Tempesta FW
Linux Application Delivery Controller

Alexander Krizhanovsky
Tempesta Technologies, Inc.

ak@tempesta-tech.com
Who am I?

- CEO & CTO at **Tempesta Technologies** *(Seattle, WA)*
- Developing **Tempesta FW** – open source Linux Application Delivery Controller (ADC)
- **Custom software development** in:
  - *high performance network traffic processing*
    e.g. **WAF** mentioned in **Gartner magic quadrant**
  - **Databases**
    e.g. **MariaDB SQL System Versioning** is coming soon
    *(https://github.com/tempesta-tech/mariadb_10.2)*
Challenges

- Usual Web accelerators aren’t suitable for HTTP filtering
- Kernel HTTP accelerators are better, but they’re dead
- \(\Rightarrow\) need a hybrid of HTTP accelerator and a firewall
  - Very fast HTTP parser to process HTTP floods
  - Very fast Web cache to mitigate DDoS which we can’t filter out
  - Network I/O optimized for massive ingress traffic
  - Advanced filtering abilities at all network layers
L7 DDoS mitigation Web accelerator?

Web Accelerator

Web cache
DDoS protection
Load balancer

Upstream servers
Application Delivery Controller (ADC)

- Web cache
- DDoS protection
- Load balancer
- Web application security
- Application performance monitoring
- SSL/TLS offloading
- Data compression
- Connections QoS

Upstream servers
Use cases

- CMSes, CDNs, virtual hostings, heavy loaded Web sites, OEMs in Web security etc.

- Usual ADC cases:
  - **When you need performance**
  - Web content acceleration
  - Web application **protection**
  - HTTP load balancing
## Application layer DDoS

<table>
<thead>
<tr>
<th></th>
<th>Service from Cache</th>
<th>Rate limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx</td>
<td>22us</td>
<td>23us</td>
</tr>
</tbody>
</table>

- *(Additional logic in limiting module)*
- **Fail2Ban**: write to the log, parse the log, write to the log, parse the log...
## Application layer DDoS

<table>
<thead>
<tr>
<th>Nginx</th>
<th>Service from Cache</th>
<th>Rate limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22us</td>
<td>23us</td>
</tr>
</tbody>
</table>

- *(Additional logic in limiting module)*
- **Fail2Ban**: write to the *log*, parse the *log*, write to the *log*, parse the *log*… - really in 21st century?!
- **tight integration** of Web accelerator and a firewall is needed
Web-accelerator capabilities

- Nginx, Varnish, Apache Traffic Server, Squid, Apache HTTPD etc.
  - cache static Web-content
  - load balancing
  - rewrite URLs, ACL, Geo, filtering etc.
Web-accelerator capabilities

- Nginx, Varnish, Apache Traffic Server, Squid, Apache HTTPD etc.
  - cache static Web-content
  - load balancing
  - rewrite URLs, ACL, Geo, filtering? etc.
Web-accelerator capabilities

- Nginx, Varnish, Apache Traffic Server, Squid, Apache HTTPD etc.
  - cache static Web-content
  - load balancing
  - rewrite URLs, ACL, Geo, filtering? etc.
  - C10K
Web-accelerator capabilities

- Nginx, Varnish, Apache Traffic Server, Squid, Apache HTTPD etc.
  - cache static Web-content
  - load balancing
  - rewrite URLs, ACL, Geo, *filtering?* etc.
  - **C10K** – *is it a problem for bot-net? SSL?* CORNER
  - what about tons of '*GET / HTTP/1.0\n\n'*? CASES!
Web-accelerator capabilities

- Nginx, Varnish, Apache Traffic Server, Squid, Apache HTTPD etc.
  - cache static Web-content
  - load balancing
  - rewrite URLs, ACL, Geo, filtering? etc.
  - **C10K – is it a problem for bot-net? SSL?**
  - what about tons of 'GET / HTTP/1.0\n\n'? **CORNER CASES!**
- **Kernel-mode Web-accelerators**: TUX, kHTTPd
  - basically the same sockets and threads
  - zero-copy → *sendfile*, lazy TLB
Web-accelerator capabilities

