Speedrun through Splicing Sockets with SOCKMAP

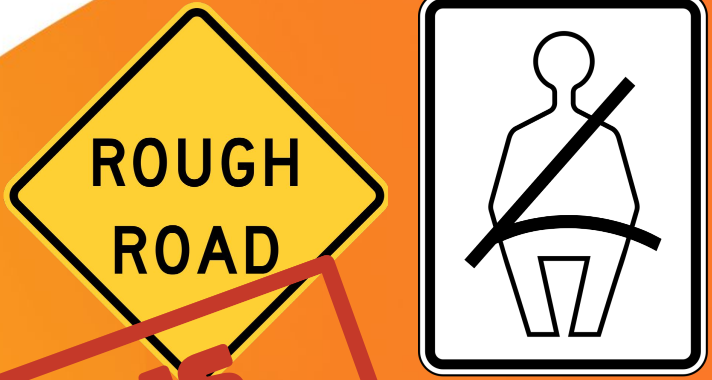
Jakub Sitnicki
Systems Engineer
Cloudflare

NDC { TechTown }

12 Sep 2024 | Kongsberg, Norway

CLOUDFLARE

Walk-through of ~~Speedrun through~~ Splicing Sockets with SOCKMAP



DIRECTOR'S CUT

Jakub Sitnicki
Systems Engineer
Cloudflare

NDC { TechTown }

12 Sep 2024 | Kongsberg, Norway

$ whoami

Linux / OS Team @ Cloudflare

🌽 roll out fresh kernels

🐞 squash bugs

🕵️ troubleshoot stuff

✨ prototype features

$ whoami

Linux / OS Team @ Cloudflare

- 🌽 roll out fresh kernels
- 🐞 squash bugs
- 🕵️ troubleshoot stuff
- ✨ prototype features

SOCKMAP co-maintainer @ Linux upstream

- 🤏 small-time (= feature) maintainer
- 🪲 fix bugs
- 🧐 review patches
- 🤔 answer questions

About this talk

Good to know:

- ❏ network programming (`socket`, `connect`, `sendmsg`, `recvmsg`)
- ❏ basics of eBPF (what are BPF maps, programs, hooks, `bpftool`)
- ❏ building blocks of containers (cgroups, namespaces)

Goals:

- ❏ know that SOCKMAP exists
- ❏ have idea how / when / what for you can use it
- ❏ feel ready to dive deeper

Agenda

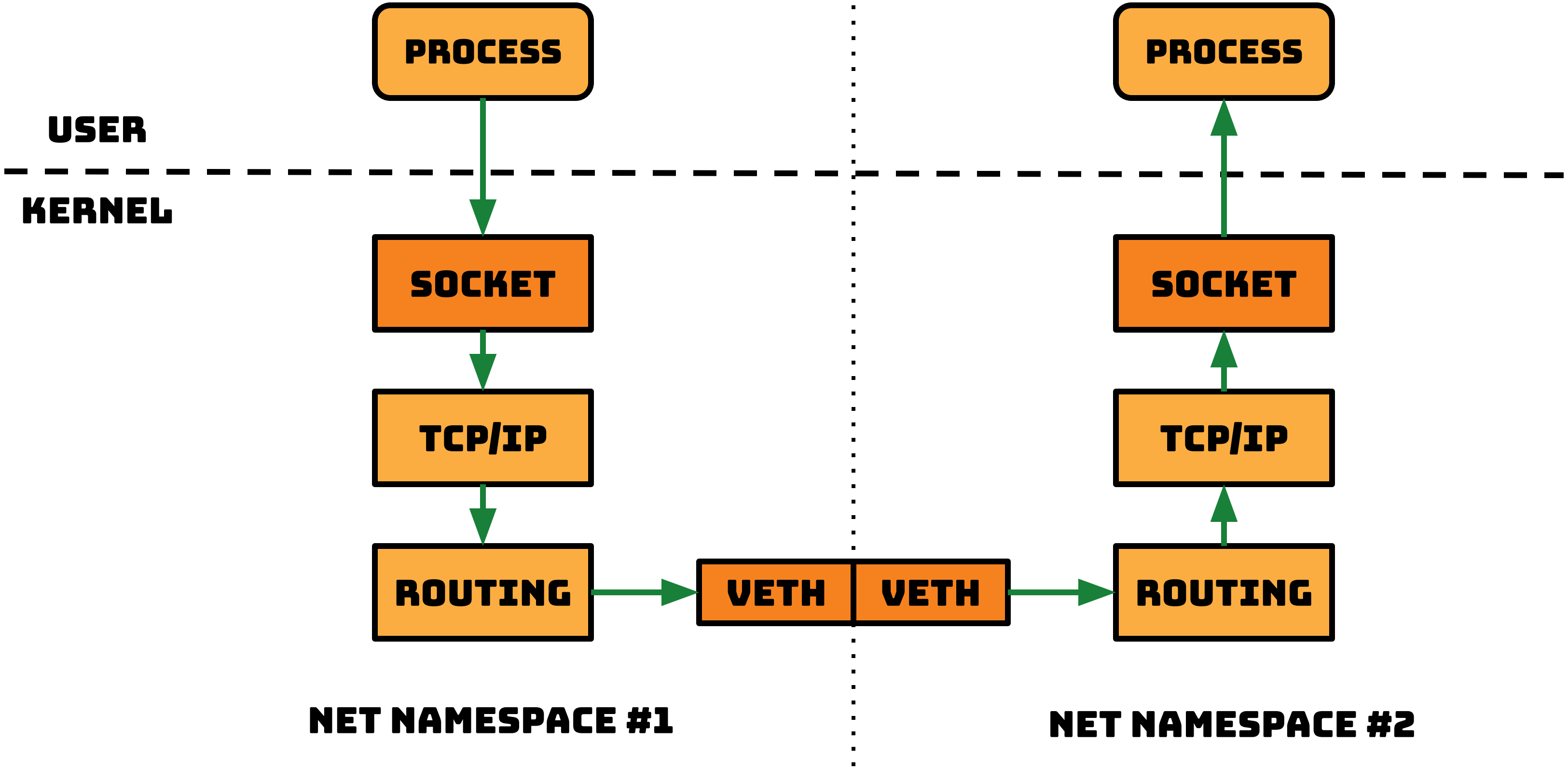
Agenda

NEW STUFF
* more benchmarks
* internal design
* how to trace it

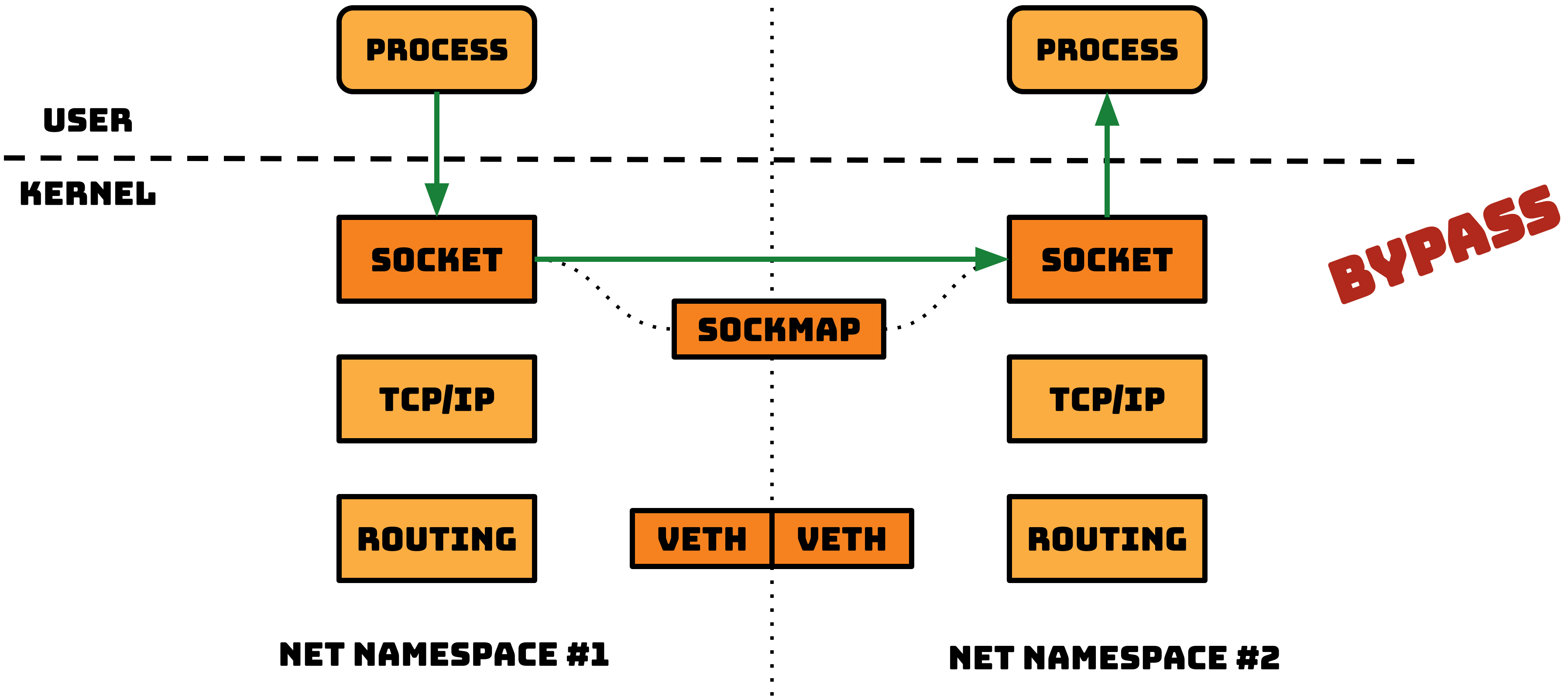
Section

What can SOCKMAP do for you?

What can SOCKMAP do for... container networking



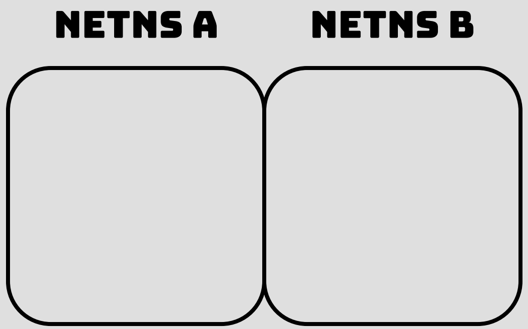
What can SOCKMAP do for… container networking



Let’s set it up!

Create two network namespaces

```
# ip netns add A
# ip netns add B
```



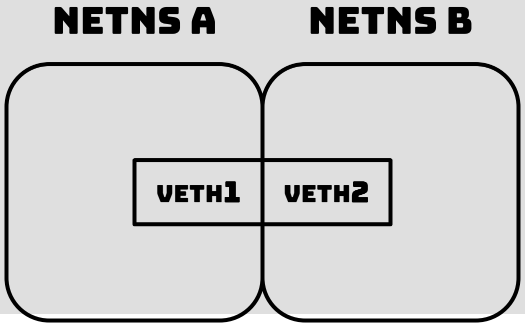
Let’s set it up!

Create two network namespaces

```
#  ip netns add A
#  ip netns add B
```

Link network namespaces with a veth pair

```
#  ip -n A link add name veth1 type veth  peer name veth2 netns B
```



Let’s set it up!

