back setup of the same kind and installing the same OpenSSL version can help avoid misconfigurations. Nevertheless, it is required to have the same OpenSSL version on both the client and server.

Make sure the desired kTLS is configured as detailed in section "Configuring TLS Offload". To test hardware offload, make sure tls-hw-tx-offload and/or tls-hw-rx-offload are on. To test kTLS software mode, make sure to turn them off.

In addition, make sure both hosts (server and client) can communicate bidirectionally through ConnectX or BlueField. One can set the interface that supports the offload (on the host) with an IP, in same subnet. Make sure that when using BlueField, an OVS bridge is set on BlueField as shown in "Configuring OVS Bridge on BlueField".
Adding Certificate and Key

The server side should create a certificate and key. The client can also use a certificate, but it is not necessary for this test case. Run the following command in the installed OpenSSL directory and fill in all the requested details:

```
host> openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365 -nodes
```

The following files are created:

- `key.pem` – private-key file used to generate the CSR and, later, to secure and verify connections using the certificate

- `cert.pem` – certificate signing request (CSR) file used to order your SSL certificate and, later, to encrypt messages that only its corresponding private key can decrypt

**Note**
The server side should be run before client side so that client’s request are answered by server.

Running Server Side

The following example works on OpenSSL version 3.1.0:

```
host> openssl s_server -key key.pem -cert cert.pem -tls1_2 -cipher ECDHE-RSA-AES128-GCM-SHA256 -accept 443 -ktls
```

ℹ️ **Note**

Notice the `-ktls` flag.

ℹ️ **Note**

Refer to official OpenSSL documentation on `s_server` for more information.

In this example, the key and certificate are provided, the cipher suite and TLS version are configured, and the server listens to port 443 and is instructed to use kTLS.

Running Client Side

The following example works on OpenSSL version 3.1.0:
host> openssl s_client -connect 4.4.4.4:443 -tls1_2

Where 4.4.4.4 is the IP of the remote server.

**Note**

Refer to official OpenSSL documentation on `s_client` for more information.

---

**Testing kTLS**

After the connection is established (handshake is done), a prompt will open and the user, both on the client and server side, can send a message to other side in a chat-like manner. Messages should appear on the other side once they are received.

The following example checks kTLS hardware offload on the tested setup by tracking Rx and Tx TLS on device counters:

```bash
host> ethtool -S $iface | grep -i 'tx_tls_encrypted\|rx_tls_decrypted' # ($iface is the interface that offloads)
```

To check kTLS over kernel counters:

```bash
host> cat /proc/net/tls_stat
```

Output example:

**Note**
The comments are not part of the output and are added as explanation.

```
host> cat /proc/net/tls_stat
TlsCurrTxSw 0 # Current Tx connections opened in SW mode
TlsCurrRxSw 0 # Current Rx connections opened in SW mode
TlsCurrTxDevice 0 # Current Tx connections opened in HW-offload mode
TlsCurrRxDevice 0 # Current Rx connections opened in HW-offload mode
TlSTxSw 2323828 # Accumulated number of Tx connections opened in SW mode
TlsRxSw 1 # Accumulated number of Rx connections opened in SW mode
TlsTxDevice 12203652 # Accumulated number of Tx connections opened in HW-offload mode
TlsRxDevice 0 # Accumulated number of Rx connections opened in HW-offload mode
TlsDecryptError 0 # Failed record decryption (e.g., due to incorrect authentication tag)
TlsRxDeviceResync 0 # Rx resyncs sent to HW's handling cryptography
TlsDecryptRetry 0 # All Rx records re-decrypted due to TLS_RX_EXPECT_NO_PAD misprediction
TlsRxNoPadViolation 0 # Data Rx records re-decrypted due to TLS_RX_EXPECT_NO_PAD misprediction
```

**Note**

More information about the kernel counters can be found in the **Statistics** section of the Kernel TLS documentation.