- Nginx, Varnish, Apache Traffic Server, Squid, Apache HTTPD etc.
  - cache static Web-content
  - load balancing
  - rewrite URLs, ACL, Geo, filtering? etc.
  - C10K – is it a problem for bot-net? SSL? CORNER
  - what about tons of 'GET / HTTP/1.0\n\n'? CASES!
- Kernel-mode Web-accelerators: TUX, kHTTPd
  - basically the same sockets and threads
  - zero-copy → sendfile(), lazy TLB => not needed
Web-accelerator capabilities

- Nginx, Varnish, Apache Traffic Server, Squid, Apache HTTPD etc.
  - cache static Web-content
  - load balancing
  - rewrite URLs, ACL, Geo, **filtering?** etc.
  - C10K – *is it a problem for bot-net? SSL?* **CORNER**
  - what about tons of `'GET / HTTP/1.0\n\n'`? **CASES!**
- Kernel-mode Web-accelerators: TUX, kHTTPd **NEED AGAIN**
  - basically the same sockets and threads
  - zero-copy → `sendfile()`, lazy TLB => not needed **DDOS**
Web-accelerators are slow: SSL/TLS copying

- User-kernel space copying
  - Copy network data to user space
  - Encrypt/decrypt it
  - Copy the data to kernel for transmission

- Kernel-mode TLS
  - Facebook, RedHat: https://lwn.net/Articles/666509/
  - TLS handshake is still an issue
Web-accelerators are slow: profile

<table>
<thead>
<tr>
<th>%</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5719</td>
<td>ngx_http_parse_header_line</td>
</tr>
<tr>
<td>1.0303</td>
<td>ngx_vslprintf</td>
</tr>
<tr>
<td>0.6401</td>
<td>memcpy</td>
</tr>
<tr>
<td>0.5807</td>
<td>recv</td>
</tr>
<tr>
<td>0.5156</td>
<td>ngx_linux_sendfile_chain</td>
</tr>
<tr>
<td>0.4990</td>
<td>ngx_http_limit_req_handler</td>
</tr>
</tbody>
</table>

=> flat profile
Web-accelerators are slow: syscalls

```
epoll_wait(., {{EPOLLIN, .}}, .)
recvfrom(3, "GET / HTTP/1.1\r\nHost:....", .)
write(1, "...limiting requests, excess...", .)
writev(3, "HTTP/1.1 503 Service...", .)
sendfile(3, ..., 383)
recvfrom(3, ..., -1 EAGAIN)
epoll_wait(., {{EPOLLIN, .}}, .)
recvfrom(3, ", 1024, 0, NULL, NULL) = 0
close(3)
```
Web-accelerators are slow: filesystem database

- Plain files database
  - Nginx, Squid, Apache HTTPD

```
/cache/0/1d/4af4c50ff6457b8cabfecd32d0b2f1d0
/cache/5/2e/9f351cdefc8027852656aad53f9372e5
/cache/f/22/554a5c654f189c1630e49834c25ae229
```

- Apache Traffic Server (ATS) uses database like Web-cache
  - Vary header requires **secondary key** (say “hello” to databases)
Web-accelerators are slow: HTTP parser

**Start:** $state = 1, \ *str\_ptr = 'b'$

```c
while (++str_ptr) {
    switch (state) { <= check state
        case 1:
            switch (*str_ptr) {
                case 'a':
                    ...
                    state = 1
                case 'b':
                    ...
                    state = 2
            }
        case 2:
            ...
    }
} ...
```

Tempesta Technologies
Web-accelerators are slow: HTTP parser

Start: state = 1, *str_ptr = 'b'

while (++str_ptr) {
    switch (state) {
    case 1:
        switch (*str_ptr) {
            case 'a':
                ...  // code
                state = 1
            case 'b':
                ...  // code
                state = 2 <= set state
        }
    case 2:
        ...  // code
    }
    ...  // code
}
Web-accelerators are slow: HTTP parser

Start: state = 1, *str_ptr = 'b'

while (++str_ptr) {
    switch (state) {
        case 1:
            switch (*str_ptr) {
                case 'a':
                    ...
                    state = 1
                case 'b':
                    ...
                    state = 2
                }
        case 2:
            ...
    }
    ... <= jump to while
}
Web-accelerators are slow: HTTP parser

