BIND 9 Security

(Part 3 - eBPF - extended Berkeley Packet Filter)

Carsten Strotmann and the ISC Team

Welcome

Welcome to part three of our BIND 9 security webinar series

In this Webinar

- The Berkeley Packet Filter
- eBPF Architecture
- Instrumenting the Linux Network Stack
- Instrumenting BIND 9
- Packet Filtering with eBPF
- Hands-On lab

The Berkeley Packet Filter

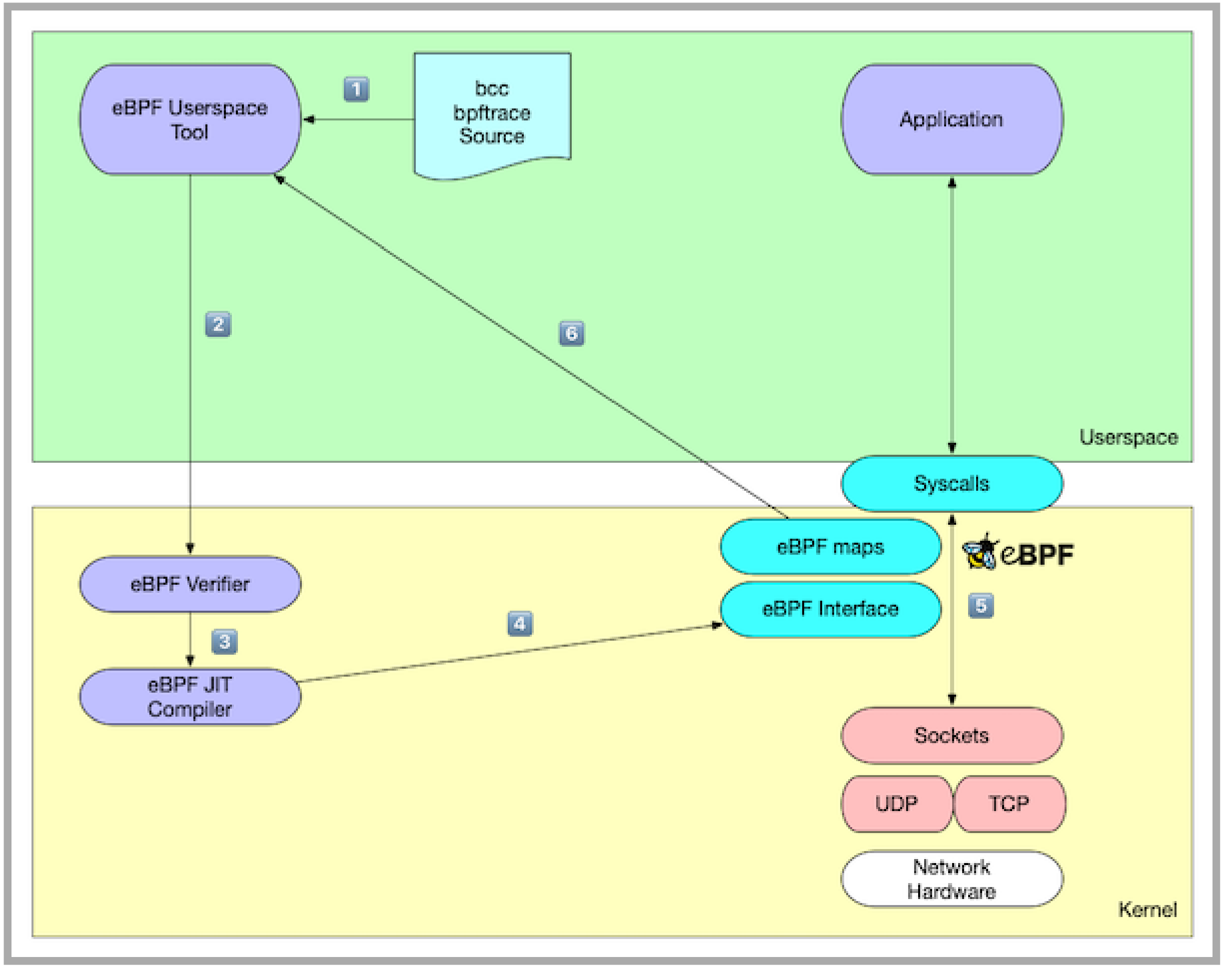
What is BPF/eBPF?

- eBPF is the *extended Berkeley Packet Filter* infrastructure inside the Linux kernel
- eBPF is a further development of the Berkeley Packet Filter technology https://en.wikipedia.org/wiki/Berkeley_Packet_Filter

The eBPF idea

- eBPF allows the administrator to execute sandbox programs inside the operating system kernel
  - eBPF is used to extend the capabilities of the kernel safely, securely and efficiently without modifying the kernel source code or loading kernel modules
  - eBPF can monitor and manipulate network packets as well as other data inside Linux kernel
  - eBPF programs are **not** kernel modules, you don't need to be a Kernel *developer* to work with eBPF
    - but some C programming knowledge is helpful

eBPF


eBPF Userspace Tool
1
bcc bpftrace Source
Application
2
6
Userspace
Syscalls
eBPF maps
eBPF
eBPF Verifier
eBPF Interface
5
4
3
eBPF JIT Compiler
Sockets
UDP
TCP
Network Hardware
Kernel

eBPF use cases

- Use cases for eBPF
  - Network security (advanced firewall functions)
  - Host security
  - Forensics
  - Fault diagnosis
  - Performance measurements
- eBPF is available on modern Linux systems (Kernel 3.18+) and is currently being ported to the Windows operating systems ported by Microsoft

