Kernel TLS handshakes for HTTPS DDoS mitigation

1st Alexander Krizhanovsky, 2nd Ivan Koveshnikov
Tempesta Technologies, Inc.
Seattle, USA
ak@tempesta-tech.com, ik@tempesta-tech.com

Abstract

Efforts to offload TLS have resulted in KTLS [19]. KTLS focuses mostly on zero-copy data transfers and only offloads encryption and decryption and ignores TLS handshakes.

However, TLS handshakes are susceptible to DDoS attacks which are very effective at depleting resources. To alleviate this we ported onto the Tempesta FW a user-space TLS library to the kernel [7]. The TLS library porting took only 1 human month showing that it is quite easy to move TLS code into the kernel.

In our work we overcame memory allocation issues and other performance issues. We optimized the TLS code and merged the TLS handshakes code of the ported library with Linux kTLS. This article discusses the performance issues which we had to solve, the resulting performance improvements for TLS handshakes which we got in Tempesta FW in comparison, and compare the results with TLS handshakes with traditional HTTP accelerators. Further directions in optimization of TLS handshakes on commodity hardware also will be covered.

Keywords

TLS, TCP/IP stack, Linux kernel, DDoS mitigation, performance

TLS Handshake and DDoS

There is a special type of very efficient application-layer DDoS attacks - asymmetric DDoS attacks [2], which waste victim's resources at much higher rate than attacker's.

SSL/TLS handshake is a very expensive operation and creates a DDoS attack vector [18]. While the most time during the handshake is spent in math operations, the additional context switches and copyings, in the case of using user space cryptography library, are undesired. As a quick solution, we ported [7] mbed TLS library [8] to the kernel.

TLS 1.3 [16] reduces number of handshake messages, so I/O becomes smaller for a server. However, TLS libraries and HTTPS servers will have to support TLS 1.2 for a long time to be compatible with old clients. Since TLS 1.2 handshakes are heavier than for TLS 1.3, it's likely that TLS DDoS attacks will continue to use TLS 1.2 pretending to be outdated clients.

TLS libraries performance issues

We measured performance [14] and collected *perf* profile for emulation of TLS DDoS attack against Nginx with OpenSSL. Actual versions was used in tests: Linux kernel 4.16, Nginx 1.15 and OpenSSL 1.1.1-pre8. The thc-tls-dos [15] tool was used to stress TLS handshake subsystem. The Yandex Tank [20] tool was used to generate load with valid HTTPS requests. An Intel Xeon E3-1240 v5 was used to run the web server and an Intel Xeon E5-1650 v3 and Intel Xeon E5-2670 were used as traffic generators.

In the handshake test a client creates a new connection to the web server. When a TLS handshake is completed, the client closes the connection and makes a new connection attempt. Hundreds of such clients are used simultaneously. This allows to evaluate performance of the target web server under TLS Handshake DDoS.

Although best practices was used to configure Nginx for the better performance results [11, 9] a certain step aside is required to emulate a real-life DDoS attack. We had to disable SSL connections cache and SSL sessions tickets. The options optimize handshakes with already known clients by reusing previous parameters (session resumption), so the full TLS handshake is not performed. The more server resources are used, the more effective DDoS attack is. It's not questionable that attackers are not going to save server resources by enabling session resumption.

Another disabled option is OCSP stapling. In normal workflow the option reduces handshake time by eliminating contact the OCSP by a client to verify the server's certificate. Since clients in the tests are high speed benchmark utility, verification is not required and the only effect is increase of the handshake message size.

Cipher suites chosen by client and server has a great impact to the handshake time. Some of them stress server many times more than a client [18], so we used two with the highest priority in OpenSSl: ECDHE-RSA-AES256-GCM-SHA384 and ECDHE-ECDSA-AES256-GCM-SHA384. The only difference is server authentication algorithm: RSA or ECDSA. RSA has a longer key (3072 vs 256 for the same security strength) and it's more CPU-intensive than ECDA algorithm, but it's still popular. Some client still don't support ECC arithmetics.

