TLS Offload to Network Devices

Boris Pismenny, Ilya Lesokhin, Liran Liss, Haggai Eran

Mellanox
Yokneam, Israel
{borisp, ilyal, liranl, haggaie}@mellanox.com

Abstract

Encrypted Internet traffic is becoming the norm, spearheaded by the use Transport Layer Socket (TLS) to secure TCP connections. This trend introduces a great challenge to data center servers, as the symmetric encryption and authentication of TLS records adds significant CPU overhead. New CPU capabilities, such as the x86 AES-NI instruction set, alleviate the problem, yet encryption overhead remains high. Alternatively, cryptographic accelerators require dedicated HW, consume significant memory bandwidth, and increase latency. We propose to offload TLS symmetric crypto processing to the network device. Our solution does not require a TCP Offload Engine (TOE). Rather, crypto processing is moved to a kernel TLS module (kTLS [5,6]), which may leverage inline TLS acceleration offered by network devices. Transmitted packets of offloaded TLS connections pass through the stack unencrypted, and are processed on the fly by the device. Similarly, received packets are decrypted by the device before being handed off to the stack. We will describe the roles and requirements of the kTLS module, specify the device offload APIs, and detail the TLS processing flows. Finally, we will demonstrate the potential performance benefits of network device TLS offloads.

Keywords

TLS Offload, Tx Offload, Network Devices, TLS, Crypto, TCP.

Introduction

In today's networks, Transport Layer Security (TLS) is widely used to securely connect endpoints both inside data centers [1] and on the internet. TLS encrypts, decrypts, and authenticates its data, but these operations incur a significant overhead on the server.

Fixed function hardware accelerators are known to give improved performance and greater power-efficiency when compared to running a software implementation on a general purpose CPU. Cryptographic operation such as those used in TLS are very suitable for such hardware accelerators but they are not widely used in the context of networking. We believe that the reason is that the offload model is not good enough.

Existing solutions fall into four categories:

- **TLS Proxy** – A middlebox [2] is used to decrypt/encrypt all incoming/outgoing traffic. The middlebox is running a TCP connection against trusted machines and a TLS connection against untrusted machines, reducing the load on the trusted machine. However, if applied inside the data center, some traffic remains unprotected.

- **TOE** – TCP offload engines have been around for a while [3]. A TOE could run a full TLS offload as well reducing PCI traffic and freeing CPU cycles even further. However, the TCP stack of TOE devices is inflexible, and hard to debug and fix when compared to a software TCP implementation. Moreover, with full TLS offload, security vulnerabilities could remain unfixed for a long time.

- **Crypto offload PCI card** – A dedicated PCIe card to accelerate cryptographic operations, such as [4]. In the case of a PCIe card performing encryption/decryption operation, the data is sent towards the card over PCIe. It is then modified and sent back for further processing. This can add a significant latency to the operation, and is relatively expensive in cost, power, PCIe lanes utilization and CPU utilization.

- **TLS in the kernel** – Kernel TLS [5][6] is kernel module for performing the bulk symmetric encryption of TLS records by the kernel instead of using a user space library. It facilities using sendfile for TLS connections. Moreover, where previously data was copied once during encryption and once again to be sent by TCP, using this approach encryption and data copy from user-space to the kernel become a single operation. This approach can leverage the x86 AES-NI instruction set for accelerating AES operations.

Motivation

To motivate this work, we conducted a simple experiment using 2 machines connected back-to-back running a TLS session between the 2 machines. We run this experiment twice: once using the AES128-GCM ciphersuite and once again using the NULL ciphersuite. We obtained the results

presented in Table 1. We observe that the CPU utilization was reduced by 85%, while throughput has increased, reaching 10.4Gbps from 6.48Gbps. We argue that our model for TLS offload would behave similarly to the NULL ciphersuite on the CPU. We therefore believe that it would benefit the community to offload TLS encryption to hardware.