Origins of BPF

- The original BSD Packet Filter (BPF) has been designed by Steven McCanne and Van Jacobson at Lawrence Berkeley Laboratory (https://www.tcpdump.org/papers/bpf-usenix93.pdf)
    - BPF has been ported to almost all Unix/Linux and some non-Unix operating systems
    - BPF is the base technology for some well known network sniffing tools such as `tcpdump` and *Wireshark*

BPF operation using tcpdump as an example

- When using a BPF-enabled tool, the filter code is compiled into bytecode for the BPF in-kernel VM and loaded into the kernel
  - The operating system kernel will execute the filter program for every network packet that traverses the network stack
  - Only packets that match the filter expression will be forwarded to the userspace tool, `tcpdump` in this example
  - BPF helps limiting the amount of data that needs to be sent between kernel and user space

BPF operation using tcpdump as an example

`tcpdump` can be instructed to emit the source code for a `tcpdump` filter expression

```
# tcpdump -d port 53 and host 1.1.1.1
Warning: assuming Ethernet
(000) ldh      [12]
(001) jeq      #0x86dd          jt 19   jf 2
(002) jeq      #0x800           jt 3    jf 19
(003) ldb      [23]
(004) jeq      #0x84            jt 7    jf 5
(005) jeq      #0x6             jt 7    jf 6
(006) jeq      #0x11            jt 7    jf 19
(007) ldh      [20]
(008) jset     #0x1fff          jt 19   jf 9
(009) ldxb     4*([14]&0xf)
(010) ldh      [x + 14]
(011) jeq      #0x35            jt 14   jf 12
(012) ldh      [x + 16]
(013) jeq      #0x35            jt 14   jf 19
(014) ld       [26]
(015) jeq      #0x1010101       jt 18   jf 16
(016) ld       [30]
(017) jeq      #0x1010101       jt 18   jf 19
(018) ret      #262144
(019) ret      #0
```

eBPF vs. BPF

- While BPF (or now called cBPF = classic BPF) filters network packets inside the operating system kernel, eBPF does also filter on
  - Kernel systemcalls
  - Kernel tracepoints
  - Kernel functions
  - Userspace tracepoints
  - Userspace functions

eBPF and the Linux kernel

- The basic eBPF was introduced into the Linux kernel in version 3.18
  - since then, most new kernel release implemented new eBPF functions
  - Linux distributions might have backported eBPF functions into older LTS kernel (Red Hat/Canonical/Suse)
  - Overview of eBPF functions by Linux kernel version: https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

The eBPF Architecture

The eBPF VM

- eBPF programs are compiled for a virtual CPU
- The code is loaded and verified in the Linux kernel
- On main architectures, the eBPF code is re-compiled into native code (Just in time compiler)

