KTLS: Linux Kernel Transport Layer Security

1st Dave Watson
Facebook
San Francisco, USA
davejwatson@fb.com

Abstract

Transport Layer Security (TLS) is a widely-deployed protocol used for securing TCP connections on the Internet. TLS is also a required feature for HTTP/2, the latest web standard. In kernel implementations provide new opportunities for optimization of TLS. This paper explores a possible kernel TLS implementation, as well as the kernel features it enables, such as sendfile(), BPF programs, and hardware TLS offload. Our implementation saves up to 7% CPU copy overhead and up to 10% latency improvements when combined with the Kernel Connection Multiplexor (KCM).

Keywords

TLS, DTLS, Linux, security, performance, sockets, OpenSSL, offload

Introduction

Transport Layer Security [2] (TLS) and Datagram Transport Layer Security (DTLS) are building blocks for transport security on the modern internet. The latest version of the Hypertext Transfer Protocol [1] (HTTP/2) specifies the use of TLS. It provides both encryption and authentication of TCP connections, but comes with a CPU cost. TLS and DTLS consists of two primary operations: first a TLS handshake is performed to negotiate a secure symmetric encryption algorithm and keys, and then TLS symmetric encryption is performed on TLS records. TLS has several types of records, including data records and control records.

DTLS is a UDP based encryption protocol. Most elements of TLS are reused, with minor changes to support the stateless datagram messages. KTLS supports DTLS messages, and implements a sliding window for replay protection.

Facebook encrypts the majority of its external traffic over HTTPS. Internal traffic is also encrypted if there is enough available CPU. Internal traffic is served over Apache Thrift [6], Facebook's RPC framework, and is also encrypted with TLS.

Facebook's HTTP/2 web servers and RPC servers both function similarly. One thread per core is dedicated to an epoll() event loop. When epoll_wait() returns a list of active connections, read() and write() are used to read in the request and then send the static or dynamic response. In a TLS enabled service, OpenSSL's SSL_read and SSL_write primitives

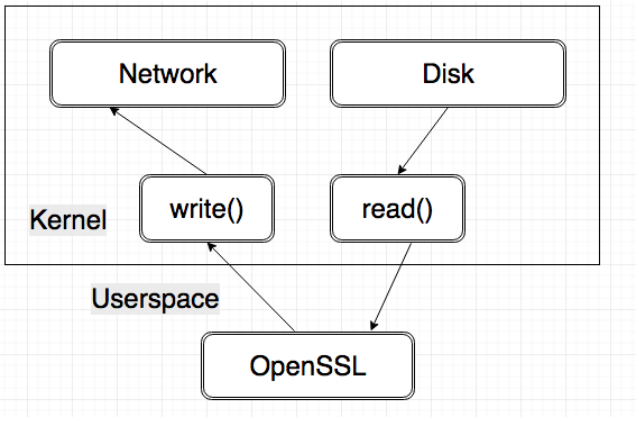

Figure 1: Standard web server with OpenSSL

are used instead. Since OpenSSL is a user space library, all data must be in user space to be encrypted. Facebook's current SSL overheads result in approximately 2% of total CPU spent on copy from/copy to user space due to encryption, and approximately 10% of total CPU is spent on encryption and decryption routines on machines that make heavy use of TLS.

To eliminate the overhead due to copies, Facebook has investigated using the sendfile() or splice() system calls to send static content directly from disk to the network, without any copies through user space. Unfortunately, do to our wide deployment of TLS, this hasn't been possible, due to the need to encrypt data in user space. Facebook has also experimented with the Kernel Connection Multiplexor [3] (KCM), and found reductions in tail latencies. Unfortunately it would require access to the unencrypted bytes in the kernel.

Encrypting the data in the kernel results in ideally zero copies, with only the encryption taking the bulk of the CPU usage. User space only needs to inform the kernel of which data needs to be encrypted. The Linux kernel has an existing crypto interface, af_alg, that can be used to do bulk encryption, but additional overhead is required to add the framing from user space. It also lacks an efficient interface to NIC hardware encryption offloads.