Cipher suite (symmetric crypto-digest)	CPU time (user/system/usage)	Bandwidth
AES128-GCM	0.54 sec/0.78 sec/94%	6.48 Gbit/s
	85% reduced CPU	
NULL	0.08 sec/0.61 sec/81%	10.4 Gbit/s

Table 1. Bandwidth and CPU comparison between a TLS socket using the AES128-GCM ciphersuite and a TLS socket with the NULL ciphersuite.

Model and Software Stack

In this paper, we propose a model (see Figure 1) where the payload of network packets is transformed in-place by the network device. This model retains all the benefits of using a robust software network stack while offloading the crypto data crunching to the device. The data need be sent only once to the network device, saving PCIe BW and latency.

In the proposed model, the keys used by the TLS socket are offloaded to the NIC to which the connected socket is routed. The socket is marked as offloaded. From this moment onward packets of this TCP socket will be encrypted by the device. The device expects software to frame TLS packets, including TLS headers and trailers, while skipping the actual encryption. The software must place those packets into the offloaded socket. The rest of the software stack remains unchanged: existing TCP/IP and memory flows are unaffected. Existing TCP features such as congestion control, retransmission, memory management, and other enhancements in the TCP stack are all left unchanged.

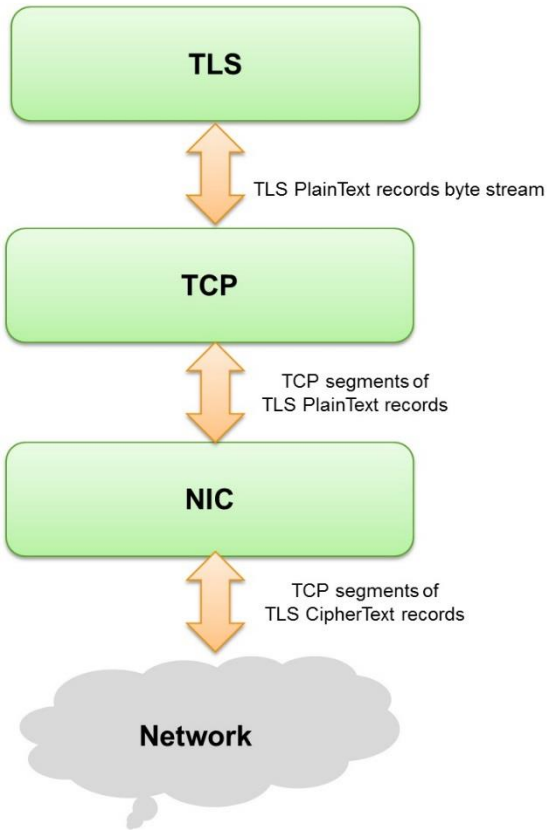

Figure 1. Kernel software stack for TLS offload. Kernel TLS provides plaintext records to TCP, the TCP/IP stack segments the records. Finally, the NIC encrypts plaintext records inside the TCP segments.

TLS Offload API

In this section we outline the data path and the control path for the TLS offload for the transmit side and suggest an API. Ideally, in order to make hardware and software simple, packets should be encrypted independently, as in the case of IPsec[7], QUIC[8] and DTLS[9]. However, in TLS *each record* is encrypted independently. A TLS record may be spread over multiple TCP segments (see Figure 2), while a TCP segment might also contain multiple TLS records. Thus, intermediate record state between packets of a single session must be tracked by the hardware to encrypt subsequent packets which are part of a TLS record that started on a previous TCP segment.

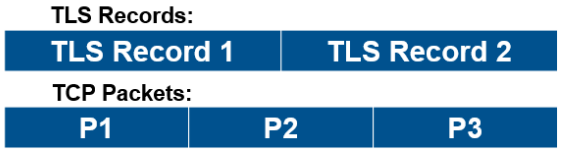

Figure 2. TLS Records split among multiple TCP packets.

Control Path

The control path is based on an extension of the kTLS[5][6] control plane, where an additional flag is added to indicate that offload is required.

In response to an offload request, kTLS calls ktls_dev_add, a new NDO, for the netdevice used by that socket. kTLS provides the socket and the crypto parameters to ktls_dev_add as input. If the device can offload this TLS session, the function returns success and the socket is marked as offloaded by setting "sk->sk_tls_offload" for that socket. From this moment onwards, any payload sent over that socket is expected to be plaintext. The device will track TCP sequence numbers, encrypt and authenticate all packets sent from this socket.