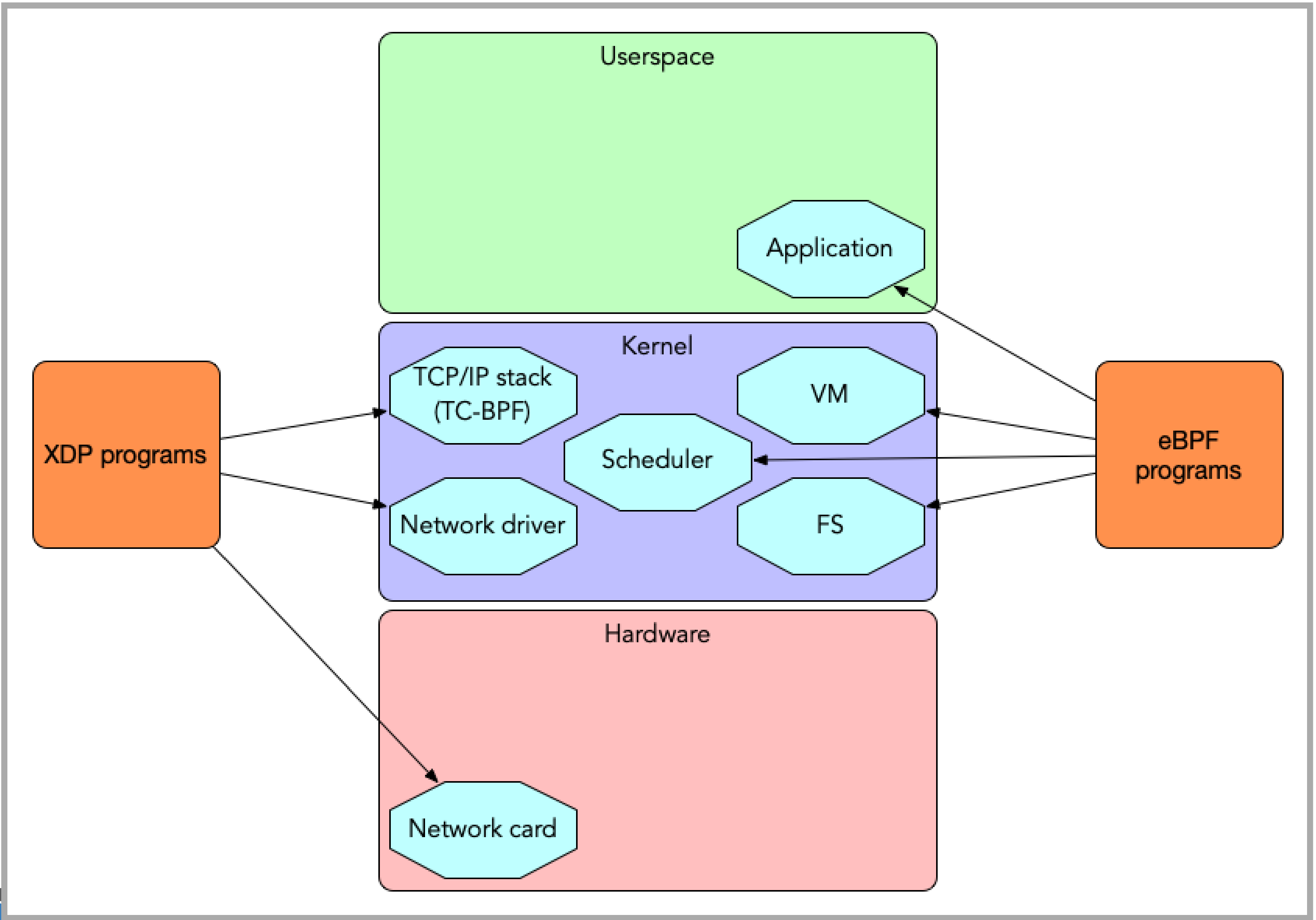
XDP - express data path

- The *express data path* (XDP) inside the Linux-Kernel is an infrastructure to gain low level control over network traffic
  - side-stepping the normal kernel network stack flow
  - eBPF programs can be loaded into the eXpress Data Path (XDP)

XDP / eBPF hardware offloading

- XDP eBPF can be loaded into different level of the Linux kernel network stack
  - **Offload XDP**: directly into the network hardware (ASIC/FPGA, needs support by the network hardware, for example Netronome NIC)
  - **Native XDP**: into the network driver (low level Linux kernel code, requires support by the driver)
  - **Generic XDP**: into the Linux kernel network stack (less performance, but universally available)

XDP / eBPF execution environments

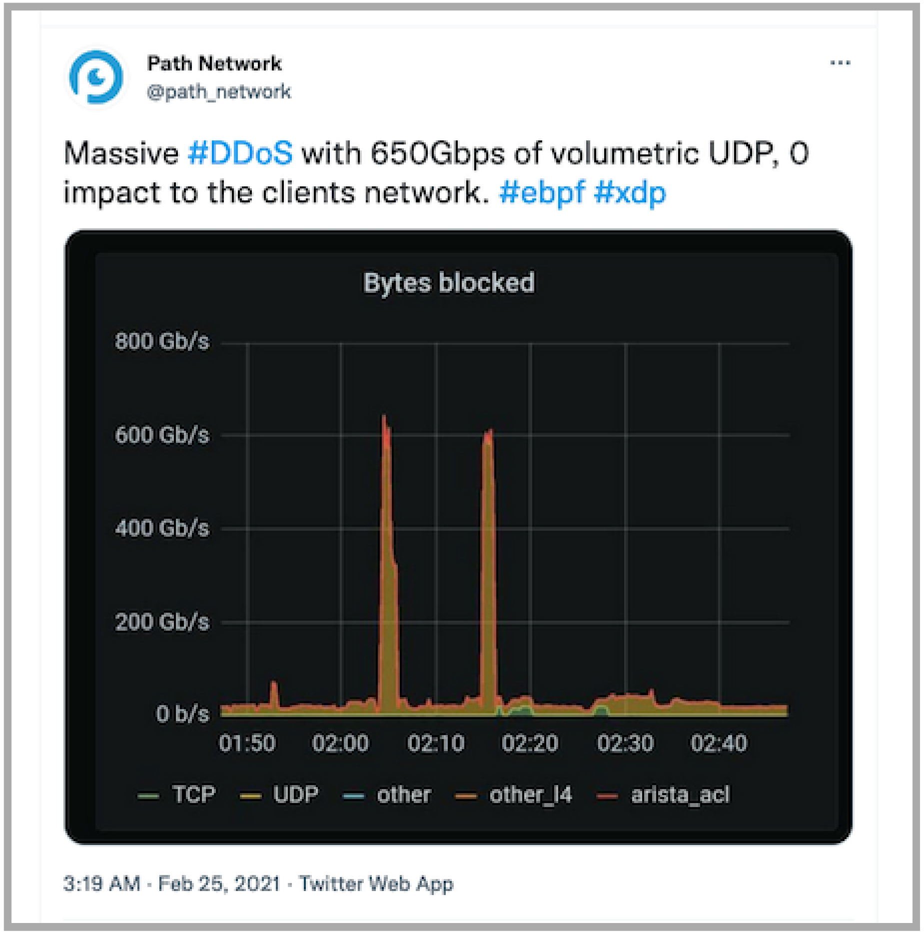


XDP functions

- XDP programs can
  - **read** network packets and collect statistics
  - **modify** the content of network packets
  - **drop** selected traffic (firewall)
  - **redirect** packets to the same or other network interfaces (switching/routing)
  - **pass** the network packet to the Linux TCP/IP stack for normal processing