Create two network namespaces

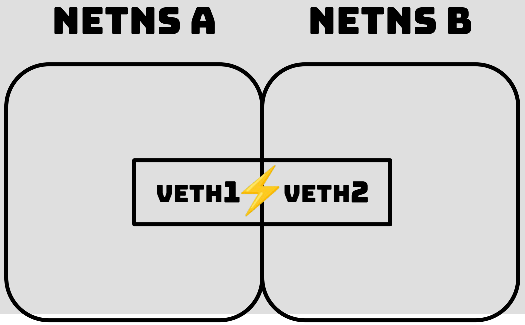
```
#  ip netns add A
#  ip netns add B
```

Link network namespaces with a veth pair

```
#  ip -n A link add name veth1 type veth peer name veth2 netns B
```

Bring up the links inside network namespaces

```
#  ip -n A link set dev veth1 up
#  ip -n B link set dev veth2 up
```



Let’s set it up!

Create two network namespaces

```
# ip netns add A
# ip netns add B
```

Link network namespaces with a veth pair

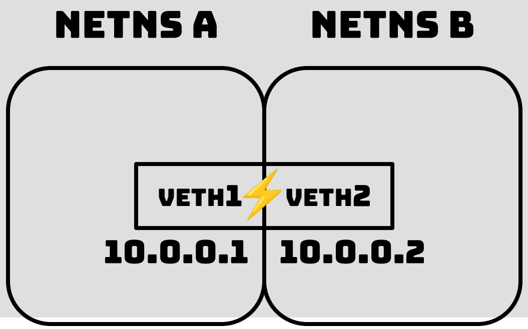
```
# ip -n A link add name veth1 type veth peer name veth2 netns B
```

Bring up the links inside network namespaces

```
# ip -n A link set dev veth1 up
# ip -n B link set dev veth2 up
```

Assign addresses to links inside network namespaces

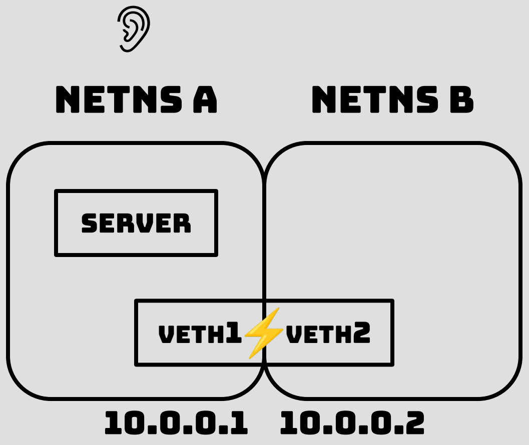
```
# ip -n A addr add 10.0.0.1/24 dev veth1
# ip -n B addr add 10.0.0.2/24 dev veth2
```



Measure latency, no SOCKMAP first

Run TCP server in netns A

```
# ip netns exec A \
  sockperf server -i 10.0.0.1 --tcp --daemonize
```



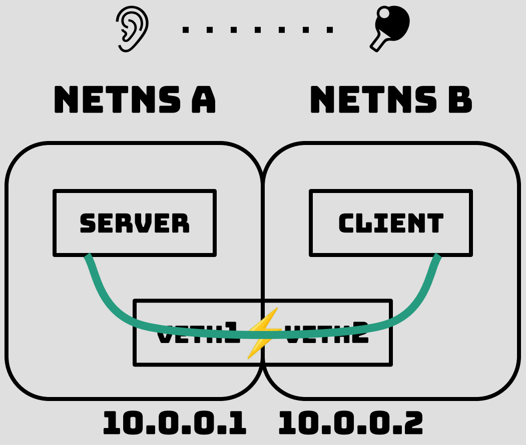
Measure latency, no SOCKMAP first

Run TCP server in netns A

```
# ip netns exec A \
  sockperf server -i 10.0.0.1 --tcp --daemonize
```

Run TCP client in netns B

```
# ip netns exec B \
  sockperf ping-pong -i 10.0.0.1 --tcp --time 30
```



Measure latency, no SOCKMAP first

Run TCP server in netns A

```
# ip netns exec A \
  sockperf server -i 10.0.0.1 --tcp --daemonize
```

Run TCP client in netns B

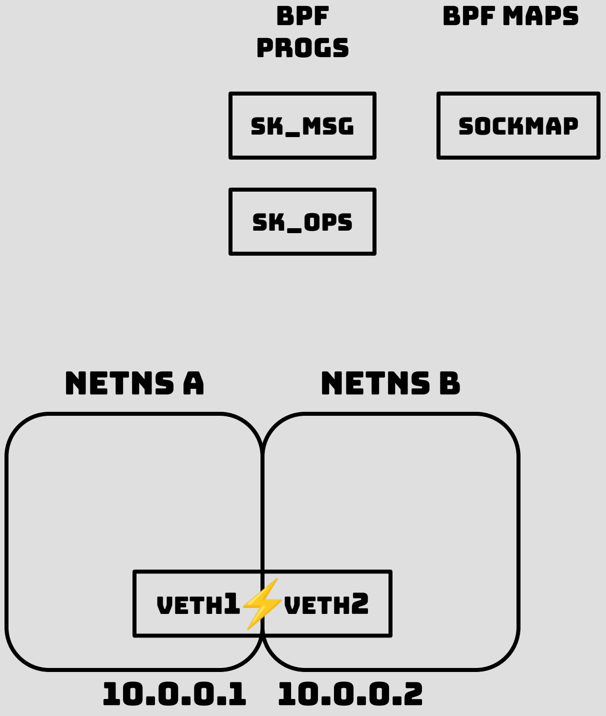
```
# ip netns exec B \
  sockperf ping-pong -i 10.0.0.1 --tcp --time 30
…
sockperf: [Total Run] RunTime=30.000 sec; Warm up time=400 msec; SentMessages=2599753;
ReceivedMessages=2599752
…
sockperf: ====> avg-latency=5.748 (std-dev=2.010, mean-ad=0.322, median-ad=0.220,
siqr=0.239, cv=0.350, std-error=0.001, 99.0% ci=[5.745, 5.751])
sockperf: # dropped messages = 0; # duplicated messages = 0; # out-of-order messages = 0
sockperf: Summary: Latency is 5.748 usec
```

5.8 ± 2.0 μsec

Set up SOCKMAP bypass

Load BPF programs and create BPF maps

```
# bpftool prog loadall \
  redir_bypass.bpf.o /sys/fs/bpf pinmaps /sys/fs/bpf
```



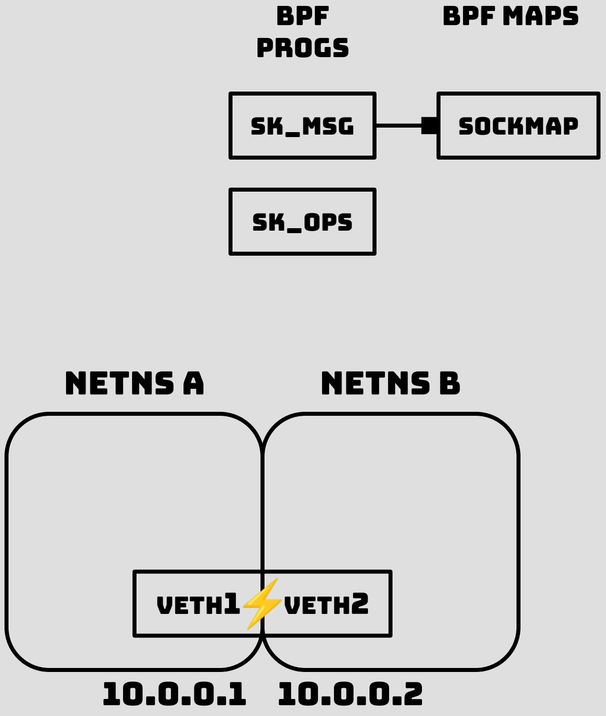
Set up SOCKMAP bypass

Load BPF programs and create BPF maps

```
# bpftool prog loadall \
  redir_bypass.bpf.o /sys/fs/bpf pinmaps /sys/fs/bpf
```

Attach BPF program to BPF map

```
# bpftool prog attach \
  pinned /sys/fs/bpf/sk_msg_prog sk_msg_verdict \
  pinned /sys/fs/bpf/sock_map
```



Set up SOCKMAP bypass

Load BPF programs and create BPF maps

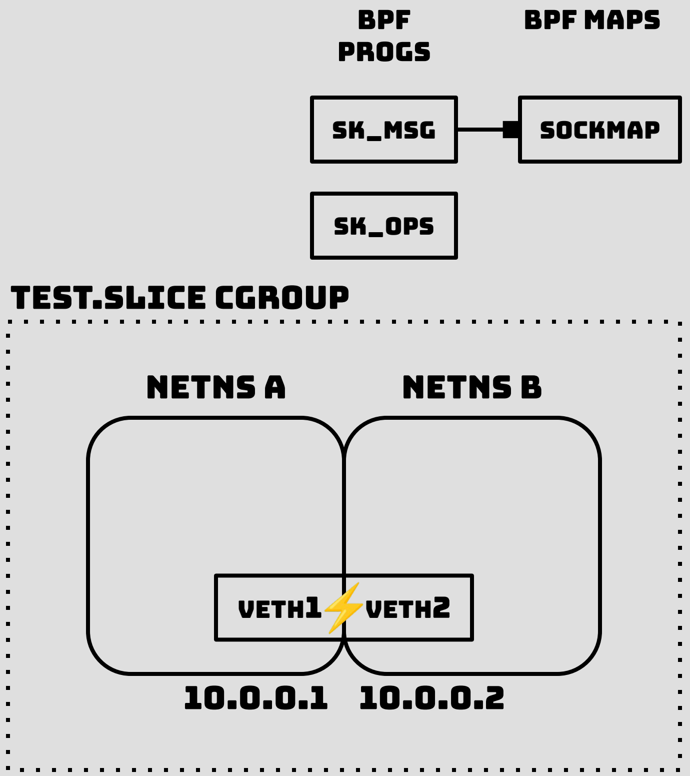
```
# bpftool prog loadall \
  redir_bypass.bpf.o /sys/fs/bpf pinmaps /sys/fs/bpf
```

Attach BPF program to BPF map

```
# bpftool prog attach \
  pinned /sys/fs/bpf/sk_msg_prog sk_msg_verdict \
  pinned /sys/fs/bpf/sock_map
```

Create a test cgroup

```
# mkdir /sys/fs/cgroup/test.slice
```



Set up SOCKMAP bypass

Load BPF programs and create BPF maps

```
# bpftool prog loadall redir_bypass.bpf.o /sys/fs/bpf \
               pinmaps /sys/fs/bpf
```