We used *prime256v1* (*secp256r1*) curve for ECDSA certificate and for ECDHE key exchange. This curve is the most

used across all HTTPS server installations [17] and the most optimized one according to *openssl speed ecdsa* results.

Handling TLS handshakes

When TLS handshake is performed, cryptographic operations for server and client differs. There are two verify operations on a client side for server's certificate and the signed ECDHE key from the server. And there are one signing operation of a ECDHE key for the server. The both, the client and the server, have to generate a shared premaster secret and generate a keypair. Sign and verify operations have opposite computation cost, if one is cheap, the late is expensive. Each operation performance for single core is listed in the table below.

Algorithm	sign/s	verify/s
rsa 2048 bits	1798.2	61312.4
256 bits ecdsa (nistp256)	19716.6	16009.0

The RSA calculations is cheap for a client and expensive for a server [18], so a server can be easily brought down with a few cores of traffic generator. This asymmetry makes RSA handshake DDoS extremely useful, since RSA certificate is still provided for the most servers for backwards compability with old clients. In this preset the server was capable to handle only around *4'900 Handshakes/sec*.

AES encryption and SHA checksums are not heavily used on this stage, and efficiently implemented in both OpenSSL and Linux kernel using Intel's AVX2 and SHA Extensions, so overhed is small comparing to key exchange and server authentication calculations.

While using RSA a web server spends most time (more than 70%) on big integer arithmetics, large amount of memory allocations can be noticed. But none of Nginx can be found in perf top output. Nginx does not implement SSL handshakes on it's own, it relies on *SSL_do_handshake()* function. That is why no Nginx functions can be found in the *perf top*.

Switching to elliptic curves cryptography allowed to handle around *15'000 Handshakes/sec* (around 1'875 handshakes per core). With ECDSA cryptographic calculations are much cheaper, but the picture is the same: large amount of memory allocations, no Nginx functions.

```
9.11%  __ecp_nistz256_mul_montx
7.80%  _int_malloc
7.03%  __ecp_nistz256_sqr_montx
3.54%  sha512_block_data_order_avx2
3.05%  BN_div
2.43%  _int_free
1.89%  OPENSSL_cleanse
1.61%  malloc_consolidate
1.49%  ecp_nistz256_avx2_gather_w7
1.41%  malloc
1.24%  ecp_nistz256_point_doublex
1.20%  ecp_nistz256_ord_sqr_montx
1.01%  __ecp_nistz256_sub_fromx
1.00%  BN_lshift
0.87%  BN_num_bits_word
0.86%  bn_correct_top
0.84%  BN_CTX_get
0.81%  __memset_avx2_unaligned_erms
0.77%  free
0.74%  __ecp_nistz256_mul_by_2x
0.71%  BN_rshift
0.59%  BN_uadd
0.59%  int_bn_mod_inverse
0.54%  __memmove_avx_unaligned_erms
0.53%  aesni_ecb_encrypt
```

Allocations took place not only at receiving or sending packets, but during most of the steps of *SSL_do_handshake()* function. Additionally *OPENSSL_cleanse()* functions wipes unused memory to protect sensitive information before releasing the memory. *BIO_*()* operations which controls sending and receiving packets takes 4.6% of time.

The bottle neck of handshakes are cryptographic calculations and memory allocations. Common technic to speed up the first is to delegate more calculations to the clients. As shown above computational cost of ECDSA signing operation is smaller than for RSA, but verify operation is more expensive. Thus ECDSA has a better performance. But further optimisations of cryptographic primitives are required to increase performance. I.e. OpenSSL uses AVX2 instruction set in Montgomery Multiplication for RSA and for *secp256r1* curve computations.

Handling established TLS connections

Once a TLS connection established, performance overhead, especially for small data files [14], moves to data copies between Nginx and OpenSSl, OpenSSL and the kernel. Several data copies are performed during message sending: in *ngx_ssl_send_chain()* to construct the 16K buffer for OpenSSL, in *do_ssl3_write()*, and at last the packed is copied to the kernel space. Around 11.5% of time is spent on that.