**Start:** state = 1, *str_ptr = 'b'

```c
while (++str_ptr) {
    switch (state) {
        <= check state
    case 1:
        switch (*str_ptr) {
            case 'a':
                ...
                state = 1
            case 'b':
                ...
                state = 2
        }
    case 2:
        ...
    }
    ...
}
```
Web-accelerators are slow: HTTP parser

Start: state = 1, *str_ptr = 'b'

while (++str_ptr) {
    switch (state) {
        case 1:
            switch (*str_ptr) {
                case 'a':
                    ...
                    state = 1
                case 'b':
                    ...
                    state = 2
            }
        case 2:
            ...
            <= do something
    }
    ...
}
Web-accelerators are slow: HTTP parser

while (++str_ptr) {
    switch (state) {
    case 1:
        switch (*str_ptr) {
        case 'a':
            ...
            state = 1
        case 'b':
            ...
            state = 2
        }
    case 2:
        ...
    }
    ...
    3
}

while (1):
    STATE_1:
        switch (*str_ptr) {
        case 'a':
            ...
        case 'b':
            ...
            ++str_ptr
            goto STATE_1
    }
        ++str_ptr
    STATE_2:
    ...
}
Web-accelerators are slow: strings

- We have AVX2, but GLIBC doesn’t still use it
- HTTP strings are special:
  - No ‘\0’-terminating (if you’re zero-copy)
  - Special delimiters (‘:’ or CRLF)
  - `strcasestr()`: no need case conversion for one string
  - `strspn()`: limited number of accepted alphabets
- `switch()`-driven FSM is even worse
Web-accelerators are slow: async I/O
Web-accelerators are slow: async I/O
Web-accelerators are slow: async I/O
Web-accelerators are slow: async I/O

DCA

(Intel’s CPUs
with Intel’s NICs)
Tempesta FW

- ADC architecture: a **hybrid of HTTP accelerator and FireWall**
- **Multi-layer FireWall**: layer 3 (IP) – layer 7 (HTTP) filter
- **Directly embedded** into Linux TCP/IP stack (as *traditional firewalls*)
- Built-in filters for **L7 DDoS** and **Web application attacks**
- Very fast **HTTP parser** and strings processing using AVX2
- **Kernel TLS** (fork from mbedTLS) – no copying!
- **NUMA-aware x86-64 cache conscious** Web-cache on **huge pages**
- **Advanced load balancing**
- This is **Open Source** (GPLv2)
Performance

Intel Xeon E3-1240v6 (4 cores); 8B response, keep-alive

https://github.com/tempesta-tech/tempesta/wiki/Tempesta-FW-benchmark
Performance analysis

- ~x3 faster than Nginx for normal Web cache operations
- Must be much faster to block HTTP DDoS
  (DDoS emulation is an issue)
- Similar to DPDK/user-space TCP/IP stacks
  - e.g. Seastar seems shows just 1.3MRPS on 4 cores
- ...bypassing Linux TCP/IP isn’t the only way to get a fast Web server
- ...can be integrated with LVS, tc, IPtables, tcpdump etc.
Synchronous sockets: HTTP/TCP/IP stack

- HTTP is built into TCP/IP stack
- Everything is processing in softirq (while the data is hot)
- No input queue
- No file descriptors
- Less locking
Synchronous sockets: HTTP/TCP/IP stack

- HTTP is built into TCP/IP stack
- Everything is processing in softirq (while the data is hot)
- No input queue
- No file descriptors
- Less locking
- Lock-free inter-CPU transport
- => faster socket reading
- => lower latency
Synchronous sockets: performance

Fast HTTP parser

  - 1.6-1.8 times faster than Nginx’s

  - ~1KB strings:
  - `Strncasecmp()` ~x3 faster than GLIBC’s
  - URI matching ~x6 faster than GLIBC’s `strspn()`

```python
while (1):
    STATE_1:
        switch (*str_ptr) {
            case 'a':
                ...
                ++str_ptr
                goto STATE_1
            case 'b':
                ...
                ++str_ptr
                STATE_2:
                ...
```
### TempestaDB

- Web cache
- Firewall rules
  - Cache conscious Burst Hash Trie
  - Lock-free index
    (data blocks still have locks)
  - Huge pages
  - NUMA aware (replication or shardning)
TempestaDB: memory optimizations

- Cache conscious Burst Hash Trie
  - short offsets instead of pointers
  - (almost) lock-free
- lock-free block allocator for virtually contiguous memory
Burst Hash Trie

"key_foo" → 0xabc123456

root (unsigned short[16] = cache line)
Burst Hash Trie

"key_foo" \rightarrow 0xabc1234567

root (unsigned short[16] = cache line)

Data page

key_foo: record_foo
Burst Hash Trie

"key_bar" -> 0x80c3491ed7

root (unsigned short[16] = cache line)

BURST!