Attach BPF program to BPF map

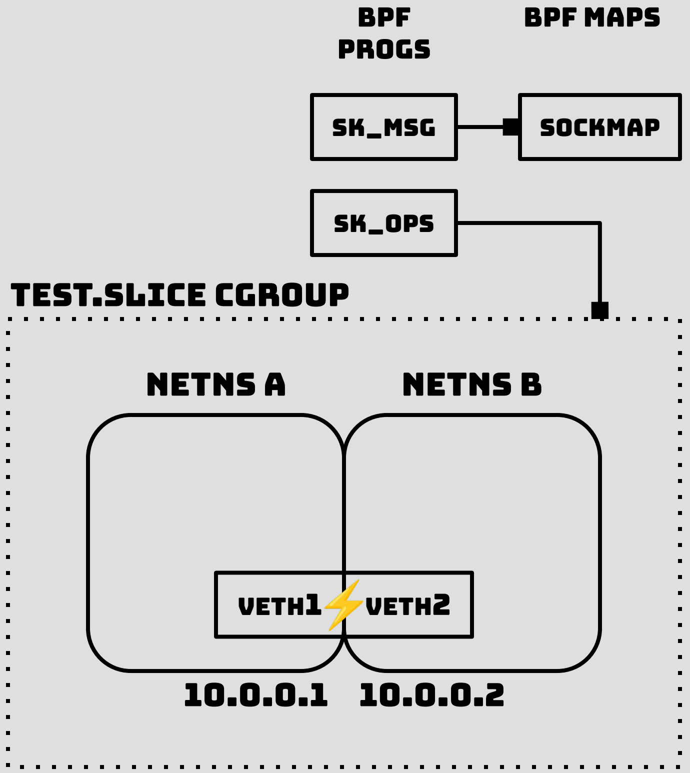
```
# bpftool prog attach \
  pinned /sys/fs/bpf/sk_msg_prog sk_msg_verdict \
  pinned /sys/fs/bpf/sock_map
```

Create a test cgroup

```
# mkdir /sys/fs/cgroup/test.slice
```

Attach BPF program to cgroup

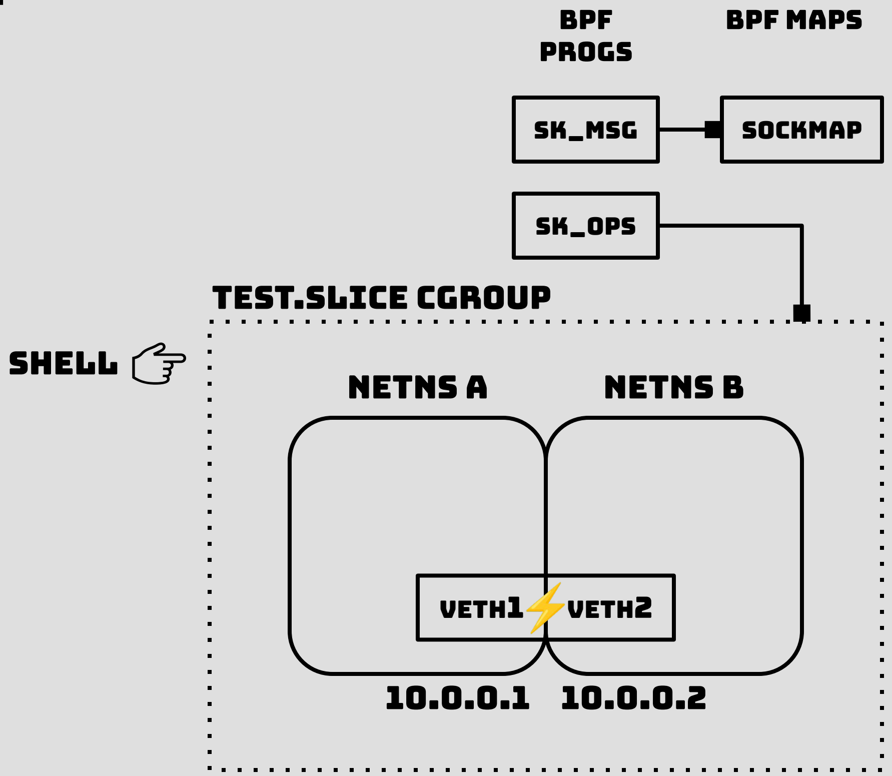
```
# bpftool cgroup attach \
  /sys/fs/cgroup/test.slice \
  cgroup_sock_ops pinned /sys/fs/bpf/sockops_prog
```



Repeat test with SOCKMAP bypass

Spawn client and server inside the test cgroup

```
# echo $$ > /sys/fs/cgroup/test.slice/cgroup.procs
```



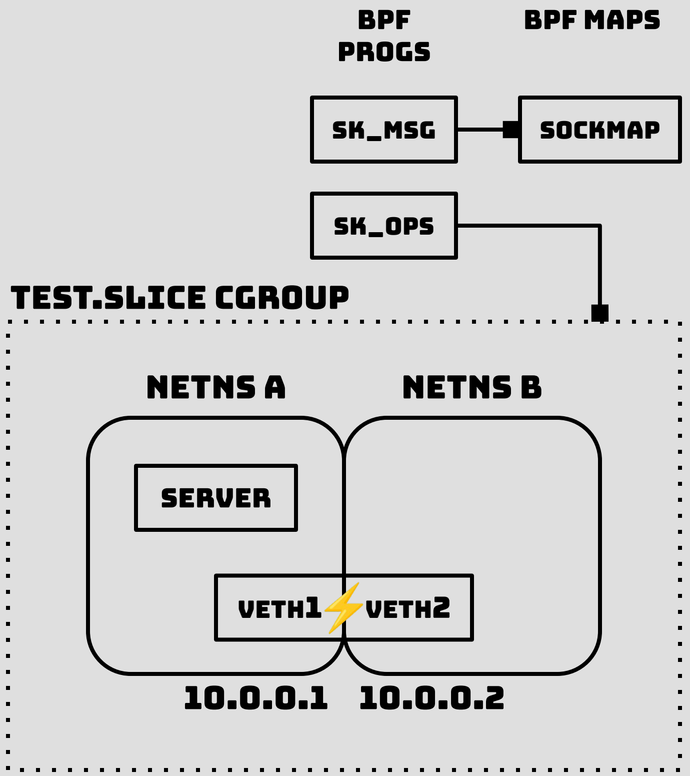
Repeat test with SOCKMAP bypass

Spawn client and server inside the test cgroup

```
# echo $$ > /sys/fs/cgroup/test.slice/cgroup.procs
```

Run TCP server in netns A

```
# ip netns exec A \
  sockperf server -i 10.0.0.1 --tcp --daemonize
```



Repeat test with SOCKMAP bypass

Spawn client and server inside the test cgroup

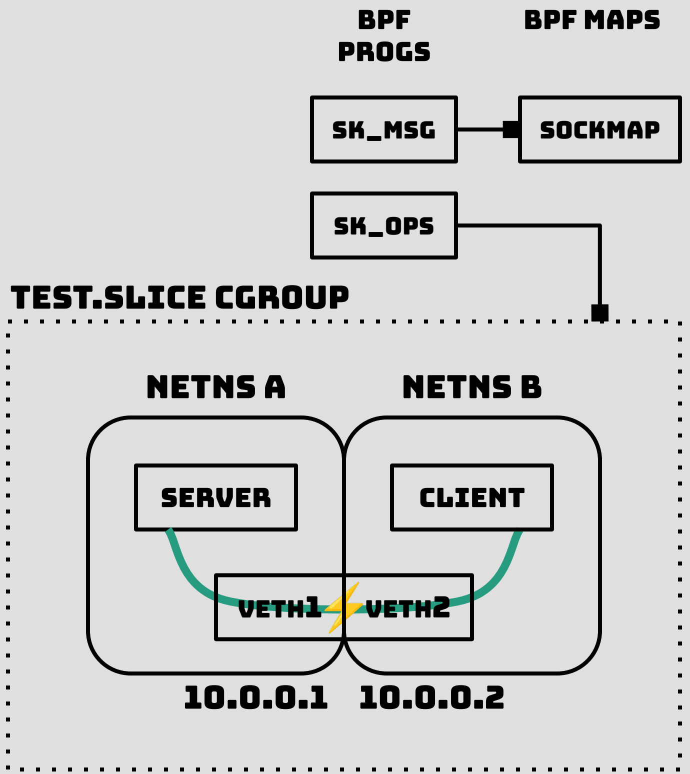
```
# echo $$ > /sys/fs/cgroup/test.slice/cgroup.procs
```

Run TCP server in netns A

```
# ip netns exec A \
  sockperf server -i 10.0.0.1 --tcp --daemonize
```

Run TCP client

```
# ip netns exec B \
   sockperf ping-pong -i 10.0.0.1 --tcp --time 30
```



Repeat test with SOCKMAP bypass

Spawn client and server inside the test cgroup

```
# echo $$ > /sys/fs/cgroup/test.slice/cgroup.procs
```

Run TCP server in netns A

```
# ip netns exec A \
  sockperf server -i 10.0.0.1 --tcp --daemonize
```

Run TCP client

```
# ip netns exec B \
   sockperf ping-pong -i 10.0.0.1 --tcp --time 30
sockperf: [Total Run] RunTime=30.000 sec; Warm up time=400 msec; SentMessages=3189584;
ReceivedMessages=3189583
…
sockperf: ====> avg-latency=4.686 (std-dev=2.862, mean-ad=0.250, median-ad=0.216,
siqr=0.173, cv=0.611, std-error=0.002, 99.0% ci=[4.682, 4.690])
sockperf: # dropped messages = 0; # duplicated messages = 0; # out-of-order messages = 0
sockperf: Summary: Latency is 4.686 usec
```

4.7 ± 2.9 µsec

Without and with SOCKMAP bypass

before: 5.8 ± 2.0 µsec

↓ **- 18.5%**

after: 4.7 ± 2.9 µsec

Run the benchmark yourself:
https://github.com/jsitnicki/sockmap-project/tree/main/examples/send-to-local

Section

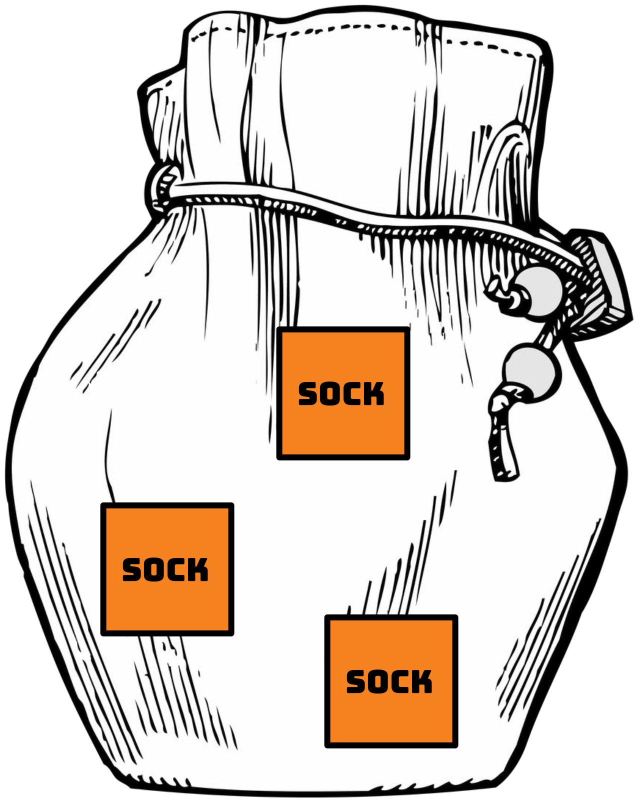
What is SOCKMAP?

What is SOCKMAP?

Two things

What is SOCKMAP?