XDP vs DDoS attack

- XDP can discard unwanted traffic very early in the network stack, defending against DDoS attacks



eBPF/XDP support in DNS software

- DNSdist (see Webinar Practical BIND 9 Management - Session 3: Load-balancing with dnsdist) can directly rate limit or block DNS traffic through eBPF and XDP
- The Knot resolver (since version 5.2.0) can bypass the Linux TCP/IP stack and send DNS traffic direct to the user space process (https://knot-resolver.readthedocs.io/en/stable/daemon-bindings-net_xdpsrv.html)

Using eBPF

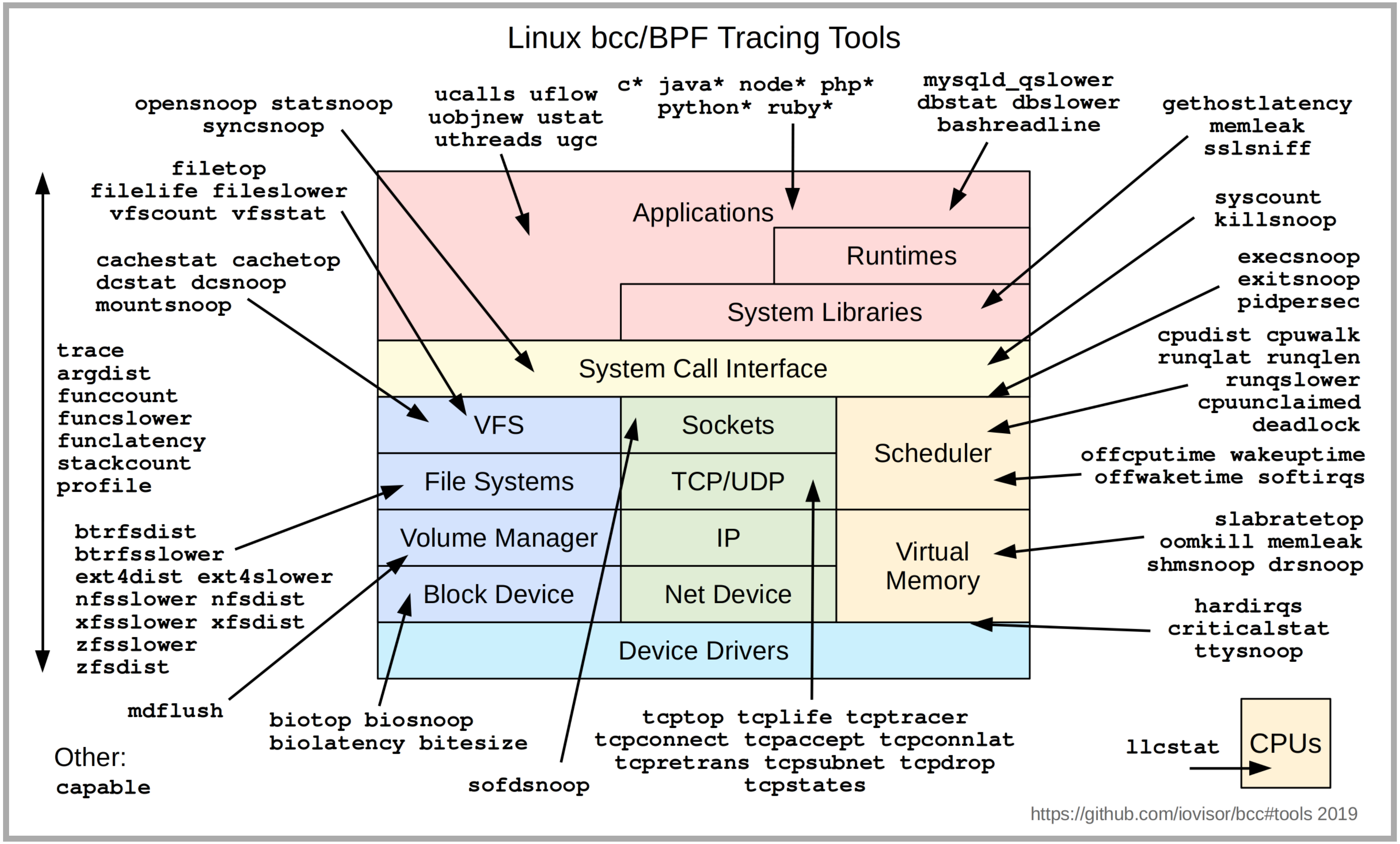
eBPF tooling

- eBPF programs can be written in many ways
  - Low level eBPF assembly code
  - High Level compiler (using LLVM): C / GO / Rust / Lua / Python …
  - Special "scripting" languages: `bpftrace`

BCC

- BCC is the BPF compiler collection
  - Website https://github.com/iovisor/bcc
  - BCC compiles C or Python code into eBPF programs and loads them into the Linux kernel

BCC tools



Linux bcc/BPF Tracing Tools
opensnoop statsnoop syncsnoop
ucalls uflow uobjnew ustat uthreads ugc
c* java* node* php* python* ruby*
mysqld_qslower dbstat dbslower bashreadline
gethostlatency memleak sslsniff
filetop filelife fileslower vfscount vfsstat
cachestat cachetop dcstat dcsnoop mountsnoop
syscount killsnoop
execsnoop exitsnoop pidpersec
cpudist cpuwalk runqlat runqlen runqslower cpuunclaimed deadlock
offcputime wakeuptime offwaketime softirqs
trace argdist funccount funcslower funclatency stackcount profile
btrfsdist btrfsslower ext4dist ext4slower nfsslower nfsdist xfsslower xfsdist zfsslower zfsdist
slabratetop oomkill memleak shmsnoop drsnoop
hardirqs criticalstat ttysnoop
Applications
Runtimes
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device
Net Device
Device Drivers
mdflush
biotop biosnoop biolatency bitesize
tcptop tcplife tcptracer tcpconnect tcpaccept tcpconnlat tcpretrans tcpsubnet tcpdrop tcpstates
llcstat
CPUs
Other:
capable
sofdsnoop
https://github.com/iovisor/bcc#tools 2019