Less copies happen when a new request is received, 4.6% is spent on *ngx_ssl_recv()*, but half of that time goes to the underlying *BIO_read()*, which receives data from the kernel.

Tempesta TLS

Tempesta TLS is a fork of mbed TLS, significantly reworked to get maximum performance. The focus of Tempesta FW is mitigation of application layer DDoS attacks, so Tempesta TLS implements server-side TLS only to provide efficient TLS offloading for protected web services. X86-64 is the only supported CPU architecture now (actually, Tempesta FW requires SSE4.2 and AVX2 due to string algorithms optimizations, so only modern CPUs are supported).

Tempesta FW is built into the Linux TCP/IP stack [7], so Tempesta TLS is also developed as a Linux kernel module.

While there isn't much what we can do with a DDoS attack employing RSA handshakes (there the math has been optimized for years after all), the overhead for I/O, data copies and memory allocations is still important for ECDSA handshakes. The only opportunity to fight against RSA handshakes DDoS attack is to introduce a rate limit for such kind of handshakes. At some point, when TLS 1.3 get wider spreading, we can do the same for legacy TLS 1.2 handshakes.

Surprisingly, recent Linux kernels already have almost all crypto algorithms required for TLS and KTLS [19] uses the

Linux crypto framework to avoid data copies on *(*sendfile(2)) and *sendmsg(2)* system calls. While Tempesta TLS uses several KTLS definitions, the two implementations are very different.

KTLS

Normal web sites, like Facebook, have TLS sessions from their clients resumed, making TLS handshakes less crucial for the whole site performance. KTLS mostly focus on performance of large data transmission avoiding copies between user and kernel spaces.

KTLS works in process context on syscalls like *sendfile(2)* and *sendmsg(2)* (TCP protocol methods *sendmsg* and *sendpage* are overwritten by UTL (Upper Layer Protocols) interface in the TLS kernel module), so it uses sleeping functions (e.g. *sk_stream_wait_memory()*). In particular, *do_tcp_sendpages()* mentioned above sleeps if there is not enough space in TCP send buffer of a socket where a TLS record is going to be sent from.

Meantime, Tempesta FW works fully in softirq context, so it can not use KTLS directly. Only several definitions are reused. Some simple functions like writing a TLS header can be reused as well, while complex data structures, like describing a TLS context, must be significantly extended to support bidirectional communications, handshakes and multiple cipher suites.

The fork of mbed TLS

Mbed TLS is a time-proven relatively small, yet feature full, and very simple TLS library. The small code base and simplicity makes it a good starting point for custom TLS implementation.

We significantly reduced size of mbed TLS by:

- Removing portability wrappers to architectures except x86-64;
- Removing some features in RFC 7525 compliance, in particular compression, renegotiations, support of old and deprecated SSLv3, TLSv1.0, and TLSv1.1, weak RC4, and truncated HMAC;
- Also we removed insecure non-AEAD ciphers using CBC, CFB, and CTR modes of operation;
- Weak ciphers and hashes, like DES, MD2, MD4, MD5, and SHA1 are also gone;
- Common reverse proxy also doesn't require Pre-Shared Keys and elliptic curve J-PAKE.

Mbed TLS copies data to an internal buffer before encryption or decryption operation. Moreover, it can block on receiving data. Since Tempesta FW works in softirq context as a part of the TCP/IP stack, the I/O model proposed by mbed TLS doesn't suite our needs.

Mbed TLS sends each TLS record, including consequent handshake records, in separated TCP segments, which also hurts performance [1]. Tempesta TLS places all consequent handshake records in the same data chunk, so less memory allocations and egress TCP segments are required.

Targeting the code simplicity, mbed TLS implements most of crypto algorithms in very simple C code without possible SIMD optimizations existing in the kernel and OpenSSL.