Collection / container for socket references in Linux kernel



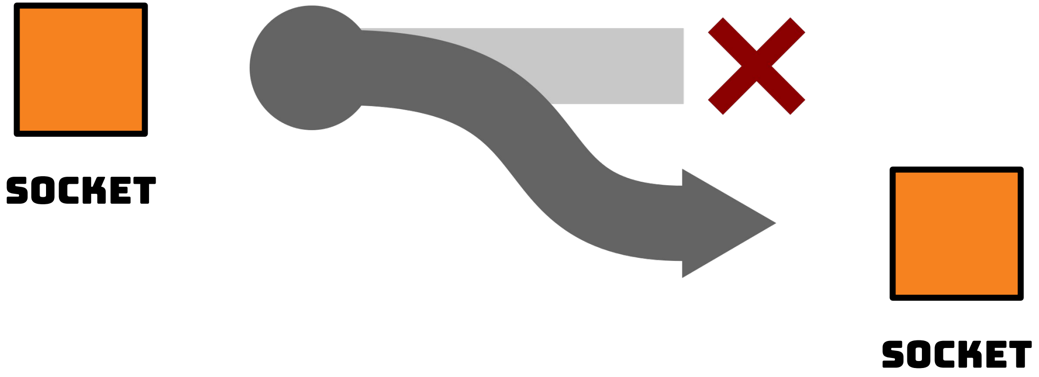
What is SOCKMAP?

SOCKMAP
API

1. **container for sockets**

- BPF map (K/V store)
- holds weak refs to sockets

What is SOCKMAP?



API for enforcing policy
and
redirecting data between sockets

What is SOCKMAP?

SOCKMAP API

1. **container for sockets**

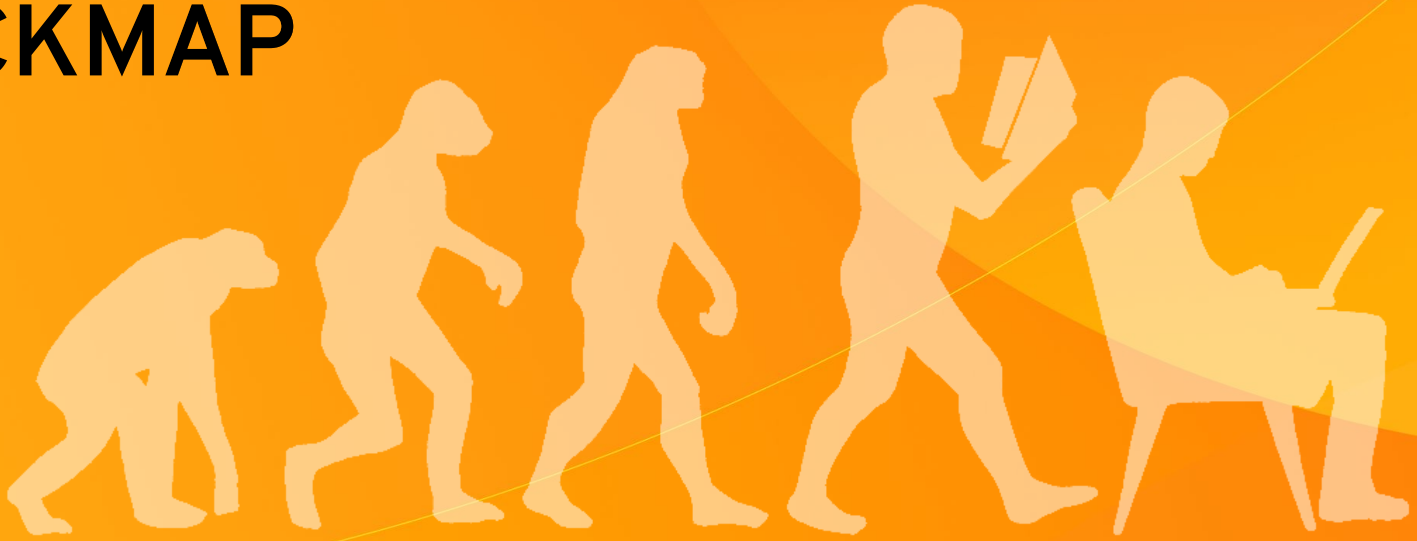
- BPF map (K/V store)
- holds weak refs to sockets

2. **policy enforcement & redirecting packets**

- BPF programs to filter or redirect (steer) data from socket to socket
- hooks into socket layer

Section

EVOLUTION OF SOCKMAP



How did we get here?

Kernel Connection Multiplexor

Added infrastructure later reused by SOCKMAP - Psock and stream parser program.

2016

v4.6

```
+------------+   +------------+   +------------+   +------------+
| KCM socket |   | KCM socket |   | KCM socket |   | KCM socket |
+------------+   +------------+   +------------+   +------------+
      |                 |               |                |
      +-----------+     |               |     +----------+
                  |     |               |     |
               +----------------------------------+
               |           Multiplexor            |
               +----------------------------------+
                 |   |           |           |  |
       +---------+   |           |           |  ------------+
       |             |           |           |              |
+----------+  +----------+  +----------+  +----------+ +----------+
|  Psock   |  |  Psock   |  |  Psock   |  |  Psock   | |  Psock   |
+----------+  +----------+  +----------+  +----------+ +----------+
      |              |           |            |             |
+----------+  +----------+  +----------+  +----------+ +----------+
| TCP sock |  | TCP sock |  | TCP sock |  | TCP sock | | TCP sock |
+----------+  +----------+  +----------+  +----------+ +----------+
```

[PATCH v2 net-next 00/13] kcm: Kernel Connection Multiplexor (KCM)

How did we get here?

Kernel Connection Multiplexor

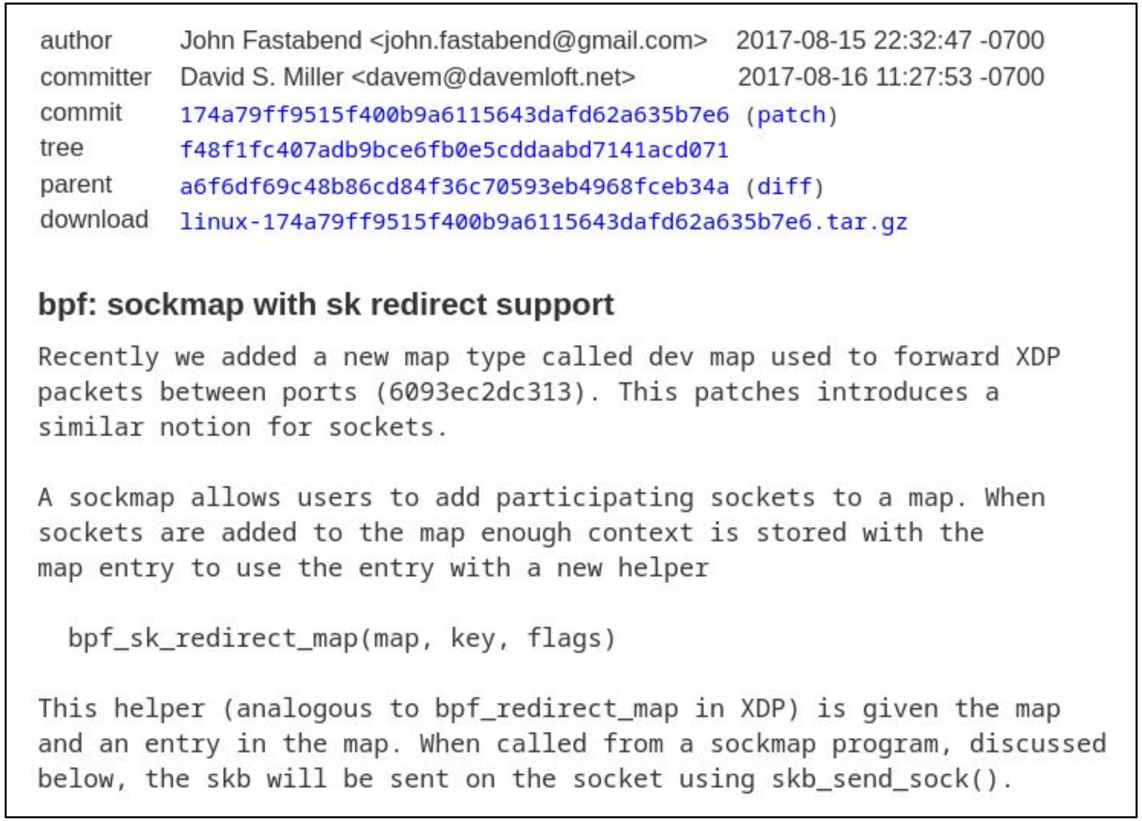
Added infrastructure later reused by SOCKMAP - Psock and stream parser program.

2016	2017
v4.6	v4.14

SOCKMAP initial version

Filtering and redirect on ingress to socket layer.

Counterpart of XDP DEVMAP.



author	John Fastabend <john.fastabend@gmail.com>	2017-08-15 22:32:47 -0700
committer	David S. Miller <davem@davemloft.net>	2017-08-16 11:27:53 -0700
commit	174a79ff9515f400b9a6115643dafd62a635b7e6 (patch)	
tree	f48f1fc407adb9bce6fb0e5cddaabd7141acd071	
parent	a6f6df69c48b86cd84f36c70593eb4968fceb34a (diff)	
download	linux-174a79ff9515f400b9a6115643dafd62a635b7e6.tar.gz	

bpf: sockmap with sk redirect support

```
Recently we added a new map type called dev map used to forward XDP
packets between ports (6093ec2dc313). This patches introduces a
similar notion for sockets.

A sockmap allows users to add participating sockets to a map. When
sockets are added to the map enough context is stored with the
map entry to use the entry with a new helper

  bpf_sk_redirect_map(map, key, flags)

This helper (analogous to bpf_redirect_map in XDP) is given the map
and an entry in the map. When called from a sockmap program, discussed
below, the skb will be sent on the socket using skb_send_sock().
```

commit 174a79ff9515 ("bpf: sockmap with sk redirect support")

How did we get here?

Kernel Connection Multiplexor

Added infrastructure later reused by SOCKMAP - Psock and stream parser program.

SK_MSG program

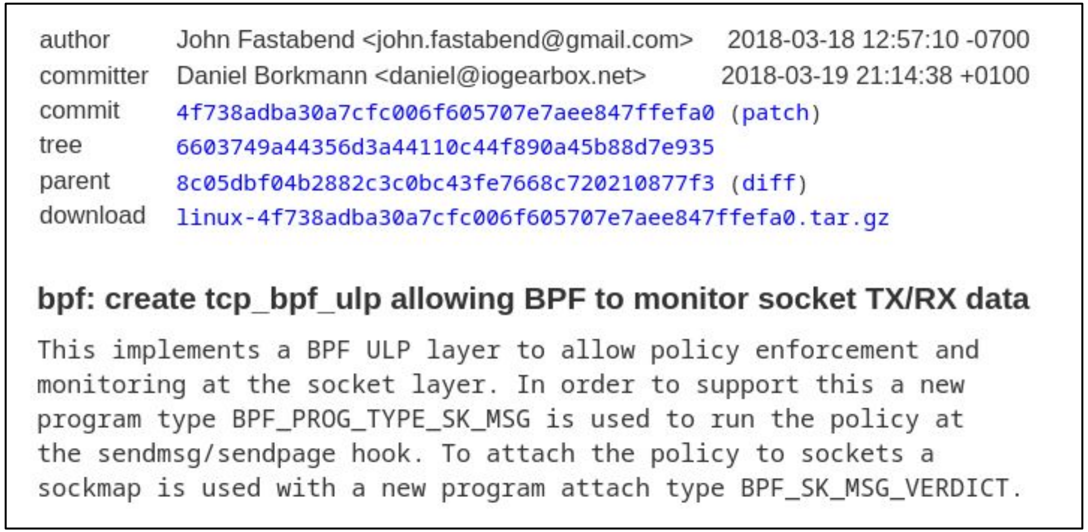
Filter and redirect at sendmsg() time (TCP only).

