Kernel HTTPS/TCP/IP stack for HTTP DDoS mitigation

1st Alexander Krizhanovsky
Tempesta Technologies, Inc.
Seattle, USA
ak@tempesta-tech.com

Abstract

Application layer HTTP DDoS attacks are usually mitigated by HTTP accelerators or HTTP load balancers. However, Linux socket interface used by the software doesn't provide reasonable performance for extreme loads caused by DDoS attacks. Thus, HTTP accelerators are starting to bypass an OS and to use user space TCP/IP stacks. This paper discusses the drawbacks of the bypassing technique and explains why a general purpose HTTP accelerator isn't suitable for filtering volumetric attacks. Neither OS nor HTTP accelerators provide enough protection against application layer DDoS attacks.

Tempesta FW, an application delivery controller (ADC), extends the Linux TCP/IP stack with HTTPS and introduces a multi-layer firewall. The resulting HTTPS/TCP/IP stack protects Web applications against DDoS and Web attacks, so user-space programs can operate with preprocessed and cleaned HTTP traffic and be protected against DDoS attacks.

Keywords

HTTPS, Linux kernel, DDoS mitigation, Web security, performance

HTTPS isn't a Second-class Protocol

Nowadays Internet applications are all-pervasive. Since the amount of data exchanged over the Internet is always growing, the performance of HTTP processing becomes a crucial issue. Application layer DDoS attacks make the issue even more crucial. Although TCP and IP, implemented in an OS kernel, mostly carry HTTPS messages over the Internet, it seems that HTTPS is never considered for inclusion into an OS kernel as a continuation of TCP/IP stack.

While OS traditionally filter attacks on TCP and IP layers using firewalls, OS kernel doesn't provide protection against application layer attacks at all. Meantime, application layer DDoS attacks are one of the most crucial security issues in the modern Internet. In particular, some DDoS attacks employ complex techniques and can efficiently mimic a flashcrowd. This makes them hard to detect, so a victim system, even employing some DDoS mitigation logic, still must process tons of malicious HTTP requests. Thus, HTTP processing of an application layer DDoS mitigation solution must have outstanding performance. The growth of SaaS businesses, Internet of Things (IoT) and RESTful APIs stimulates the active development of various Web applications, which are vulnerable to application layer DDoS attacks and Web attacks.

Application layer HTTP DDoS attacks are usually mitigated by HTTP accelerators. However, Linux socket interface, used by the software, doesn't provide reasonable performance for efficient filtration of DDoS attacks. Also HTTP accelerators have no adequate interfaces to low layer OS information required for accurate classification of clients and HTTP requests.

The kernel bypass approaches [19, 10] address this sockets API performance issue by building fast Web servers on top of user-space TCP/IP stacks. However, kernel bypass Web servers can't be efficiently integrated with many of the useful tools which are provided by the mature OS TCP/IP stack (e.g. IPTables, tc, LVS, tcpdump), which limits their real life usage.

Linux provides efficient zero-copy IO interfaces like *sendfile(2)* and *vmsplice(2)*. However, HTTPS makes data copying between user and kernel address spaces a serious performance hurdle: the standard OS API provides no way to send a file over a TLS connection without copying. There are efforts [4, 21] to move TLS partially to kernel space. Also, establishing a new TLS connection requires 4 context switches for the initial handshake which introduces additional overhead. That creates an attack vector for HTTPS DDoS.

All in all, to be able to filter volumetric application layer attacks, we have to either move HTTPS processing to the kernel or move TCP/IP stack with Netfilter and traffic control to user space. However, the full TCP/IP stack is a very huge and complex code, so it's not wise to implement and run it twice in user and kernel spaces. The paper proposes to move only basic and the most crucial parts of HTTPS processing to the kernel.

Tempesta FW extends the Linux TCP/IP stack by HTTPS and implements application layer firewall. As the a result, it provides Web applications protection against various application layer attacks, including DDoS, and reaches the same performance as HTTP servers built on top of user space TCP/IP stacks.

HTTP DDoS Resistance

While the OS kernel doesn't deal with application (HTTPS) layer, it can not provide solid protection against application layer DDoS attacks. These attacks are usually mitigated

by user space HTTP accelerators. This section discusses why traditional HTTP accelerators aren't suitable to mitigate HTTP DDoS attacks and how to improve them to be HTTP DDoS resistant.