These design constrains, as well as many unnecessary memory allocations, made us almost fully rewrite I/O mbed TLS routines and TLS handshakes code. Almost the whole crypto code was removed and replace by the calls to the linux crypto API.

Handshakes state machine

Ingress TLS handshake messages may arrive into the TCP/IP stack split into several *skb*s. A user space TLS library copies the socket data into a buffer for further processing.

Zero-copy for ingress data processing requires to properly handle a case when an ingress TLS record can be split among two TCP segments at any point, e.g. at the middle of 2-bytes data length. To handle the cases we use pushdown automaton, saving small chunks of the same message field among states, for ingress TLS handshake message processing. E.g. if a 2 byte field length is split among two skbs, then the first byte is saved in the automaton temporary memory until the second byte is read as well.

Handshake server states like *ServerHello*, *ServerCertificate*, *ServerKeyExchange*, *CertificateRequest*, and *ServerHelloDone* remain separate, but all of them write their data into the same page fragment and sent to the TCP/IP stack in the same *scatterlist*. The page fragment is allocated by the same memory allocator, *pg_skb_alloc()* [7], used for socket buffers allocation, so there is small memory overhead and the page fragment can be directly used as skb page fragments. The TCP/IP stack is responsible for splitting the data, so the minimal number of TCP segments are sent.

Tempesta TLS implements only server-side TLS, so we define ingress path as receiving client TLS records, either handshake or application data, and egress path as sending, probably forwarded from a backend server, data to a client.

Ingress path

Linux crypto framework calls for authenticated encryption with associated data (AEAD), e.g. *crypto_aead_decrypt()*, accept full encrypted message with final authentication tag, so we have to collect all ingress skbs in a list before sending them to crypto layer.

During collection of a message skbs we count number of contiguous memory chunks in the skbs, so when the whole message is collected we can allocate a crypto request with *scatterlist* in one shot.

Similar to the Linux IPSec implementation and unlike KTLS, Tempesta TLS performs decryption in-place, passing the same *scatterlist* as source and destination operands to crypto framework.

Egress path

In-place decryption is trivial since decrypted plaintext is smaller than authenticated ciphertext plus IV and a TLS header.

In-place encryption is not trivial. When Tempesta FW receives an HTTP response, always unencrypted since we do

not support TLS client mode so far, it has no idea whether the message is going to be forwarded to a client through encrypted connection or unencrypted HTTP connection. So having a list of skbs with an HTTP response data we have to allocate additional skb fragments at the beginning and the end of the message. It's worth mentioning that skb itself and its fragments are frequently allocated with some alignment overhead, an unused small memory areas. Meantime, the AEAD encryption and TLS headers require relatively small additional memory.

TLS encryption is performed on socket buffer (*struct sk_buff*) basis. Tempesta FW implements additional API for extending socket buffers with paged data [7]. Moreover, if we need a small adtitional data for a socket buffer, the API scans all current page fragments for available memory overhead - if it's found, then it's used for the additional page fragment. So we use the same technique as for zero-copy HTTP headers modifications [7]: lookup for a free memory areas among skb fragments and set fragment pointers to them to store TLS header and AEAD tag.

Servicing an HTTP client request from the web cache is trivial since we create necessary socket buffers from scratch instead of forwarding existing skbs from the backend server. In this case, we just allocate a socket buffer with additional space required for TLS header and tag.

Dynamic TLS records

In opposite to TCP operating with streams, TLS works with records, so TLS libraries and web servers using the libraries have to choose a size for TLS records. A receive side doesn't start to process a TLS record until it's fully read from a socket. Too small record size causes too much overhead. On the other side too large record size can lead to significant delays on receive side if current TCP congestion and/or the receiver's advertised window are smaller than a TLS record size [4].