2016	2017	2018
v4.6	v4.14	v4.17

SOCKMAP initial version

Filter and redirect on ingress to socket layer (TCP only).

Counterpart of XDP DEVMAP.



```
author     John Fastabend <john.fastabend@gmail.com>   2018-03-18 12:57:10 -0700
committer  Daniel Borkmann <daniel@iogearbox.net>      2018-03-19 21:14:38 +0100
commit     4f738adba30a7cfc006f605707e7aee847ffefa0 (patch)
tree       6603749a44356d3a44110c44f890a45b88d7e935
parent     8c05dbf04b2882c3c0bc43fe7668c720210877f3 (diff)
download   linux-4f738adba30a7cfc006f605707e7aee847ffefa0.tar.gz
```

bpf: create tcp_bpf_ulp allowing BPF to monitor socket TX/RX data

```
This implements a BPF ULP layer to allow policy enforcement and
monitoring at the socket layer. In order to support this a new
program type BPF_PROG_TYPE_SK_MSG is used to run the policy at
the sendmsg/sendpage hook. To attach the policy to sockets a
sockmap is used with a new program attach type BPF_SK_MSG_VERDICT.
```

commit 4f738adba30a ("bpf: create tcp_bpf_ulp allowing BPF to monitor socket TX/RX data")

How did we get here?

Kernel Connection Multiplexor

Added infrastructure later reused by SOCKMAP - Psock and stream parser program.

2016
v4.6

2017
v4.14

SOCKMAP initial version

Filter and redirect on ingress to socket layer (TCP only).

Counterpart of XDP DEVMAP.

SK_MSG program

Filter and redirect at sendmsg() time (TCP only).

2018
v4.17

2018
v4.18

SOCKHASH map

Same as SOCKMAP but with flexible lookup key - blob of bytes, e.g. 5 tuple.

Backed by a hash table.

How did we get here?

2016 v4.6

Kernel Connection Multiplexor

Added infrastructure later reused by SOCKMAP - Psock and stream parser program.

2017 v4.14

SOCKMAP initial version

Filter and redirect on ingress to socket layer (TCP only).

Counterpart of XDP DEVMAP.

2018 v4.17

SK_MSG program

Filter and redirect at sendmsg() time (TCP only).

2018 v4.18

SOCKHASH map

Same as SOCKMAP but with flexible lookup key - blob of bytes, e.g. 5 tuple.

Backed by a hash table.

2018 v4.20

Switch to sk_msg API

Kernel TLS and SK_MSG can gracefully coexist.

Allows introspection / policy enforcement before in-kernel encryption.

commit 4f738adba30a ("bpf: create tcp_bpf_ulp allowing BPF to monitor socket TX/RX data")

How did we get here?

Store TCP or UDP sockets

SOCKMAP becomes a generic BPF map for sockets.

It can hold both connected and listening TCP sockets, and any bound UDP socket.

2020

v5.7

How did we get here?

Store TCP or UDP sockets

SOCKMAP becomes a generic BPF map for sockets.

It can hold both connected and listening TCP sockets, and any bound UDP socket.

2020	2020
v5.7	v5.10

BPF iterators support

Iterate over SOCKMAP from BPF context.

Allows copying socket references from one SOCKMAP to another.

How did we get here?

Store TCP or UDP sockets

SOCKMAP becomes a generic BPF map for sockets.

It can hold both connected and listening TCP sockets, and any bound UDP socket.

2020
v5.7

2020
v5.10

BPF iterators support

Iterate over SOCKMAP from BPF context.

Allows copying socket references from one SOCKMAP to another.

2021
v5.13

Redirect for UDP

Use new SK_SKB_VERDICT BPF program to redirect packets between UDP sockets.

How did we get here?

2020 v5.7

Store TCP or UDP sockets

SOCKMAP becomes a generic BPF map for sockets.

It can hold both connected and listening TCP sockets, and any bound UDP socket.

2020 v5.10

BPF iterators support

Iterate over SOCKMAP from BPF context.

Allows copying socket references from one SOCKMAP to another.

2021 v5.13

Redirect for UDP

Use new SK_SKB_VERDICT BPF program to redirect packets between UDP sockets.

2021 v5.15

UNIX domain sockets

Allow storing and redirecting packets from/to UNIX sockets (dgram and stream).

How did we get here?

VSOCK domain sockets

Redirecting from / to VSOCK sockets – stream and seqpacket.

2023

v6.4

How did we get here?

VSOCK domain sockets

Redirecting from / to
VSOCK sockets –
stream and seqpacket.

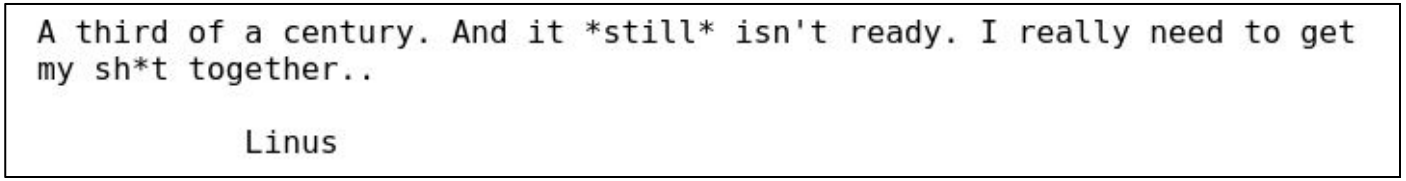
2023
v6.4

202x
v6.x

Your contribution here

The code continues to evolve...

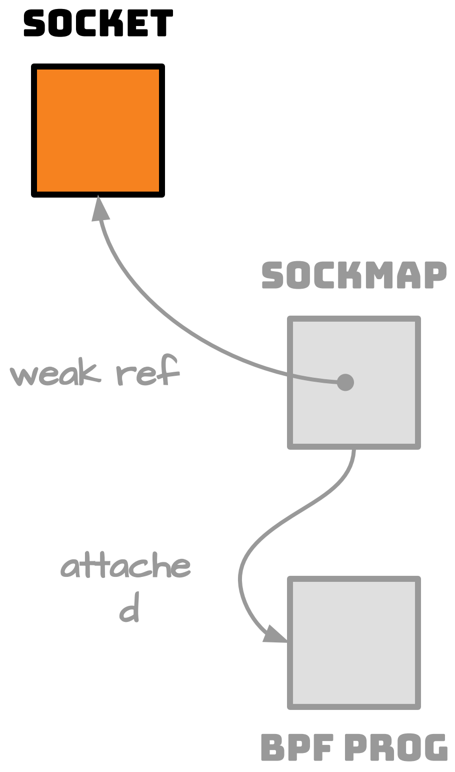
How did we get here?


```
A third of a century. And it *still* isn't ready. I really need to get
my sh*t together..

              Linus
```

https://lore.kernel.org/all/CAHk-=whsqTTsiZ=XmecYwQqqya2C4ufysiDj2bOPhvke4mR2mg@mail.gmail.com/

How to set up SOCKMAP?

① Open a connected (established) socket



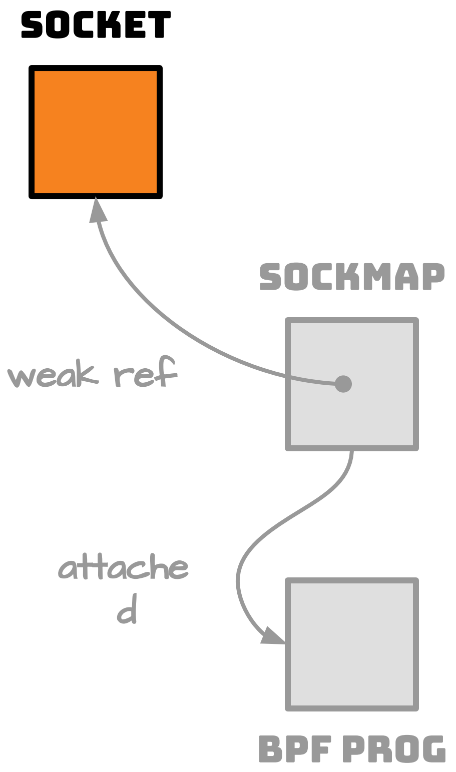
active open

```
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 8
connect(8, {sa_family=AF_INET,
            sin_port=htons(41895),
            sin_addr=inet_addr("127.0.0.1")}, 16) = 0
```

passive open

```
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 7
bind(7, {sa_family=AF_INET,
         sin_port=htons(41895),
         sin_addr=inet_addr("127.0.0.1")}, 16) = 0
listen(7, 4096)                             = 0
accept(7, NULL, NULL)                       = 9
```

What sockets can you use?

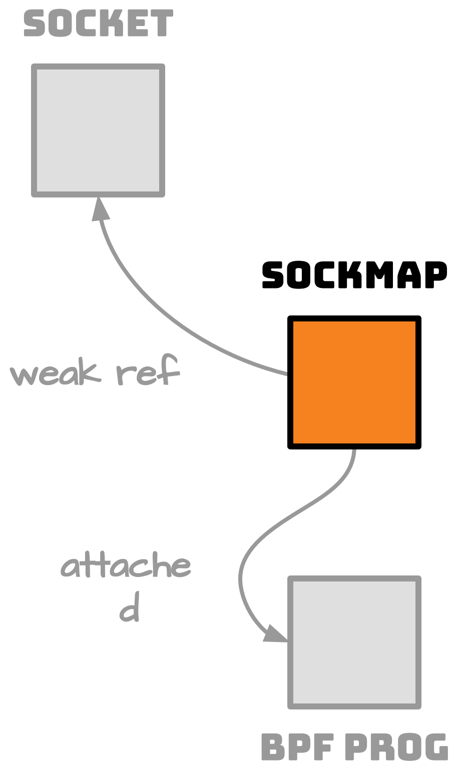


connected (established) socket:

- ❏ **TCP**
- ❏ **UDP**
- ❏ **UNIX** (STREAM, DGRAM)
- ❏ **VSOCK** (STREAM, SEQPACKET)

`man 7 {tcp,udp,unix,vsock}`

② Create a BPF map - SOCKMAP or SOCKHASH

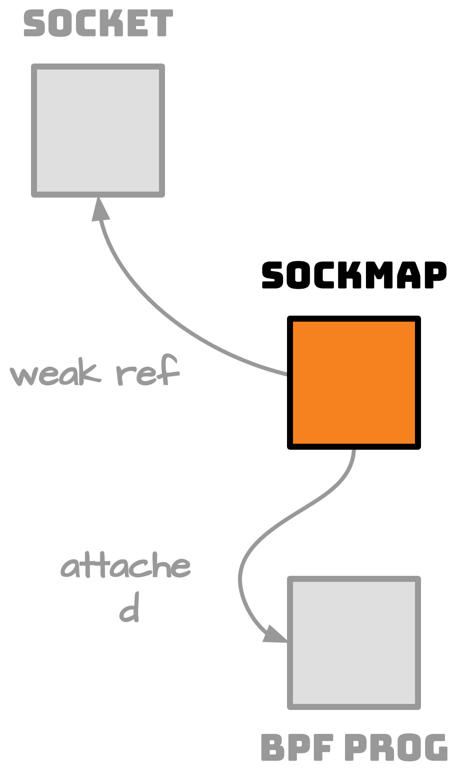


use the `bpf()` syscall

or a library wrapper (`ebpf-go`, `libbpf`)