BCC Tool examples (1/2)

- Count syscalls from the BIND 9 process with `syscount`

```
# syscount-bpfcc -p `pgrep named` -i 10
Tracing syscalls, printing top 10... Ctrl+C to quit.
[07:34:19]
SYSCALL                   COUNT
futex                       547
getpid                      121
sendto                      113
read                         56
write                        31
epoll_wait                   31
openat                       23
close                        20
epoll_ctl                    20
recvmsg                      20
```

BCC Tool examples (2/2)

- Tracing Linux capability checks

```
# capable-bpfcc | grep named
07:36:17  0      29378  (named)          24   CAP_SYS_RESOURCE     1
07:36:17  0      29378  (named)          24   CAP_SYS_RESOURCE     1
07:36:17  0      29378  (named)          12   CAP_NET_ADMIN        1
07:36:17  0      29378  (named)          21   CAP_SYS_ADMIN        1
07:36:17  0      29378  named            6    CAP_SETGID           1
07:36:17  0      29378  named            6    CAP_SETGID           1
07:36:17  0      29378  named            7    CAP_SETUID           1
07:36:17  109    29378  named            24   CAP_SYS_RESOURCE     1
```

bpftrace

- `bpftrace` is a little language similar to `awk` or `dtrace`
  - Website https://bpftrace.org
- `bpftrace` programs subscribe to eBPF probes and executes a function whenever an event occurs (systemcall, function-call)
- `bpftrace` comes with many helper functions to handle eBPF data structures
- `bpftrace` allows one to write eBPF programs in a more concise way compared to BCC

Instrumenting the Linux Network Stack

BCC and bpftrace tools

- Literally hundreds of little eBPF programs exists to look deep into the Linux network stack
  - The BCC example tools
  - The `bpftrace` examples
  - Examples from eBPF books

gethostlatency

- The BCC tool `gethostlatency` measures the latency of client DNS name resolution through function calls such as `getaddrinfo` or `gethostbyname`

```
# gethostlatency-bpfcc
TIME      PID    COMM                  LATms HOST
10:21:58  19183  ping                 143.22 example.org
10:22:18  19184  ssh                    0.03 host.example.de
10:22:18  19184  ssh                   60.59 host.example.de
10:22:35  19185  ping                  23.44 isc.org
10:22:49  19186  ping                4459.72 yahoo.co.kr
```

netqtop

- `netqtop` - Summarize PPS, BPS, average size of packets and packet counts ordered by packet sizes on each queue of a network interface.

```
# netqtop-bpfcc -n eth0 -i 10
Mon Nov 15 07:43:29 2021
TX
 QueueID     avg_size    [0, 64)     [64, 512)  [512, 2K)  [2K, 16K)  [16K, 64K)
 0           297.82      2           48         1          4          0
 Total       297.82      2           48         1          4          0

RX
 QueueID     avg_size    [0, 64)     [64, 512)  [512, 2K)  [2K, 16K)  [16K, 64K)
 0           70.95       43          34         0          0          0
 Total       70.95       43          34         0          0          0
-----------------------------------------------------------------------------
```

tcptracer

- Tracing TCP connections showing source and destination addresses and ports and the TCP state (accept, connect, close)

```
# tcptracer-bpfcc -p $(pgrep named)
Tracing TCP established connections. Ctrl-C to end.
T  PID    COMM             IP SADDR            DADDR            SPORT  DPORT
C  29404  isc-net-0000     4  127.0.0.1        127.0.0.1        41555  953
A  29378  isc-socket-0     4  127.0.0.1        127.0.0.1        953    41555
X  29404  isc-socket-0     4  127.0.0.1        127.0.0.1        41555  953
X  29378  isc-socket-0     4  127.0.0.1        127.0.0.1        953    41555
C  29378  isc-net-0000     4  46.101.109.138   192.33.4.12      43555  53
C  29378  isc-net-0000     4  46.101.109.138   192.33.4.12      33751  53
X  29378  isc-socket-0     4  46.101.109.138   192.33.4.12      43555  53
X  29378  isc-socket-0     4  46.101.109.138   192.33.4.12      33751  53
C  29378  isc-net-0000     4  46.101.109.138   193.0.14.129     38145  53
C  29378  isc-net-0000     4  46.101.109.138   192.33.14.30     40905  53
X  29378  isc-socket-0     4  46.101.109.138   193.0.14.129     38145  53
X  29378  isc-socket-0     4  46.101.109.138   192.33.14.30     40905  53
```