Some of web servers (e.g. Nginx) statically define the record size in configuration file (16KB by default for Nginx [10]) while the others (e.g. H2O [5], ATS [13], or Nginx patched by CloudFlare [3]) define policies (also in a server configuration) to dynamically modify the buffer size. Such configurations can specify particular rules to change the buffer size or expected TCP parameters used for the buffer size calculation, but neither of the approaches are aware of precise values of current TCP congestion and receive windows.

KTLS also works with full 16KB TLS records pushing them into the TCP/IP stack by *do_tcp_sendpages()*.

Tempesta FW is built into the Linux TCP/IP stack, so it precisely knows current TCP congestion and send windows and even current MSS. We introduced a new socket hook *sk_write_xmit()* which is called from *tcp_write_xmit()* and peforms actual data encryption for data maximum allowed by the TCP windows minus required TLS overhead.

Unfortunately, we have no control how large socket buffers are queued in a TCP send queue of a socket, so to build TLS records of optimal size, we have to process several skbs to build a single TLS record if the queued skbs are too small. To do so we pass to our hook the *limit* for current maximum data transmission calculated in *tcp_write_xmit()*. Having the limit we can process following skbs in the send queue and add them to encryption *scatterlist* which will be used to build a TLS record. Once being encrypted skbs are flagged, so *tcp_write_xmit()* won't call the hook when it processes them.

memcpy(), memset(), and the kernel

Crypto libraries extensively use *memset()* to zeroize memory regions used for security-sensitive data, e.g. key material. While Tempesta TLS are designed in zero-copy fashion, *memcpy()* calls still happens , mostly to build a TLS record from internally stored configuration data and generated during handshake.

Since most of the kernel code can not use FPU and FPU context saving and restoring is expensive operation, the functions are implemented without SIMD processor extensions with obvious impact on performance.

Meantime, most of Tempesta FW code works in softirq and we store FPU context when we enter softirq and restore it when we leave softirq [7]. We made this for fast SIMD HTTP strings processing, so we made the same to implement vectorized versions of *memset()* and *memcpy()* [6].

Performance results for the functions are depicted on the figures 1 and 2 at the below.

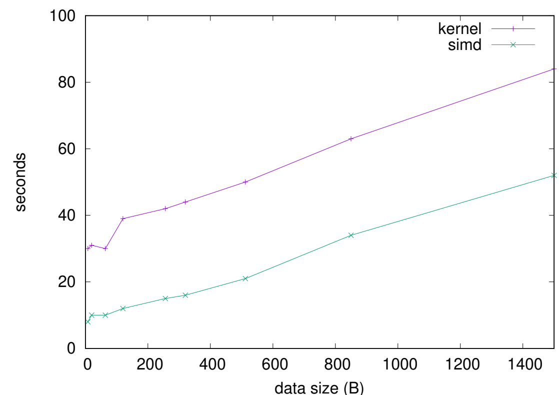

Figure 1: Comparison of kernel memcpy() and the optimized SIMD version

Discussion

Current version of the code is still in early prototype state. We'll publish results of performance measurements in several days.

Excluded from the library TLS_RSA_WITH_AES_128_CBC_SHA is required by RFC 5246 [12], but it's not recommended to use and also isn't supported by TLS 1.3 [16]. It seems most of the modern implementations use GCM mode and exclusion of the cipher suite won't hurt any real setups.

Mbed TLS source code isn't fully adopted and a lot of code cleanups are still to be done. There are still no TLS 1.3 and OCSP stapling implementations and these features are also in our to do list.