There are many HTTP accelerators developed to handle a lot of concurrent connections and tens thousands of requests per second. However, it seems as though none of them were designed to process millions of requests per second, of which most of them are malicious and must be filtered out. The OS kernel doesn't deal with HTTP at all, so it can't filter out DDoS attacks as well. I.e. neither HTTP accelerators nor OS provide enough protection against application layer DDoS attacks.

Optimizing HTTP Parser

We have studied several HTTP servers and proxies (Nginx, Apache Traffic Server, Cherokee, node.js, and Varnish) and learned that all of them use *switch* and/or *if-else* driven finite state machines (FSMs). If logging is switched off and all content is in the cache, then HTTP parser becomes the hottest spot. Simplified output from *perf* for Nginx under simple HTTP flood is shown below (Nginx's calls begin with *'ngx_'* prefix):

```
    %          symbol name
  1.5719   ngx_http_parse_header_line
  1.0303   ngx_vslprintf
  0.6401   memcpy
  0.5807   recv
  0.5156   ngx_linux_sendfile_chain
  0.4990   ngx_http_limit_req_handler
```

Listing 1: Nginx profile for HTTP flood

Listing 2 shows a simplified switch driven HTTP parser:

```
        // state = 2, *str_ptr = 'b'.
1: 6:   while (++str_ptr) {
2: 7:       switch (state) {
            case 1:
                // ...
            case 2:
                switch (*str_ptr) {
                case 'a':
                    // ...
                    state = 1;
                    break;
                case 'b':
3:                  // ...
                    state = 3;
                    break;
                case 'c':
                    // ....
                }
4:              break;
            case 3:
                // ...
            }
5:          // ...
        }
```

Listing 2: Dummy swtich driven HTTP parser

If the FSM starts at state *2* with input character *'b'*, then the FSM performs the following jumps in code:

1. start and check current character;
2. check the state variable and jump to code for state *2* (label *3* on the listing);
3. assign next state to variable *state* and *break* to the end of *switch* statement, i.e. do the second jump;
4. this is the end of code for state *2*, so we do the 3rd jump to the end of the outer *switch* statement;
5. now we're at the end of the *while* statement and we do the 4th jump to the beginning of the loop;
6. move and check the next input character;
7. jump to state *3*.

Thus, the FSM jumps a lot. We dont actually need the variable *state* and instead of assigning value *3* to the variable at step *3*, we could just read the next character, check it's value and unconditionally jump to state *3*. Basically, Ragel [17] generates FSMs in exactly this way, without unnecessary jumps and without the *state* variable. An HTTP parser constructed using the more accurately coded FSM is up to 3 times faster [12] than a dummy FSM on long HTTP header lines.

HTTP-aware String Processing

While a developer can be optimistic about the most common cases and write an HTTP accelerator in assumption that most of HTTP fields have normal lengths, DDoS attacks are always about corner cases. A careful attacker can craft a worklod specially for a victim Web application. If your Web server isn't good at processing tons of packets, then the attacker launches a flood attack against the server. If the server can't handle many connections, then the attacker will establish a lot of connections with the server. If your server can't efficiently process long strings (e.g. values of HTTP headers), then the attacker can send you HTTP requests with enormously long HTTP headers. Generally, DDoS attacks are about corner cases: if an HTTP accelerator is very fast in normal circumstances, it still can be quite inefficient under specially crafted DDoS attack.

The FSM's performance, shown on the listing 2, quickly degrades as an input string grows. Some HTTP servers use LIBC functions like *memchr()*, which employ SIMD instructions, to process HTTP strings. LIBC string functions are developed to process common zero-terminated strings, and they are not designed to validate input against allowed alphabet (i.e. to verify that URI of an HTTP request contains only RFC allowed characters).

Besides HTTP parsing, HTTP accelerator does a lot of operations with parsed HTTP messages. In particular, *strncasecmp()* and family calls are very frequent in a common HTTP accelerator.

Specially designed and implemented using AVX2 instruction set strings processing algorithms, with strong input validation against RFC-allowed character sets, can deliver up to 13x better performance. The algorithms' descriptions and benchmarks are out of the scope of the paper. The full description can be found in our blog article [13].