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_SOCKMAP,
                     key_size=4,
                     value_size=8,
                     max_entries=1,
                     map_flags=0,
                     …}, 72) = 5
```

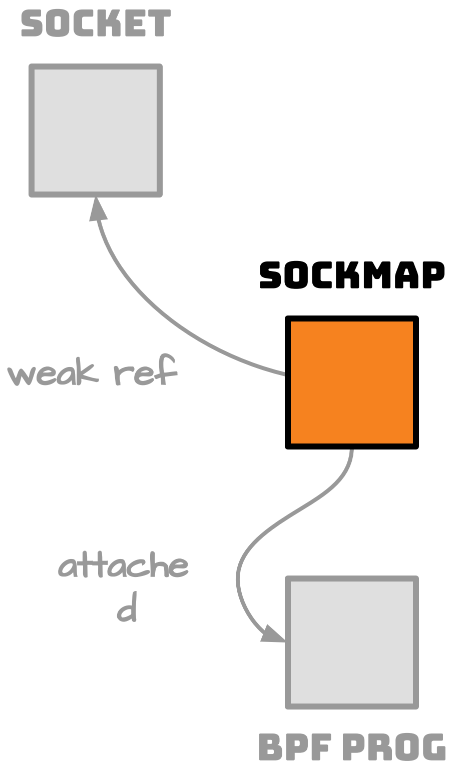
② Create a BPF map



or use `bpftool map create` command

```
bpftool map create                                          \
  /sys/fs/bpf/sockmap `# path on bpffs`                     \
  type sockmap        `# sockmap or sockhash`               \
  key 4               `# always 4 bytes for sockmap`        \
  value 8             `# use 8 bytes for dump to work`      \
  entries 1                                                 \
  name sockmap

bpftool map show pinned /sys/fs/bpf/sockmap
3: sockmap  name sockmap  flags 0x0
        key 4B  value 8B  max_entries 1  memlock 328B
```

What BPF maps can you use?



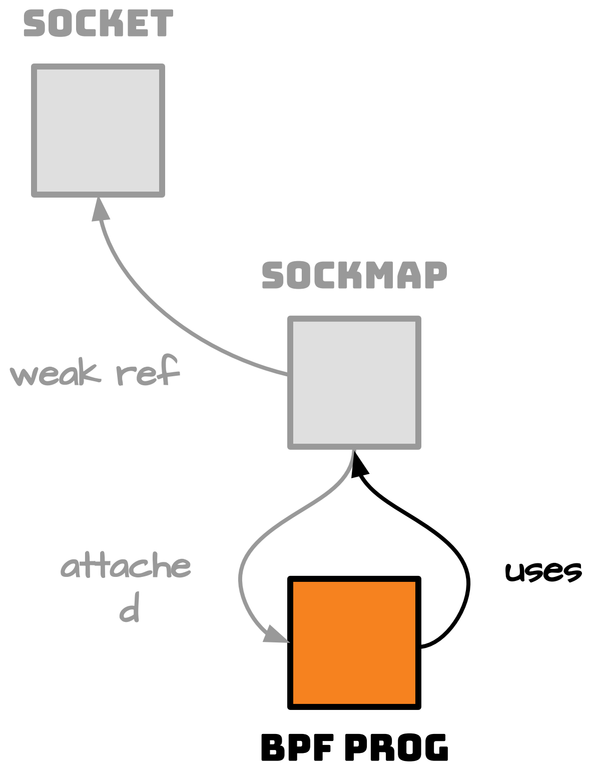
Map types:

- ❏ BPF_MAP_TYPE_**SOCKMAP**
  - ❏ **32-bit integer key**
- ❏ BPF_MAP_TYPE_**SOCKHASH**
  - ❏ **binary blob key**

Not to be confused with
BPF_MAP_TYPE_**REUSEPORT_SOCKARRAY**

https://docs.kernel.org/bpf/map_sockmap.html

③ Load a BPF program

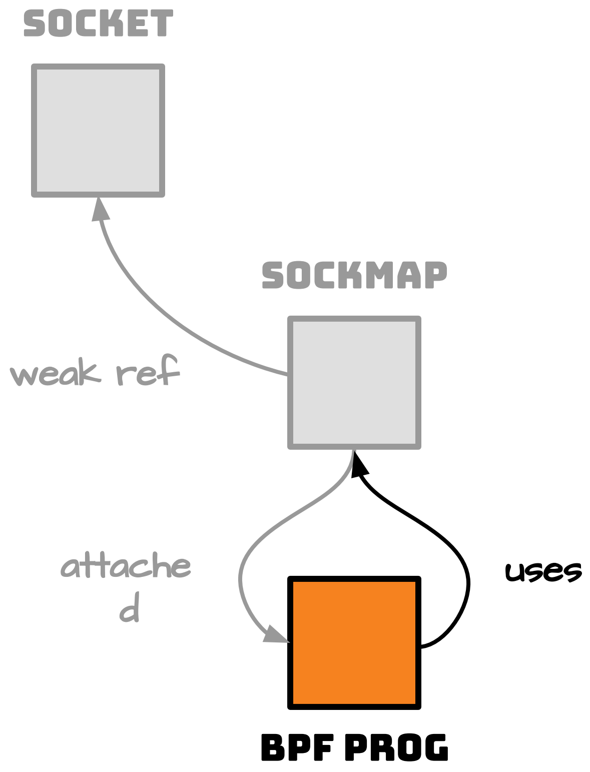


```
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_SK_MSG,
                    insn_cnt=6,
                    insns=0xcf2a70,
                    expected_attach_type=BPF_SK_MSG_VERDICT,
                    …}, 128) = 6
```

Program types:

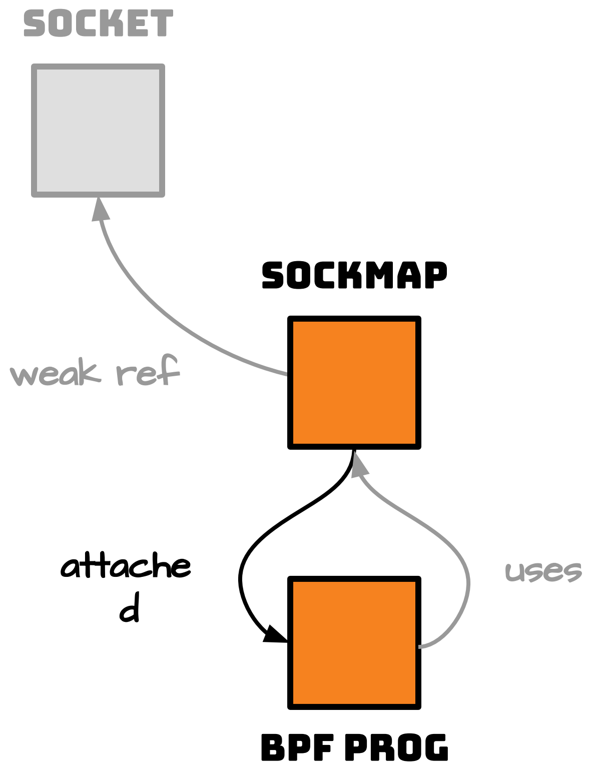
- ❏ BPF_PROG_TYPE_**SK_MSG**
- ❏ BPF_PROG_TYPE_**SK_SKB**

③ Load a BPF program - it uses SOCKMAP



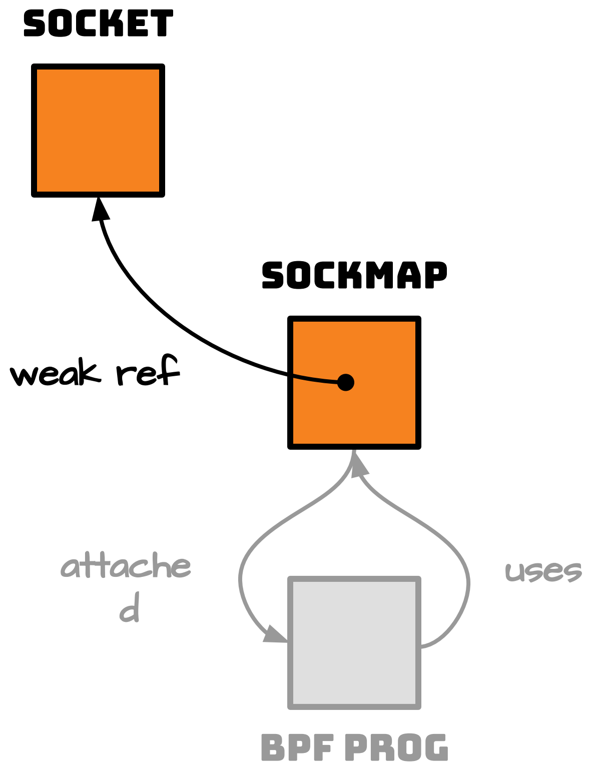
```
# bpftool prog dump xlated id 42
int prog_msg_redir_ingress(struct sk_msg_md * msg):
   0: (18) r2 = map[id:17]
   …
   5: (95) exit
# bpftool map show id 17
17: sockmap  name output  flags 0x0
        key 4B  value 8B  max_entries 1  memlock 328B
        pids sockmap-redir-m(331)
```

④ Attach BPF program to SOCKMAP



```
bpf(BPF_PROG_ATTACH, {target_fd=5,
                      attach_bpf_fd=6,
                      attach_type=BPF_SK_MSG_VERDICT,
                      attach_flags=0,
                      replace_bpf_fd=0}, 20) = 0
```

⑤ Insert socket into SOCKMAP