Data page

key_foo: record_foo
key_zoo: record_zoo
key_xyz: record_xyz
Burst Hash Trie
Frang: HTTP DoS

- **Rate limits**
  - request_rate, request_burst
  - connection_rate, connection_burst
  - concurrent_connections

- **Slow HTTP**
  - client_header_timeout, client_body_timeout
  - http_header_cnt
  - http_header_chunk_cnt, http_body_chunk_cnt
Frang: WAF

- **Length limits**: `http_uri_len`, `http_field_len`, `http_body_len`
- **Content validation**: `http_host_required`, `http_ct_required`, `http_ct_vals`, `http_methods`
- **HTTP Response Splitting**: count and match requests and responses
- **Injections**: carefully verify allowed character sets
- ...and many upcoming filters: [https://github.com/tempesta-tech/tempesta/labels/security](https://github.com/tempesta-tech/tempesta/labels/security)
- Not a featureful WAF
Load balancing

- Dynamic **reconnections**
- Configurable number of upstream **keep-alive** connections
- Configurable **non-idempotent** requests handling
- Schedulers
  - **HTTP** (server groups):
    - Method, URI, Host & other headers, wildcards, full match, prefix
  - **Rendezvous hashing** (<Method, URI,Host>, inside server group)
  - **Ratio** (weighted round-robin, inside server group)
  - **Adaptive & predictive** load balancing
Load balancing: configuration example

```
srv_group static { # sched=round-robin
    server 10.10.0.1:8080;
    server [fc00::2]:8081;
}
	srv_group dynamic sched=hash {
    server 10.10.0.3:8080; # conns_n = 4
    server [fc00::4]:8081 conns_n=32;
}
	srv_group black_hole { }

sched_http_rules {
    match black_hole hdr_raw prefix "X-Bad:";
    match static uri prefix "/static/";
    match dynamic * * *
}
```
Sticky cookie

- User/session identification
  - Cookie challenge for dummy DDoS bots
  - Persistent/sessions scheduling (no rescheduling on a server failure)
- **Enforce:** HTTP 302 redirect

```java
sticky name=__tfw_user_id__ enforce;
```
Prerequisites

- Haswell: AVX2, SSE 4.2 ("avx2", "sse4_2" in /proc/cpuinfo)
- Huge pages ("pse" in /proc/cpuinfo)
- Custom Linux kernel (KVM or dedicated server)
Build the kernel

$ git clone https://github.com/tempesta-tech/linux-4.8.15-tfw.git
$ cd linux-4.8.15-tfw
$ make && make modules && make modules_install && make install
$ reboot
Build & run

$ git clone https://github.com/tempesta-tech/tempesta.git
$ cd tempesta && make
$ cat > etc/tempesta_fw.conf

  server 127.0.0.1:8080;  # upstream
  cache 1;              # cache sharding

^D

$ ./scripts/tempesta.sh --start
Is it safe to live in kernel?

- Just **30K LoC** (compare w/ 120K LoC of Btrfs)
- Tests, tests, tests, tests, tests, tests...
- Mandatory code reviews
- Upcoming zero-copy **kernel-user space transport** for minimizing kernel code

**Usability:**
- Debian and CentOS packages in 0.5 (current)
- Full Linux distribution in 0.6
Why Tempesta FW?

- Faster than user space Web-accelerators
- Built-in filtering to block L7 DDoS and Web application attacks
- Many HTTP schedulers
Thanks!

- Web-site: http://tempesta-tech.com (Powered by Tempesta FW)
- Availability: https://github.com/tempesta-tech/tempesta
- Blog: http://natsys-lab.blogspot.com
- E-mail: ak@tempesta-tech.com