tcpconnlat

- Display the connection latency for outgoing TCP based DNS queries from a BIND 9 resolver (in this example a query for `microsoft.com txt`, which is too large for 1232 byte UDP)
  - `isc-net-0000` is the internal name of a BIND 9 thread

```
# tcpconnlat-bpfcc
PID    COMM         IP SADDR            DADDR            DPORT LAT(ms)
29378  isc-net-0000 4  46.101.109.138   193.0.14.129     53    37.50
29378  isc-net-0000 4  46.101.109.138   192.52.178.30    53    14.01
29378  isc-net-0000 4  46.101.109.138   199.9.14.201     53    8.48
29378  isc-net-0000 4  46.101.109.138   192.42.93.30     53    1.90
29378  isc-net-0000 4  46.101.109.138   40.90.4.205      53    14.27
29378  isc-net-0000 4  46.101.109.138   199.254.48.1     53    19.21
29378  isc-net-0000 4  46.101.109.138   192.48.79.30     53    7.66
29378  isc-net-0000 4  46.101.109.138   192.41.162.30    53    7.97
29396  isc-net-0000 4  127.0.0.1        127.0.0.1        53    0.06
```

udplife

- A `bpftrace` script to trace UDP session lifespans (DNS round trip time) with connection detail (by Brendan Gregg, see link collection)

```
# udplife.bt
Attaching 8 probes...
PID   COMM       LADDR           LPORT RADDR           RPORT   TX_B   RX_B MS
29378 isc-net-00 46.101.109.138  0     199.19.57.1     16503     48    420 268
29378 isc-net-00 46.101.109.138  0     51.75.79.143    81        49     43 13
29378 isc-net-00 46.101.109.138  0     199.6.1.52      16452     48    408 24
29378 isc-net-00 46.101.109.138  0     199.249.120.1   81        44     10 9
29378 isc-net-00 46.101.109.138  0     199.254.31.1    32891     64     30 273
29378 isc-net-00 46.101.109.138  0     65.22.6.1       32891     64     46 266
```

Server agnostic DNS augmentation using eBPF

- A master thesis by Tom Carpay (supported by NLnet Labs)
  - eBPF Query-Name rewriting
  - In-Kernel DNS server agnostic response rate limiting (RRL)
- https://www.nlnetlabs.nl/downloads/publications/DNS-augmentation-with-eBPF.pdf

Instrumenting BIND 9

Use case -> Forward logging

- A BIND 9 DNS resolver has forward zones configured:

```
zone "dnslab.org" {
        type forward;
        forwarders { 1.1.1.1; 8.8.8.8; };
};
```

- The BIND 9 logging subsystem, while very powerful, does not support the logging of forwarding decisions
- Goal: Create a `bpftrace` script that writes out BIND 9 forwarding decisions

Step 1 - Use the ~~force~~ source

- The BIND 9 source code is public, available on the ISC gitlab service https://gitlab.isc.org
- A search through the source for *forwarding* finds the function `dns_fwdtable_find` in `/lib/dns/forward.c`. This sounds promising:

```
isc_result_t
dns_fwdtable_find(dns_fwdtable_t *fwdtable, const dns_name_t *name,
                  dns_name_t *foundname, dns_forwarders_t **forwardersp) {
        isc_result_t result;

        REQUIRE(VALID_FWDTABLE(fwdtable));

        RWLOCK(&fwdtable->rwlock, isc_rwlocktype_read);

        result = dns_rbt_findname(fwdtable->table, name, 0, foundname,
                                  (void **)forwardersp);
        if (result == DNS_R_PARTIALMATCH) {
                result = ISC_R_SUCCESS;
        }

        RWUNLOCK(&fwdtable->rwlock, isc_rwlocktype_read);

        return (result);
}
```

Step 2 - A proof of concept test

- The function `dns_fwdtable_find` takes a domain name and returns `0` if the name must be resolved through forwarding, and a value greater than 0 if not
  - A quick `bpftrace` one-liner will validate that this indeed works:

```
bpftrace -e 'uretprobe:/lib/x86_64-linux-gnu/libdns-9.16.22-Debian.so:dns_fwdtable_find { print(retval)
```

Step 2 - A proof of concept test

```
root@ebpf-test:~# bpftrace -e 'uretprobe:/lib/x86_64-linux-gnu/libd
16.22-Debian.so:dns_fwdtable_find { print(retval) }'
Attaching 1 probe...
0
23
23
23
23
23
23
23
```

```
root@ebpf-test:~# dig @localhost ns200a.dnslab.org +short
167.172.136.154
root@ebpf-test:~# dig @localhost isc.org +short
149.20.1.66
root@ebpf-test:~#
```

Step 3 - Planning the probe script

- Now we are certain that we have a function to work with, we write a `bpftrace` script
- The script will
  - Store the domain name requested from `dns_fwdtable_find` when the function is called
  - Check the return code (`retval`) of the function when it returns, and print the domain name when the return value is zero (`0`), do nothing otherwise

Challenge - Wrangling with structs

- The domain name to check for forwarding is given to the function as a struct of type `dns_name_t`
    - It's not a simple pointer to a string that we can print
- A search through the ISC BIND 9 source code documentation reveals the structure of `dns_name_t`. The 2nd field is `unsigned char * ndata`, which looks like the domain name

Challenge - Wrangling with structs

- The definition of `dns_name_t` can be found in `lib/dns/include/dns/name.h`

```c

/*%
 * Clients are strongly discouraged from using this type directly,  with
 * the exception of the 'link' and 'list' fields which may be used directly
 * for whatever purpose the client desires.
 */
struct dns_name {
        unsigned int   magic;
        unsigned char *ndata;
        unsigned int   length;
        unsigned int   labels;
        unsigned int   attributes;
        unsigned char *offsets;
        isc_buffer_t  *buffer;
        ISC_LINK(dns_name_t) link;
        ISC_LIST(dns_rdataset_t) list;
};
```