```
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=5,
                         key=0x7ffeb3803870,
                         value=0x7ffeb3803868,
                         flags=BPF_NOEXIST}, 32) = 0
```

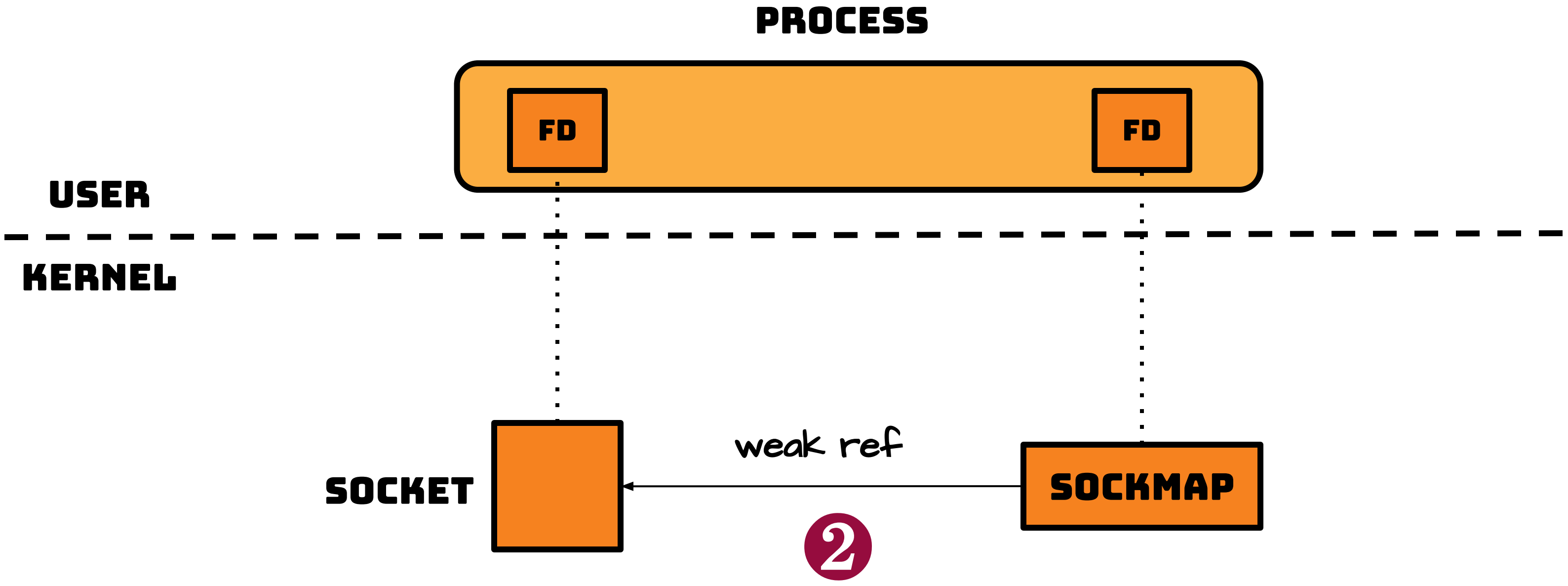
⚠ must be done after attaching the program

Section

CLOUDFLARE

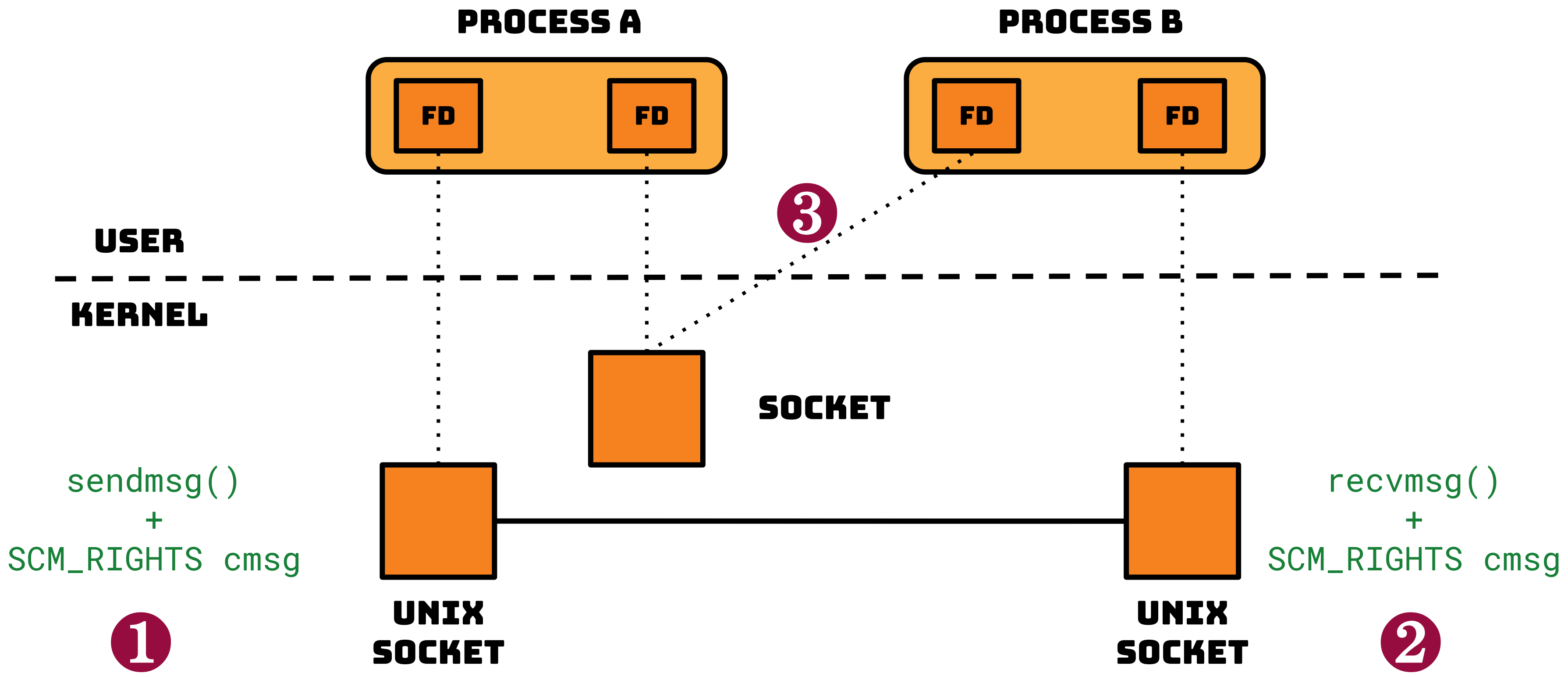
How to get sockets into a SOCKMAP?

Easy case - Single process



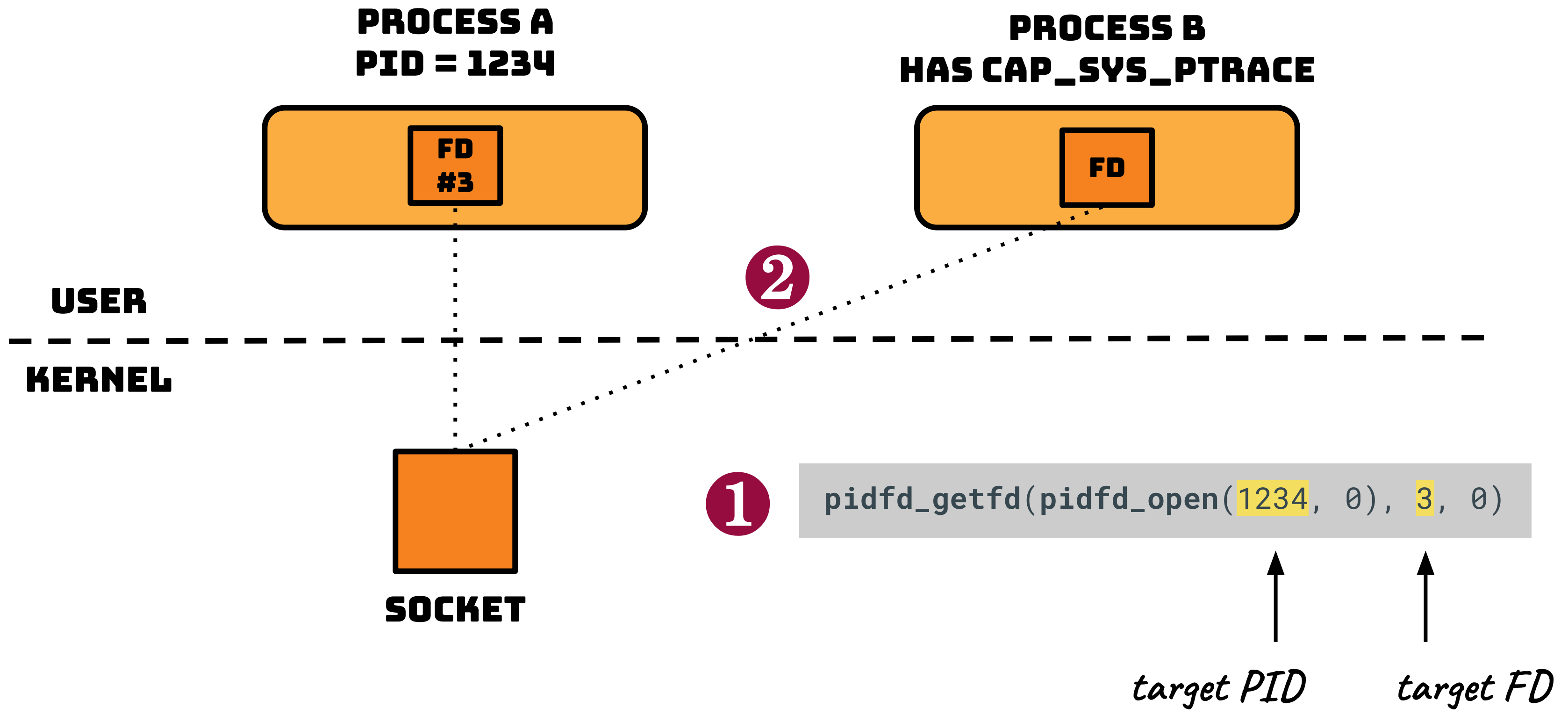
❶ `bpf(BPF_MAP_UPDATE_ELEM, …)`

Socket FD handover with SCM_RIGHTS



https://manpages.debian.org/unstable/manpages/unix.7.en.html#SCM_RIGHTS
https://blog.cloudflare.com/know-your-scm_rights/

“Steal” a socket FD

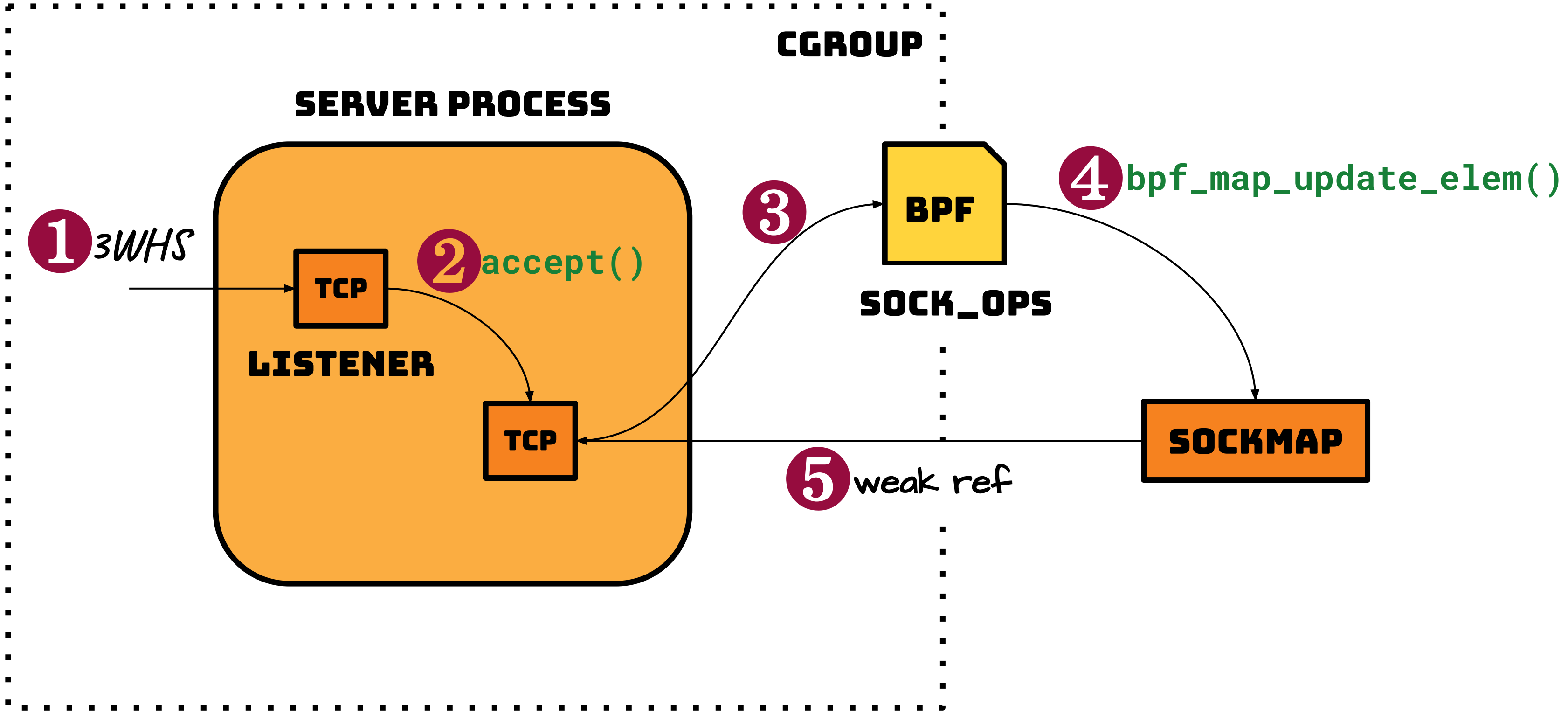