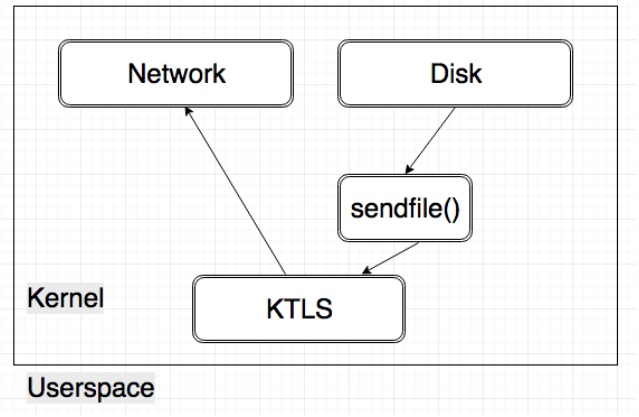

Figure 2: Server using KTLS and sendfile

Approach

Facebook, in collaboration with RedHat, have implemented a Linux kernel TLS socket. To avoid putting unnecessary complexity in the kernel, the TLS handshake is kept in user space. A full TLS connection using the socket is done using the following scheme:

- Call connect() or accept() on a standard TCP file descriptor.
- A user space TLS library is used to complete a handshake. We have tested with both GnuTLS and OpenSSL.
- Create a new KTLS socket file descriptor.
- Extract the TLS Initialization Vectors (IVs), session keys, and sequence IDs from the TLS library. Use setsockopt on the KTLS fd to pass them to the kernel.
- Use standard read(), write(), sendfile() and splice() system calls on the KTLS fd.

Upon receipt of a non-data TLS message (a control message), the KTLS socket returns an error, and the message is instead left on the original TCP socket. The KTLS socket is automatically unattached. Transfer of control back to the original encrypted FD is done by calling getsockopt to receive the current sequence numbers, and inserting them in to the TLS library. Example:

```
if (read(...) < 0) {
  getsockopt(tls_fd,
    AF_KTLS,
    KTLS_GET_IV_RECV,
    ssl_->s3->read_sequence,
    &optlen);
  /* Similar for IV_SEND */
  SSL_read(tcp_fd, ...);
}
```

In this scheme, the TLS library is used to handle the control messages and do the handshake, and does not need to be modified. It can maintain control of the original TCP fd, while unencrypted data flows through the KTLS socket. The user space application only needs to handle application data, and use standard socket system calls.

Most of the complexity in this scheme is the buffer management between the two FDs, and handing off control when control messages are received. While it is reasonable to not handle most control messages – Facebook's servers shutdown the connection on receipt of a control message – the client sending the control message is still expecting a response, so to enable a clean shutdown, control must still be passed back to the original TLS handshake library to send the appropriate response. To help manage this complexity, the strparser [9] library was developed to manage parsing TCP buffers as datagram messages.

Crypto Framework Changes

The Linux crypto framework already contains the two symmetric ciphers included in the TLS 1.3 draft, GCM-AES and ChaCha/Poly. Current Intel chipsets support AESNI instruction set, which allows fast encryption and decryption routines using GCM-AES. These routines were already implemented in assembly for IPSEC, however, they required minor modifications to work with TLS. TLS's AAD data is 13 bytes, while IPSEC uses 16 bytes. An additional template was added to support the correct AAD size. The asm routines still require a full 16 bytes for padding.

The current AESNI crypto interface requires all parts of the message to be contiguous - including the AAD and tag data. This presents a slight performance hit on both send and receive – for send, we can reuse almost the same AAD every time, and don't need to reallocate space for it on encrypt. On receive, we want to strip both the AAD and tag data before passing it to user space. Minor changes to the interface can be made to support separate locations for the AAD, user data, and tag.