Challenge - Wrangling with structs

- `bpftrace` uses a syntax similar to the C programming language, we can import the struct from the BIND 9 source code into the script
    - we don't need the linked list and the `isc_buffer_t` fields for our script, and these are not *native* types, so we comment these lines out

```
#!/usr/bin/bpftrace

struct dns_name {
        unsigned int   magic;
        unsigned char *ndata;
        unsigned int   length;
        unsigned int   labels;
        unsigned int   attributes;
        unsigned char *offsets;
//      isc_buffer_t  *buffer;
//      ISC_LINK(dns_name_t) link;
//      ISC_LIST(dns_rdataset_t) list;
};
[...]
```

Printing a message at probe start

- The `BEGIN` pseudo-probe fires at the start of the script and prints a message, informing the user that the script has been started

```
[...]
BEGIN
{
  print("Waiting for forward decision...\n");
}
[...]
```

Probing the function call

- This probe fires when the function is called
  - it's a `uprobe` (User-Space probe)
  - the function to be probed is `dns_fwdtable_find` in the dynamic library `/lib/x86_64-linux-gnu/libdns-9.16.22-Debian.so`
  - The 2nd argument to the call (`arg1`) is cast into a `struct dns_name`, and then the field `ndata` is referenced
  - This data is stored into the variable `@dns_name[tid]` indexed by the thread ID (`tid`) of the running thread

```
[...]
uprobe:/lib/x86_64-linux-gnu/libdns-9.16.22-Debian.so:dns_fwdtable_find
{
  @dns_name[tid] = ((struct dns_name *)arg1)->ndata
}
[...]
```

Probing the function exit

- The 3rd probe is firing at function exit (`uretprobe` - User-space function *return* probe)
    - Same library and function as before
- If the return value of the function is zero `0` (domain name needs to be forwarded), the stored data in `@dns_name[tid]` is converted into a string and printed out
- The variable `@dns_name[tid]` is deleted as it's not needed any longer

```
uretprobe:/lib/x86_64-linux-gnu/libdns-9.16.22-Debian.so:dns_fwdtable_find
{
 if (retval == 0) {
    printf("Forwarded domain name: %s\n", str(@dns_name[tid]));
 }
 delete(@dns_name[tid]);
}
```

The final script

```
#!/usr/bin/bpftrace

struct dns_name {
        unsigned int   magic;
        unsigned char *ndata;
        unsigned int   length;
        unsigned int   labels;
        unsigned int   attributes;
        unsigned char *offsets;
//      isc_buffer_t  *buffer;
//      ISC_LINK(dns_name_t) link;
//      ISC_LIST(dns_rdataset_t) list;
};

BEGIN
{
  print("Waiting for forward decision...\n");
}
uprobe:/lib/x86_64-linux-gnu/libdns-9.16.22-Debian.so:dns_fwdtable_find
{
  @dns_name[tid] = ((struct dns_name *)arg1)->ndata
}

uretprobe:/lib/x86_64-linux-gnu/libdns-9.16.22-Debian.so:dns_fwdtable_find
{
 if (retval == 0) {
    printf("Forwarded domain name: %s\n", str(@dns_name[tid]));
 }
 delete(@dns_name[tid]);
}
```

The script in operation

- The script *fires* whenever a domain name is to be forwarded
  - In this example, all queries for the domain `dnslab.org` are forwarded, but not `ietf.org`

```
root@ebpf-test:~# ./forward.bt
Attaching 3 probes...
Waiting for forward decision...

Forwarded domain name: zone203dnslaborg
Forwarded domain name: dnslaborg
Forwarded domain name: dnslaborg
Forwarded domain name: dnslaborg
Forwarded domain name: dnslaborg

--------------------------------------------------------------------------
root@ebpf-test:~# dig zone203.dnslab.org +short
137.184.150.214
root@ebpf-test:~# dig ietf.org +short
4.31.198.44
root@ebpf-test:~#
```

Packet Filtering with eBPF

eBPF as a network firewall

- eBPF can be a very efficient firewall
  - It can stop network packets before they enter the Linux TCP/IP stack or the userspace application
  - As eBPF runs full programs, the firewall can work on complex rules
    - DNS query names
    - DNSSEC data in answers
    - Source IP of nameserver
    - EDNS data (prioritize DNS messages with DNS cookies)
    - …

Example: Block-Non-DNS

- In the Hands-On part of this training, we show a simple eBPF network filter
  - Block all UDP traffic towards a network interface except DNS (Port 53)
  - Helps in non-DNS DDoS attacks against an authoritative DNS server

Example: XDP Firewall

- The XDP Firewall is a new project to create a firewall tool leveraging XDP
  - https://github.com/gamemann/XDP-Firewall
  - Example rule-set to block all DNS traffic on Port 53