Copyings and Syscalls

Listing 1 also shows copyings (*memcpy()*) and I/O (*recv()* and *ngx_linux_sendfile_chain()*) as hot spots. Data copyings between kernel and user spaces, context switches introduced by system calls, and slowness of the system calls are the subjects for user space TCP/IP implementations.

Performance measurements of Tempesta FW show that if HTTP is coupled with the Linux TCP/IP stack, then performance of the resulting HTTP accelerator will be comparable with an HTTP accelerator implemented on top of user space TCP/IP stack.

Simple Web servers built on top of user space TCP/IP stack can show outstanding performance. Sandstorm [10] proposes an extreme case of a very simple web server. They built a light-weight network stack in user-space directly operating with the network adapter and bypassing OS. The server pregenerates packets, avoiding allocation and the initialization of socket buffers. The packets already have established link-layer, IP and TCP headers with calculated checksums. However, such a tightly optimized approach has several limitations for usage in real world applications:

- path MTUs, and correspondingly TCP MSS, can differ on network interfaces and also can be changed any time, so different packet sizes must be used;
- TCP connections can use different TCP and IP options as well as using IPv4 or IPv6;
- some HTTP headers, like cache and connection control, must be generated on-the-fly depending on current connection properties;
- the technique can't be used for TLS connections which are almost mandatory for modern Web resources.

Sandstorm's network stack employs NIC's TX rign as the sole output queue ignoring traffic control (TC) queueing disciplines. Meantime queueing discipline provides fair packet scheduling among many network flows as well as controlling network delays, mitigating bufferbloat at transit network points.

Meanwhile, coupling HTTPS with the Linux TCP/IP stack also allows for the following optimizations:

- The removal of operations with file (*struct file*) and BSD socket (*struct socket*) descriptors;
- The elimination of accept and receive socket queues because all processing of ingress traffic can be done in socket callbacks;
- The extension of *sk_buff* API for paged fragments to implement zero-copy HTTP messages processing and transformation (e.g. removing an HTTP header from a request or replacing it with a longer or shorter header).
- The use of controlled preemption to allow for the implementation of more efficient lock-free algorithms;

Coupling a Firewall with HTTP

Tight binding of HTTP to TCP/IP is required for efficient execution of multi-layer firewall rules. Let's consider Wordpress "Pingback" DDoS attack [24] as an example. This is a reflective DDoS attack which uses the Wordpress Pingback feature. The result of this attack is that a third party Wordpress installation sends requests to a victim's site. Since the attack uses a regular Wordpress installation, the HTTP requests contain the User-Agent header with the "WordPress" string.

To block the attack one can use a Web server's filtering abilities, e.g. Nginx's User-Agent matching [14]. However, the Web server executes several system calls and does several data copyings to block *each* request. That doesn't work quickly. A log monitor [6] like Fail2Ban[5] can be used to generate firewall rules on-the-fly, but that requires parsing of access log which is very expensive as well.

Another way to mitigate the attack is using IPTables [8] to block all the packets containing the "WordPress" string. However, IPTables blindly inspects *each* packet using plain string matching. For example, if a request contains a long URI, then IPTables tries to find the string in the URI as well. This makes the processing inefficient and error-prone, since the URI can contain the string as well. Thus several non-trivial optimizations must be used to make IPTables rules match User-Agent value correctly. Moreover, IPTables can find strings in single packet scope only, so if User-Agent value crosses IP packet boundary, then IPTables can't match it.

The HTTPS/TCP/IP Stack

This section describes the architecture of Tempesta FW, a Linux application delivery controller (ADC), in the context of HTTPS processing and multi-layer firewall. The core of the project is a fast HTTP state machine built into the Linux TCP/IP stack, so that there is a uniform stack of protocols: IP, TCP, TLS, and HTTP.

HTTP and TLS messages are processed in softirq context, right after TCP processing. Thus, TCP receive and accept queues are not used. That removes unnecessary queueing operations and makes HTTP processing faster because all packet data is hot in all CPU caches. Ingress TCP segments don't wait in the receive queue and don't waste system memory. Instead, they are immediately passed to HTTP parser and then to a classifier module (Frang), which sorts them as either innocent or malicious. Thus, like most firewalls Tempesta FW efficiently uses drop early strategy.