Kernel Connection Multiplexor

Facebook's primary motivation was to gain access to the unencrypted bytes in kernel space. KCM is used to decode the framing, and make intelligent scheduling choices, before sending the frames to user space. KTLS sockets are mapped 1:N to user space sockets, where N is the number of user space threads, which are usually mapped to cores. Using this scheme, KTLS + KCM is able to reduce the total number of thread migrations of an individual request.

Results

We have implemented the KTLS Linux kernel module, and run it in production. KTLS encryption speed is on par with user space encryption speed. The number of active file descriptors increased, due to using an FD for both the TCP socket and the KTLS socket. The services ran without any change in functionality, and only minor service code changes, for the duration of the test. A KCM socket scheme, as described above, was run on top of the KTLS socket for the service – KCM was updated to be able to directly attach to a KTLS socket. KCM results in a 10%-20% drop in the 99th percentile latency for the service. See figure 3.

We also tested the performance of using sendfile(). The base scheme of read() followed by write() of static data from a file to a socket was benchmarked, followed by several other schemes.

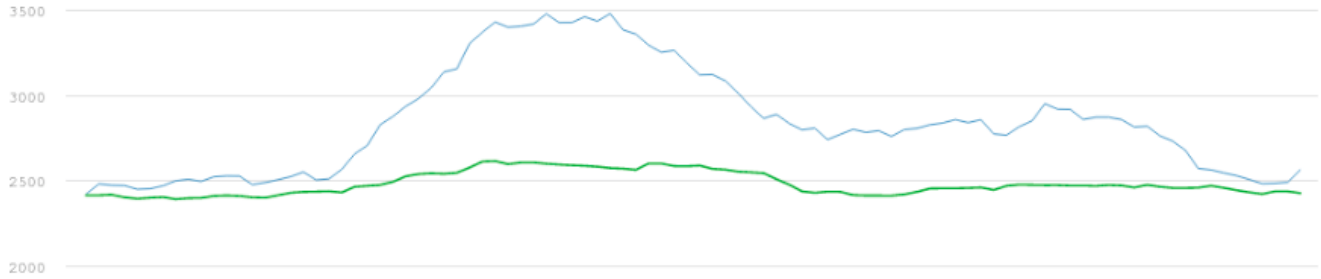

Figure 3: KTLS + KCM 99th percentile latency (green) vs. OpenSSL (blue) in ms

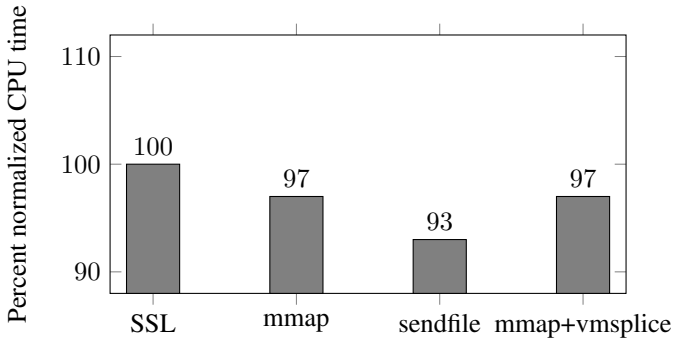

Figure 4: Various schemes to send files form disk, normalized to read(file) SSL_write(tcp fd)

SSL Use OpenSSL to read() a file to a user space buffer and SSL_write() it to a tcp fd.

mmap Mmaping a file, then calling SSL_write

sendfile Calling sendfile on a KTLS fd

mmap+vmsplice mmaping file data, using OpenSSL to frame and encrypt it, then calling vmsplice() to send the data to a tcp fd.

Multiple splice() calls can be used in place of sendfile(). Testing configuration was Intel®Xeon®CPU E5-2660 @ 2.20GHz. Test was a send of a 2GB data file from disk, normalized CPU usage for ten runs. Sender used above schemes, receiver used SSL_read.