The sk_destruct function of the TCP socket is replaced to free resources related to TLS in the socket layer. Similarly, kTLS goes on to call another new NDO called ktls_dev_del, in order to free device driver and hardware resources.

Data Path

The data path consists of a fast path and a slow path. The following pseudo code is performed by the device driver for each packet:

1. Check packet belongs to offloaded socket (skb->sk->sk_tls_offload != 0)
    a. If failed, goto normal packet transmit.
2. Check packet TCP sequence number against expected TCP sequence number
    a. If failed, perform resync (rebuild hardware TLS context for the given packet)
3. Send packet to be encrypted, authenticated and sent to the network by the device.

First, the driver checks whether the socket is offloaded; otherwise, normal packet processing takes over. Second, the TCP sequence number is checked against the expected TCP sequence number initialized when the socket was offloaded. If the sequence numbers do not match, then the slow path (resync) is triggered. Finally, the packet is sent to the device to be encrypted, authenticated and sent to the wire.

Resync Flow

As explained in above, the hardware has to track the crypto context between TCP packets of an offloaded socket in order to process them. It follows that, when the device receives a TCP sequence number it does not expect (e.g. during retransmission), then additional information is required for the hardware to encrypt and authenticate the packet.

To resynchronize hardware state, the prefix of the TLS record is needed by the network device. This is sufficient because each TLS record is encrypted and authenticated independently of other TLS records[1].

For example, in Figure 3, after transmitting packets P1-7, packet P5 is retransmitted. The device has the state required to encrypt packet P8. However, to encrypt packet P5 the payload of TLS record 2 has to be passed to hardware.

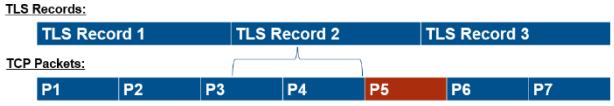

Figure 3. Packet P5 is retransmitted triggering the resync flow.

TLS record 2 is split among 3 TCP packets, some of which could have been acknowledge and released from memory. To enable the resync flow, the payload of partially acknowledged TLS records must not be released from memory. To prevent this, kTLS will take an additional reference on all payload pages and TCP will call kTLS during tcp_clean_rtx_queue() to release acknowledged TLS records.

Additionally, kTLS provides a mapping from TCP sequence numbers to the TLS record payload. This mapping is exposed to the device driver. The driver uses this mapping during the resync flow. For example, in Figure 4, the resync flow for SKB 2 queries kTLS for the mapping of the TCP sequence number of SKB2, in order to acquire the payload of TLS Record 1.

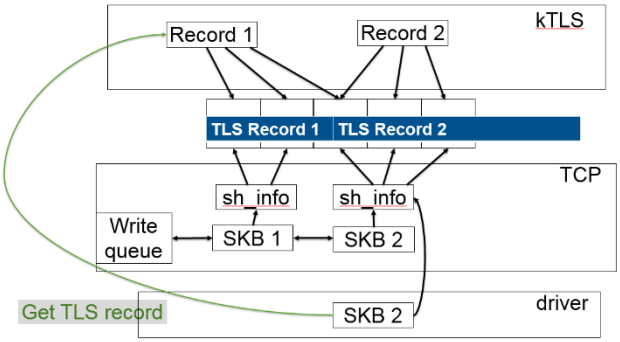

Figure 4. The use of a mapping from SKB2 TCP sequence number to the payload of TLS record 1.

Zero-Copy Sendfile

With crypto offload it would have been possible to enable zero-copy sendfile functionallity, which is not possible without crypto offload. Similarly to the way sendfile works with TCP sockets. However, with TLS there is an undesirable side-effect that occurs during retransmissions. If the data being transmitted is dropped and the new data is

[1] This is at least true for GNUTLS where the TLS record sequence number and the IV are the same.

different, then the crypto offload would rencrypt the new data as part of a previously transmitted TLS record. Resulting in authentication tag failure on the receiving side, as described in figure 5. Another problem, is that with AES-GCM, a counter mode cipher, the data retransmitted will use the same counter with different plaintext. This enables an attacker to XOR the ciphertext transmitted with ciphertext retransmitted, resulting in the elimination of the keystream, i.e. a XOR between original plaintext and new plaintext. In our design, we decided against supporting zero-copy sendfile with TLS to avoid these problems.