Figure 1 shows the operation of Tempesta FW on an ingress HTTP request:

1. HTTP request is received by the network adapter and verified against filtering tables of Frang limiting module.
2. Client's account is found for the TCP socket. TCP window, sent with appropriate ACK, is defined by QoS for this client according to the client's accounting.
3. The HTTP request is immediately parsed in softirq handler while data is hot in CPU caches.
4. The request is analyzed by Frang and can be blocked at IP layer using filter rule propagation so that subsequent queries from the client are blocked at step 1.
5. The request is serviced from Web cache or forwarded to an upstream server according to established load balancing policy. The web cache is built on top of TempestaDB, an

in-memory database built on top of cache conscious lock-free hash trie.

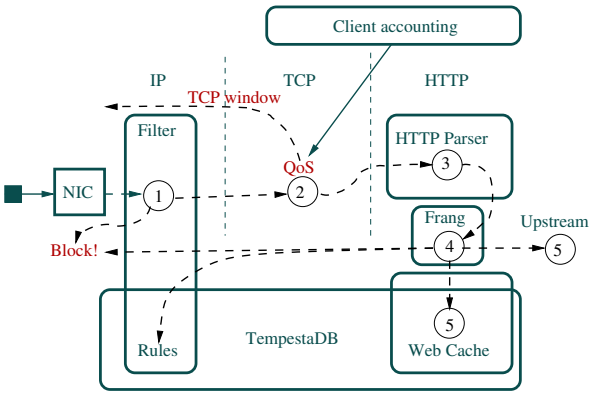

Figure 1: The HTTP/TCP/IP stack with filtering and QoS

Frang module implements strict validation of HTTP requests according to a number of defined hard limits. The limits include connection and request rates, size of different HTTP fields, number of chunks in HTTP request to prevent Slow HTTP [20], and many others. Frang implements a hard blocking strategy suitable for stopping a Web application attack: a malicious request is immediately blocked along with the client's IP.

HTTP sticky cookies module is integrated into the HTTPS/TCP/IP stack. Apart from load balancing of persistent HTTP sessions among a number of upstream servers, the module is useful for cookies challenge. If an attacker uses an HTTP client with restricted functionality to launch a DDoS attack, then the bots can be unable to process Set-Cookie header and will be blocked by Tempesta FW. We're also working on a JavaScript challenge module, which requires the usage of an even more advanced HTTP client for DDoS bots.

SSL/TLS handshake is a very expensive operation and this creates a DDoS attack vector [22]. 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 mbed TLS [11] to the kernel. Surprisingly, recent Linux kernels already have almost all TLS required crypto algorithms, so our next step is to have mbed TLS crypto algorithms replaced by appropriate Linux kernel crypto framework routines.

Inter-CPU Sockets Transport

Tempesta FW is a reverse proxy, so 2 CPUs can send two packets in opposite directions through the same two, client and server, sockets. Since TCP control blocks of in the sockets must be locked, it could lead to a deadlock. Moreover, it's not desirable to access to the same TCBs from different CPUs.

Thus, we use a lock-free ring buffer as an inter-CPU transport, shown by Figure 2. Each socket with its TCB is hosted at a particular CPU. Each CPU has its own lock-free queue based on a ring buffer. If a CPU needs to send some packet through a socket owned by another CPU, then it sends a job to the ring buffer of the CPU owning the target socket. If the socket is owned by the same CPU, then the ring buffer isn't used. Thus, there is no lock contention and the crucial data stays per-CPU. The architecture provides faster socket reading which is important for managing huge ingress traffic caused by DDoS. Also, thanks to absence of traditional input queue and socket locks, we get lower latency for HTTP processing.