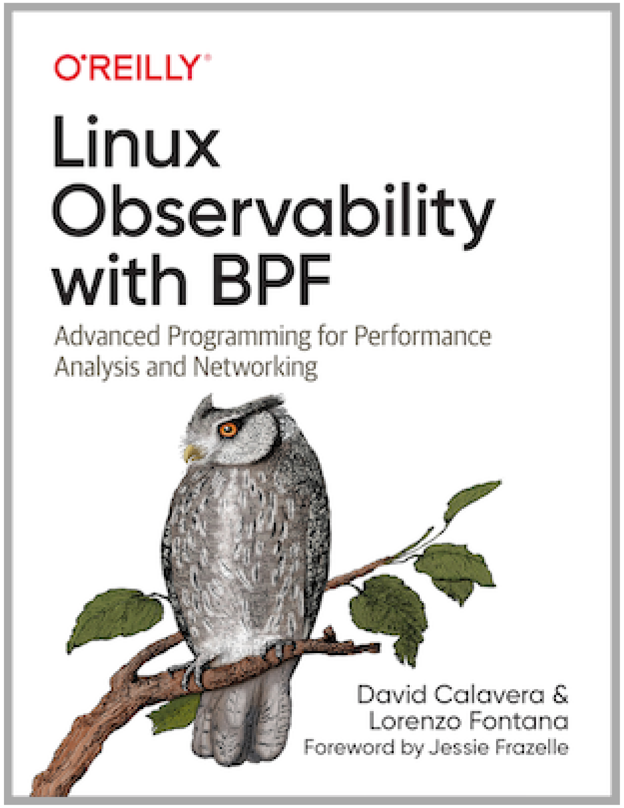
```
interface = "eth0";
updatetime = 15;

filters = (
    {
        enabled = true,
        action = 0,
        udp_enabled = true,
        udp_dport = 53
    }
);
```

Literature and Links

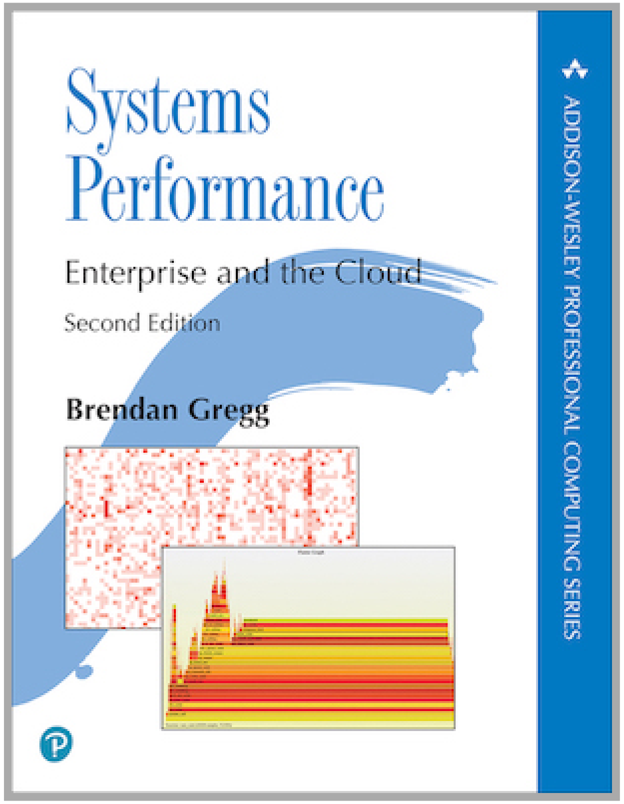
Book: Linux Observability with BPF

By David Calavera, Lorenzo Fontana (November 2019)



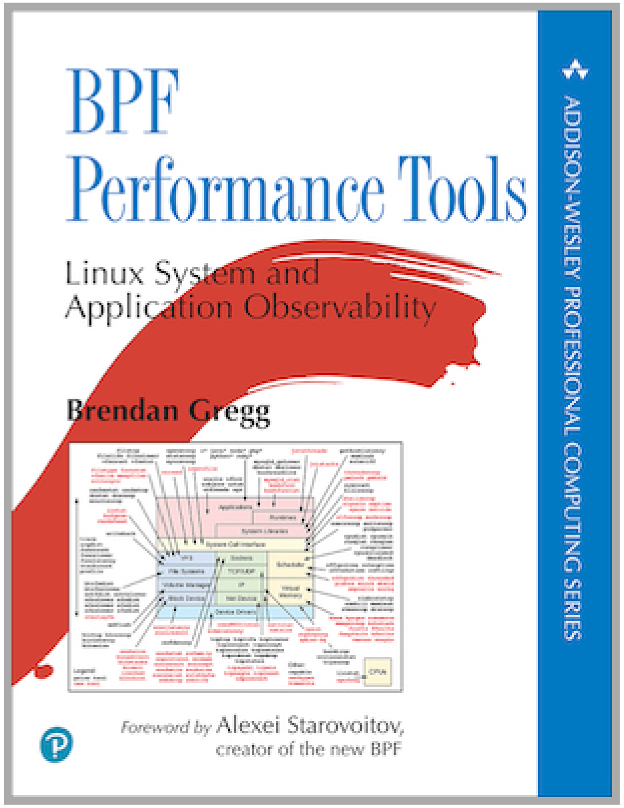
Book: Systems Performance (2nd ed.)

By Brendan Gregg (December 2020)



Book: BPF Performance Tools

By Brendan Gregg (December 2019)



Links

- For the webinar we have a extensive list of links that can be found at https://webinar.defaultroutes.de/webinar/08-ebpf-links.html

Next webinars

- December 15 - DNS Fragmentation: Real-World measurements, impact and mitigation

Questions and Answers

Hands-On

- We have prepared a VM machine for every participant
- Find the instructions at https://webinar.defaultroutes.de/webinar/08-ebpf-workshop.html