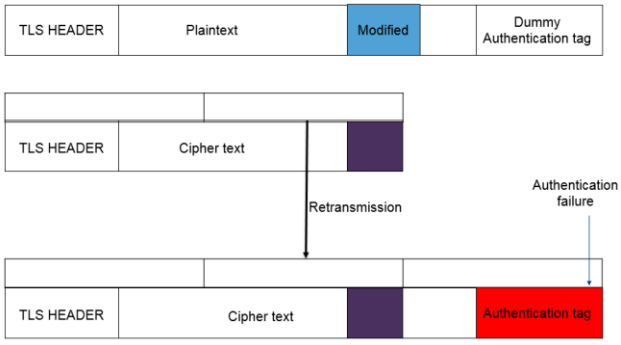

Figure 5. Authentication tag failure due to zero-copy sendfile retransmission with TLS crypto offload.

Evaluation

In Table 2, we present preliminary results of transmit side TLS offload. The test setup has two machines connected back-to-back. We compared between TCP, user space TLS (GNU TLS), kernel space TLS (kTLS) and kernel space TLS with offload. We used the AES128-GCM ciphersuite in all scenarios. We did not decrypt traffic on the receiving side, to prevent it from becoming a bottleneck. We used small TLS records of 1457 bytes.

Using kTLS with offload we obtain 8.8Gbps, which is twice the throughput obtained by kTLS.

Metric	TCP	GNU TLS	kTLS	kTLS + Offload
Throughput	10-23Gbps	5.1Gbps	4.3Gbps	8.8Gbps
CPU	100%	100%	100%	100%

Table 2. Preliminary performance evaluation of TLS offload.

Conclusion

We suggest a kernel API for TLS offloading, providing an initial performance evaluation. The TLS offload improves performance by at least 2x over current state-of-the-art kernel implementation, reducing per packet CPU overhead and enabling the use of encryption in high throughput.

Future Work

For receive side TLS offload, there are additional challenges. Packets cannot be delayed by hardware, thus some packets might be received unencrypted, while others could be decrypted. As in the transmit case, HW must be resynchronized following jumps in TCP sequence numbers.

References

1. “The Fully Encrypted Data Center”, Oracle Technical White Paper, accessed September 22, 2016, http://www.oracle.com/technetwork/server-storage/hardware-solutions/fully-encrypted-datacenter-2715841.pdf

2. “Server Farm Security in the Business Ready Data Center Architecture”, Cisco design guide, Chapter 6 “Catalyst SSL Services Module Deployment in the Data Center with Back-End Encryption” http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/ServerFarmSec_2-1/ServSecDC/DC_Pref.html

3. “Why TOE is bad?”, accessed September 22, 2016. https://wiki.linuxfoundation.org/networking/toe

4. “Intel QuickAssist”, accessed September 22, 2016. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/quickassist-adapter-8950-brief.pdf

5. Optimizing TLS for “High-Bandwidth” Applications in FreeBSD, R. Stewart, et. al. accessed September 22, 2016. https://people.freebsd.org/~rrs/asiabsd_2015_tls.pdf

6. Crypto Kernel TLS socket, D. Watson, accessed September 22, 2016. https://lwn.net/Articles/665602/

7. RFC 4303: IP Encapsulation Security Payload (ESP), S. Kent, accessed November 1, 2016, https://www.ietf.org/rfc/rfc4303.txt

8. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2 draft-hamilton-early-deployment-quic-00, J. Iyengar, et. al. accessed November 1, 2016. https://tools.ietf.org/html/draft-hamilton-early-deployment-quic-00

9. RFC 6347: Datagram Transport Layer Security Version 1.2, E. Rescorla, accessed November 1, 2016. https://tools.ietf.org/html/rfc6347