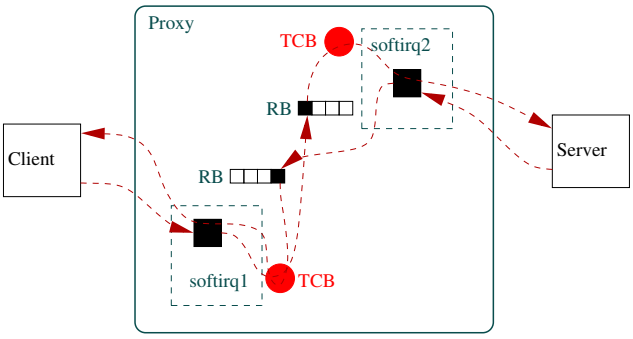

Figure 2: Inter-CPU sockets transport

QoS for Accurate DDoS Mitigation

While QoS is still under development, it must be mentioned because it impacts the overall system design. Since sometimes DDoS requests are indistinguishable from normal client requests, QoS module must smoothly reduce bandwidth for a particular client (malicious or innocent) depending on a classification decision. The classification is performed based on the client's accounting data. Examples are the amount of CPU and system memory used to service the client, and the ratio of the number of HTTP requests from the client to the number of responses from an upstream server. QoS can also be triggered to reduce bandwidth of the less rated clients if the system undergoes some *stress* conditions, such as massive packet drops, insufficient CPU or memory resources, overrun queues and so on. A small reduction of the bandwidth of the most active or suspicious clients will protect the system from overloading by either a flashcrowd or a DDoS attack.

The classification logic must intensively gather system statistics to classify the clients. The system resource accounting is extremely important for protecting against asymmetric DDoS attacks [2, 16, 22], which waste victim's resources at a much higher rate than attacker's, e.g. SSL/TLS [22] or compression DDoS [16] attacks. The statistics can be collected in user space using *getrusage(2)* system call, but the system call must be called too frequently, at the cost of numerous context switches. Thus, efficient filtration at application layer requires access to lower kernel layer, and user space TCP/IP doesn't solve the issue.

Keep the Kernel Small

Basically, the idea to move HTTPS to OS kernel isn't new. Several mainstream operating systems implement HTTP and

TLS logic in kernel. AIX uses the kernel Web-cache accelerator AFPA (FRCA) [9, 23, 7]. Solaris provides SSL Kernel Proxy [15] for a while. There are also experiments [4, 21] with kernel space TLS for zero-copy bulk data transfers over encrypted connections.

Although AFPA [9] has shown very high performance, the authors regarded the approach as awkward and predicted that it would lead to a paradigm where the benefits of address spaces were lost. Thus, Tempesta FW's architecture proposes to implement only the following in the kernel space to keep the kernel as small as possible:

- *generic functionality*, like basic HTTP. It must be generic enough to support different Web and database applications.
- *efficient and flexible interfaces* to the generic functionality which allows for user-space applications use of the kernel API, without architecture or performance restrictions.
- *performance critical and quick synchronous operations*, as kernel is not a good place to run heavy logic. Things like client classification for DDoS mitigation using machine learning can be done asynchronously and should not be done in kernel space.
- *mission critical logic*. An HTTP accelerator should at least return the 503 Service Unavailable error instead of becoming unresponsive due to overload.

Blocking malicious HTTP requests, load balancing, and servicing HTTP requests from Web cache are simple and fast processes. At the same time, other required logic can be too slow or too big for implementation in kernel space. A good example of heavy and noncritical logic is HTTP compression. In fact, if a client sends Accept-Encoding which requires some compression, a servers still can send plain text representation. Such logic must be implemented in user space to minimise kernel space code. Thus, we should be able to pass some HTTP requests to user space for complex processing and get appropriate responses from user space. To address the issue, we're developing a fast zero-copy transport of HTTP messages between kernel and user spaces.

The scenario for processing an ingress HTTP request and sending a generated HTTP response is shown in Figure 3:

1. Softirq handler receives packets that hold an HTTP message. The Linux TCP/IP stack is patched so that the packet's payload is always placed in memory pages, which can be *mmap()*'ed.
2. The message is parsed and all required data, including the parsing meta- information and the packet's data, are placed in several memory pages. HTTP messages are processed in a zero-copy fashion, i.e. HTTP fields are not copied. Instead, appropriate pointers are stored in the parsing meta-information which point into the received packet data, like the start of HTTP header field name and value.
3. When memory pages of the HTTP message are mapped to the advanced classification process' address space, the softirq handler wakes up the process.
4. Now the process can run heavy logic on the *mmap()*'ed message (let it be a request to simplify the example).
5. The process can generate a response for the request (e.g. with HTTP error code for an improper request). The same memory mapped region is used to pass the HTTP response to the kernel.
6. Finally, softirq handler can send the response to the client.