---

**Optimizations over kTLS**

**XLIO**
The NVIDIA accelerated IO (XLIO) software library boosts the performance of TCP/IP applications based on Nginx (e.g., CDN, DoH) and storage solutions as part of SPDK. XLIO is a user-space software library that exposes standard socket APIs with kernel-bypass architecture, enabling a hardware-based direct copy between an application’s user-space memory and the network interface. In particular, XLIO can boost the performance of applications that use the kTLS hardware offload as OpenSSL and Nginx. Read more about XLIO in the NVIDIA XLIO Documentation and XLIO TLS HW-offload over kTLS in the TLS HW Offload section.

**Note**

Even though XLIO is a kernel-bypass library, the kernel must support kTLS for the bypass to work properly.

**Performance Tuning Options**

TLS offload performance is related to how fast data can be pumped though the offload engine. In the case of user space applications, certain system configurations can be tuned to optimize its performance.

The following are items that can be tuned for optimal performance, mainly focusing on dedicating the server’s work to the NUMA, or non-uniform memory access, cores:

**Note**

Non-uniform memory access (NUMA) cores are cores with a dedicated memory for each of them, granting cores fast access to their own memory and slower access to others’. This architecture is best for scenarios when it is not necessary to share memory between cores.
1. Add NUMA cores of the NIC to the `isolcpus` kernel boot arguments for each server so that the kernel scheduler does not interrupt the core's running user thread. The following are examples of adding commands:

1. Identify the NIC NUMA node (see NUMA column):

```
host> mst status -v
DEVICE_TYPE      MST      PCI    RDMA   NET     NUMA
ConnectX6DX(rev:0) /dev/mst/mt4125_pciconf0 41:00.0  mlx5_0   net-enp65s0f0np0
```

2. Identify the cores of the NIC NUMA node using the NUMA node number acquired from the previous output:

```
host> lscpu | grep "NUMA node1"
NUMA node1 CPU(s): 1,3,5,7,9,11,13,15,17,19,21,23
```

3. Add the NIC NUMA cores to a grub file (e.g., `/etc/default/grub`) by adding the line `GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=<NUMA-cores-from-previous-output>"`. For example:

```
GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=1,3,5,7,9,11,13,15,17,19,21,23"
```

4. Update grub:

```
host> sudo update-grub
```

5. Reboot and check that the configuration has been applied:

```
host> cat /proc/cmdline
```
2. Disable `irqbalance` service:

```
host> service irqbalance stop
```

Note

Interrupt request, or IRQ, determines what hardware interrupts arrive to each core.

3. Run `set_irq_affinity.sh` to redistribute IRQs to various cores.

Note

The script is within MLNX_OFED's sources:

1. You can find it in MLNX_OFED downloads.
2. Under "Download" select the correct version and download the "SOURCES" .tgz file.
3. Extract the .tgz.
4. Under SOURCES, extract the mlnx_tools.

You should find both files `set_irq_affinity.sh` and its helper file `common_irq_affinity.sh` under the sbin directory.
4. Set the interface RSS to the number of cores to use:

```bash
host> ./set_irq_affinity.sh <ConnectX_or_BlueField_network_interface>
```

5. Set the interface queues for number of cores to use:

```bash
host> ethtool -X <ConnectX_or_BlueField_network_interface> equal <number_of_isolcpus_cores>
host> ethtool -L <ConnectX_or_BlueField_network_interface> combined <number_of_isolcpus_cores>
```

6. Pin the application with `taskset` to the `isolcpus` cores used. For example:

```bash
host> taskset -c 1,3,5,7,9,11,13,15,17,19,21,23 openssl s_server -key key.pem -cert cert.pem -tls1_2 -cipher ECDHE-RSA-AES128-GCM-SHA256 -accept 443 -ktls
```

**Additional Reading**

- [Linux kernel TLS documentation](#)
- [Linux kernel TLS offload documentation](#)
- [Autonomous NIC offloads research paper](#)