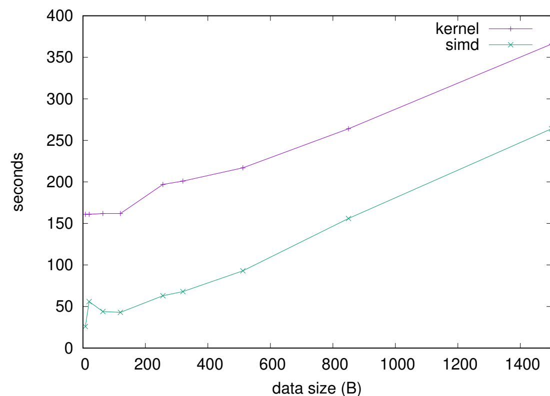

Figure 2: Comparison of kernel memset() and the optimized SIMD version

Big numbers arithmetic used in TLS handshakes remains the serious bottleneck. CPU can handle only modest number of handshakes per second, so at some rate it has sense to offload big number arithmetic from CPU. These days GPU is provided by many cloud and dedicated server providers, but loading data into a GPU memory is a heavy operation and has sense only for massively parallel computations. However, if a server experiences significant load, i.e. there are many concurrent requests for TLS handshake, we can offload the computations to GPU. We're exploring the possibility to use GPU for the big numbers arithmetic.

Availability

The Tempesta TLS is published under GPLv2 and is available as a part of Tempesta FW source code at https://github.com/tempesta-tech/tempesta.

References

[1] Case, H. mbedTLS vs BoringSSL on ARM. http://hdc.amongbytes.com/post/201804-comparing-mbedtls-to-boringssl/.

[2] Chen, A.; Sriraman, A.; Vaidya, T.; Zhang, Y.; Haeberlen, A.; Loo, B. T.; Phan, L. T. X.; Sherr, M.; Shields, C.; and Zhou, W. 2016. Dispersing asymmetric ddos attacks with splitstack. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 197–203.

[3] CloudFlare. Optimizing TLS over TCP to reduce latency. https://blog.cloudflare.com/optimizing-tls-over-tcp-to-reduce-latency/.

[4] Grigorik, I. Optimize TLS Record Size. In *High Performance Browser Networking*. https://hpbn.co/transport-layer-security-tls/#optimize-tls-record-size.

[5] H2O. Configuration derictives, latency optimization. https://h2o.example.net/configure/http2_directives.html#latency-optimization.

[6] Krizhanovsky, A. PoC and benchmark for kernel SIMD memcpy() and memset(). https://github.com/natsys/blog/tree/master/kstrings.

[7] Krizhanovsky, A. 2017. Kernel http/tcp/ip stack for http ddos mitigation. In *Netdev 2.1*. https://www.netdevconf.org/2.1/session.html?krizhanovsky.

[8] The mbed TLS project. https://tls.mbed.org/.

[9] Mozilla. Security/Server Side TLS. https://wiki.mozilla.org/Security/Server_Side_TLS.

[10] Nginx. Module ngx_http_ssl_module. https://nginx.ru/en/docs/http/ngx_http_ssl_module.html.

[11] Nginx. NGINX SSL Termination. https://docs.nginx.com/nginx/admin-guide/security-controls/terminating-ssl-http.

[12] The Transport Layer Security (TLS) Protocol Version 1.2. https://www.ietf.org/rfc/rfc5246.txt.

[13] Server, A. T. Dynamic TLS record size tuning. https://issues.apache.org/jira/browse/TS-2503.

[14] Tempesta Tech. HTTPS performance. https://github.com/tempesta-tech/tempesta/wiki/HTTPS-performance.

[15] The Hacker's Choice. SSL DoS utility. https://github.com/vanhauser-thc/THC-Archive/blob/master/Exploits/thc-ssl-dos-1.4.tar.gz.

[16] The Transport Layer Security (TLS) Protocol Version 1.3, Draft. https://tools.ietf.org/html/draft-ietf-tls-tls13-28.

[17] Valenta, L.; Sullivan, N.; Sanso, A.; and Heninger, N. 2018. In search of CurveSwap: Measuring elliptic curve implementations in the wild.

[18] Vincent Bernat. TLS computational DoS mitigation. https://vincent.bernat.im/en/blog/2011-ssl-dos-mitigation.

[19] Watson, D. 2016. Kernel tls (transport layer security) socket. In *Netdev 1.2*. http://www.netdevconf.org/1.2/session.html?dave-watson.

[20] Yandex. Load and performance benchmark tool. https://github.com/yandex/yandex-tank.