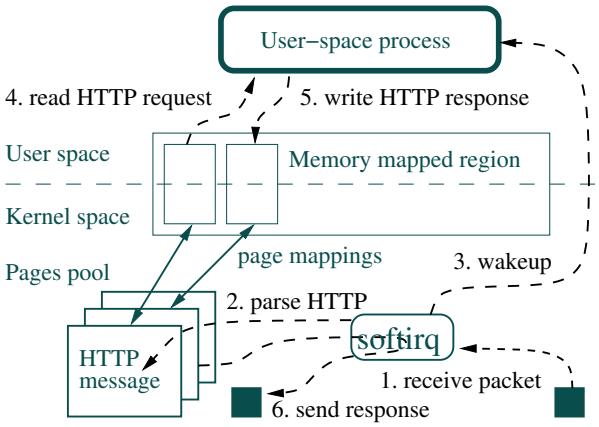

Figure 3: Transport of HTTP messages between the kernel and user spaces

Performance

We run benchmark tests to measure servicing of pure Web requests from Web cache. Tempesta FW reached 1.8M requests per seconds on a cheap 4-core machine. Our best tuned Nginx on the same hardware showed results that were 3 times worse. Seastar [19], when built on top of a user-space TCP/IP stack (DPDK [3] in particular), only shows [18] just 1.3M RPS on 4 cores. A full description of Tempesta FW's benchmark (the used hardware, the configuration, and the workload) is available at the project's Wiki [1].

It's a challenge to build a testbed to emulate a DDoS attack and measure how quickly the attack is blocked. However, we expect that Tempesta FW's filtering of HTTP DDoS requests should outperform modern Web accelerators and low-layer firewalls, even more due to the native filtering abilities.

Thus, bypassing Linux TCP/IP stack isn't the only way to get a fast HTTP accelerator or Web server.

Discussion

This paper proposes to move a Web accelerator, which is traditionally placed in user space, to OS kernel. Nevertheless, it's important not to end up with a "macrokernel" (as opposed to microkernel) operating system, which moves everything to kernel space. Several of our considerations regarding the functionality of what should be in kernel space were defined above. We must keep kernel code as small as possible. Now Tempesta FW only has about 30,000 lines of kernel C code (for example that can be compared to BtrFS code which has about 120,000 lines of C code in Linux 4.8). Evolved interfaces between the user and kernel spaces, such as passing HTTP messages instead of raw TCP data, will help to keep the kernel code as small as possible.

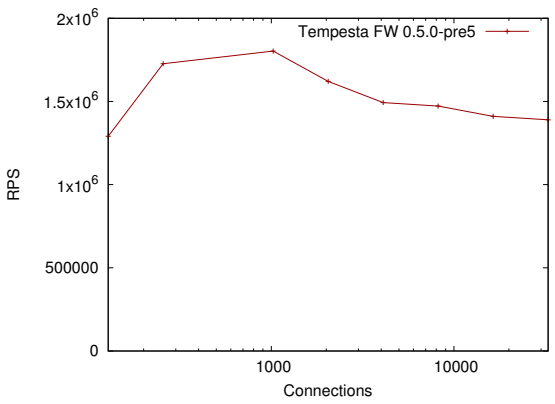

Figure 4: HTTP Requests per second serviced from the Tempesta FW's cache

While TLS is very complex code, it doesn't require complex locking, advanced memory management, and so on. It only took 1 human month for us to port mbed TLS, with all necessary HTTPS interfaces to the kernel. Thus, it's easier to move TLS to the kernel than it is to move TCP/IP stack to user space. Reusing Linux kernel crypt API for TLS will reduce the effort, and the project code, even more.

A previous study [9] considered the moving of an HTTP server to kernel space as a bad practice. Currently, Tempesta FW's core should not be regarded as a drop-in replacement for traditional Web servers or even HTTP accelerators. Instead, it's supposed to stay in front of a more powerful, but slower, HTTP server, accelerating normal traffic and dropping early malicious traffic. Tempesta FW works perfectly with traditional Web servers on the same host. Further features development in user space using the kernel-user space zero-copy transport will develop Tempesta FW nto a real drop-in replacement for current HTTP servers and accelerators.