A rough breakdown of top CPU usage for the kernel sendfile scheme:

```
72.38%  _encrypt_by_8_new8
2.49%  _encrypt_by_88
1.45%  _get_AAD_loop2_done1962
0.71%  tls_sendpage
```

and for a mmap + SSL_write scheme:

```
46.60%  gcm_ghash_clmul
25.82%  aesni_ctr32_encrypt_blocks
5.44%  copy_user_enhanced_fast_string
0.94%  tcp_sendmsg
0.76%  filemap_map_pages
```

Utilizing vmsplice we were able to remove the copy from user space to the kernel's tcp buffers, but it was replaced by VM page management overhead.

Discussion

KTLS enables access to the unencrypted bytes in the kernel. The majority of the code is related to buffer management and translation from the message-oriented TLS protocol to the stream-oriented BSD socket interface. This message-to-stream interface could be generalized to other use cases using strparser.

Enabling sendfile() for Linux results in a 7% performance win vs. a naive read/write strategy. Crypto af_alg sockets can do encryption in kernel space, but current implementations still add copy overhead in the crypto library. Strategies involving mmap or splice can improve on a naive read/write strategy, but introduce complexity and still add user space VM management overhead, such that a kernel TLS implementation still seems to have a slight advantage.

Future Directions

The feasibility of a TLS hardware offload is being investigated. A hardware offload would require a kernel interface to access it. Hardware offload would be able to instrument the ktls socket as well as the driver, to set the correct encryption keys. While a PCI or on-board offload could utilize a generic offload like the af_alg sockets with some work to remove some overhead, a NIC offload would require a socket type that ties together the sending socket and the crypto, as in ktls, to avoid additional round trip latency.

Related Work

Netflix has implemented a similar sendfile() scheme for the BSD operating system [8]. Unlike Linux, BSD does not support the asynchronous splice() primitives, so sendfile() must be used. They also needed changes to BSD's Open Crypto Framework to get the expected performance improvements, and eventually abandoned the interface altogether. [7].

Solaris supported a similar kernel ssl feature, kssl [4]. Its intended use was "to provide SSL protection even for applications which are only able to communicate in clear text over TCP", and not strictly function as a performance improvement. Unlike the proposed Linux and BSD solutions, kssl

managed the handshake and certificates. Server-side applications did not need any changes, and clear text TCP was transparently proxied.

Fridolín Pokorný and RedHat have explored the possibility of using the KTLS kernel module to optimize the performance of VPN software [5]. They include extensive performance numbers and comparisons.

Code

Code can be found at `https://github.com/ktls/af_ktls`.

References

[1] Belshe, M.; Peon, R.; and M. Thomson, E. 2015. Hypertext transfer protocol version 2 (HTTP/2). RFC 7540, BitGo and Google, Inc and Mozilla.

[2] Dierks, T., and Rescorla, E. 2008. The transport layer security (TLS) protocol version 1.2. RFC 5246, Independent and RTFM, Inc.

[3] Kernel connection multiplexor (kcm). `https://lwn.net/Articles/657999/`. Accessed: 2016-8-11.

[4] Kssl. `https://docs.oracle.com/cd/E23823_01/html/816-5175/kssl-5.html`. Accessed: 2016-8-11.

[5] Pokorný, F. 2015. Linux vpn performance and optimization. Master's thesis, Brno University of Technology, Faculty of Information Technology.

[6] Slee, M.; Agarwal, A.; and Kwiatkowski, M. 2007. Thrift: Scalable cross-language services implementation. paper, Facebook.

[7] Stewart, R., and Long, S. 2016. Improving highbandwidth tls in the FreeBSD kernel. paper, Netflix Inc.

[8] Stewart, R.; Gurney, J.-M.; and Long, S. 2015. Optimizing TLS for high-bandwidth applications in FreeBSD. paper, Netflix Inc.

[9] Strparser. `https://lwn.net/Articles/695982/`. Accessed: 2016-9-13.