The alternative way to build such ADC is to use user space TCP/IP stack and build efficient HTTP accelerator and a firewall on top of it. However, TCP/IP stack is basically huge and complex code, so it's not wise to implement and run it twice in user and kernel spaces. Next, kernel TCP/IP stack is already well integrated with many powerful tools like IPTables, IPVS, tc, tcpdump and many others. The tools are unavailable for a user space TCP/IP stack, or require complex interfaces.

Reimplementing all the necessary features in user space to get a featureful ADC brings additional code complexity and makes the code slower. Small specialized implementations [10] outperform the Linux TCP/IP stack, but introducing more features will typically hit the same issues as in the Linux TCP/IP stack.

Availability

The Tempesta FW's source code is published under GNU GPLv2 and is available at `https://github.com/tempesta-tech/tempesta`.

References

[1] HTTP cache performance. In *Tempesta FW Wiki*. `https://github.com/tempesta-tech/tempesta/wiki/HTTP-cache-performance`.

[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] Data Plain Development Kit. `http://dpdk.org/`.

[4] Edge, J. 2015. TLS in the kernel. In *LWN*. `https://lwn.net/Articles/666509/`.

[5] Fail2ban. `http://www.fail2ban.org`.

[6] Glass, B. 2002. Log Monitors in BSD UNIX. In *BSDCon*, 131–142.

[7] IBM. Serving static content faster with Fast Response Cache Accelerator. In *IBM Knowledge Center*. `https://www.ibm.com/support/knowledgecenter/SSEQTJ_8.5.5/com.ibm.websphere.ihs.doc/ihs/tihs_cacheenable.html`.

[8] 2015. IPTABLES block User-Agent. In *ServerFault discussion*. `http://serverfault.com/questions/690870/iptables-block-user-agent`.

[9] Joubert, P.; King, R.; Neves, R.; Russinovich, M.; and Tracey, J. 2001. High-performance memory-based web servers: Kernel and user-space performance. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, 175–187.

[10] Marinos, I.; Watson, R. N. M.; and Handley, M. 2014. Network stack specialization for performance. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, 175–186.

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

[12] 2014. Fast Finite State Machine for HTTP Parsing. `http://natsys-lab.blogspot.ru/2014/11/the-fast-finite-state-machine-for-http.html`.

[13] 2016. HTTP Strings Processing Using C, SSE4.2 and AVX2. `http://natsys-lab.blogspot.ru/2016/10/http-strings-processing-using-c-sse42.html`.

[14] 2015. Mitigating DDoS Attacks with NGINX and NGINX Plus. In *Nginx Blog*. `https://www.nginx.com/blog/mitigating-ddos-attacks-with-nginx-and-nginx-plus/`.

[15] Oracle. SSL Kernel Proxy Encrypts Web Server Communications. In *Securing the Network in Oracle Solaris 11.1*. `https://docs.oracle.com/cd/E26502_01/html/E28990/webk-2.html`.

[16] Pellegrino, G.; Balzarotti, D.; Winter, S.; and Suri, N. 2015. In the compression hornet's nest: A security study

of data compression in network services. In *Proceedings of the 24th USENIX Conference on Security Symposium*, 801–816.

[17] Ragel State Machine Compiler. http://www.colm.net/open-source/ragel/.

[18] Seastar HTTP performance. http://www.seastar-project.org/http-performance/.

[19] The Seastar project. http://www.seastar-project.org/.

[20] Slowloris, the low bandwidth, yet greedy and poisonous HTTP client. http://ha.ckers.org/slowloris/.

[21] Stewart, R.; Gurney, J.; and Long, S. 2015. Optimizing tls for highbandwidth applications in freebsd. In *Proceedings of AsiaBSDCon Conference*. https://people.freebsd.org/~rrs/asiabsd_2015_tls.pdf.

[22] The thc-ssl-dos tool. The Hackers Choice. https://www.thc.org/thc-ssl-dos.

[23] Voigt, T.; Tewari, R.; Freimuth, D.; and Mehra, A. 2001. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, 189–202.

[24] More Than 162,000 WordPress Sites Used for Distributed Denial of Service Attack. In *Sucuri Blog*. https://blog.sucuri.net/2014/03/more-than-162000-wordpress-sites-used-for-distributed-denial-of-service-attack.html.