```
pidfd_getfd(pidfd_open(1234, 0), 3, 0)
```

Linux 5.6+, requires CAP_SYS_PTRACE

https://manpages.debian.org/unstable/manpages-dev/pidfd_getfd.2.en.html

BPF sock_ops program attached to cgroup (TCP only)

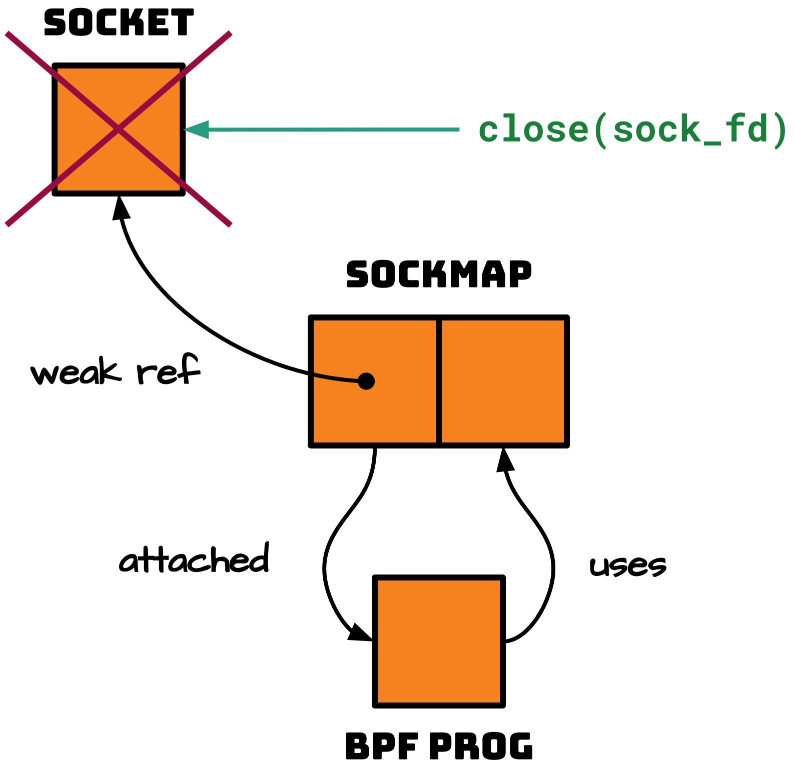


https://github.com/jsitnicki/sockmap-project/blob/main/examples/send-to-local/redir_bypass.bpf.c

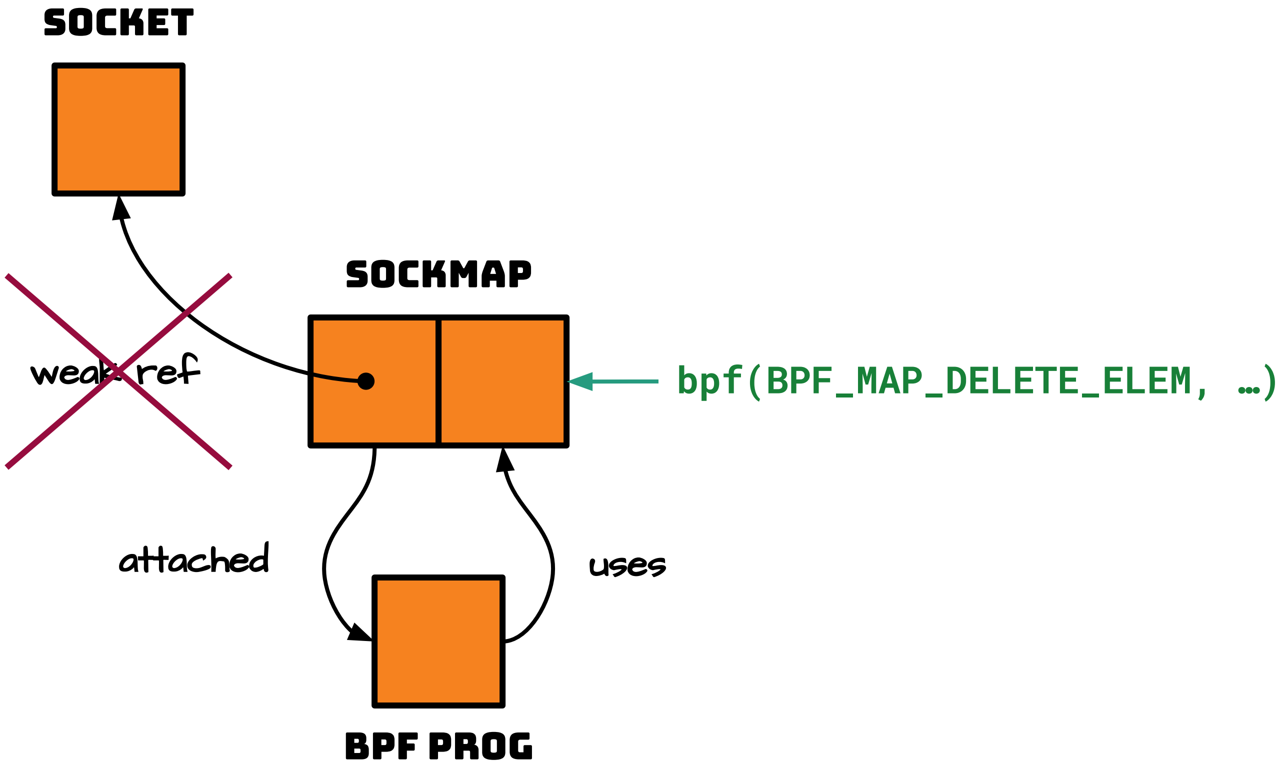
How to tear it down?

pxhere.com, CC0

(A) destroy the socket

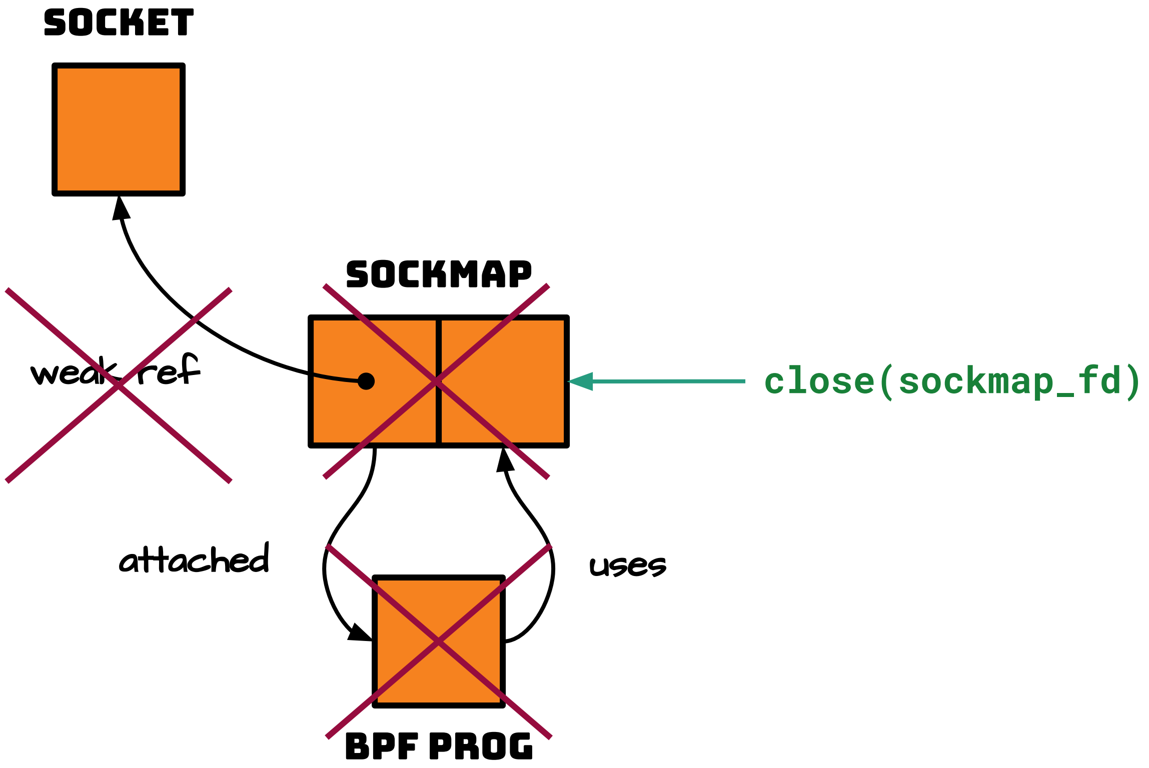


(B) remove socket from sockmap

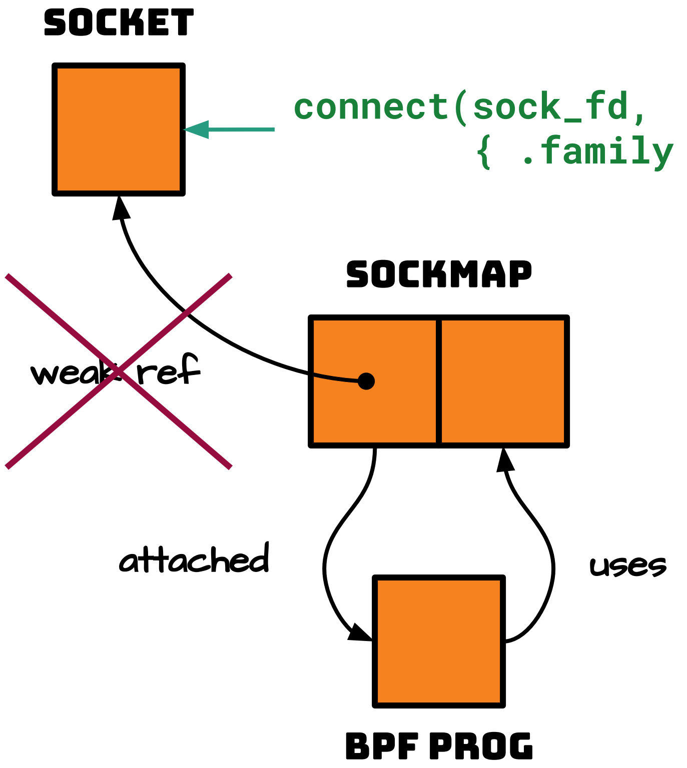


`bpf(BPF_MAP_DELETE_ELEM, …)`

(C) destroy the sockmap



(D) disconnect the socket (rare)

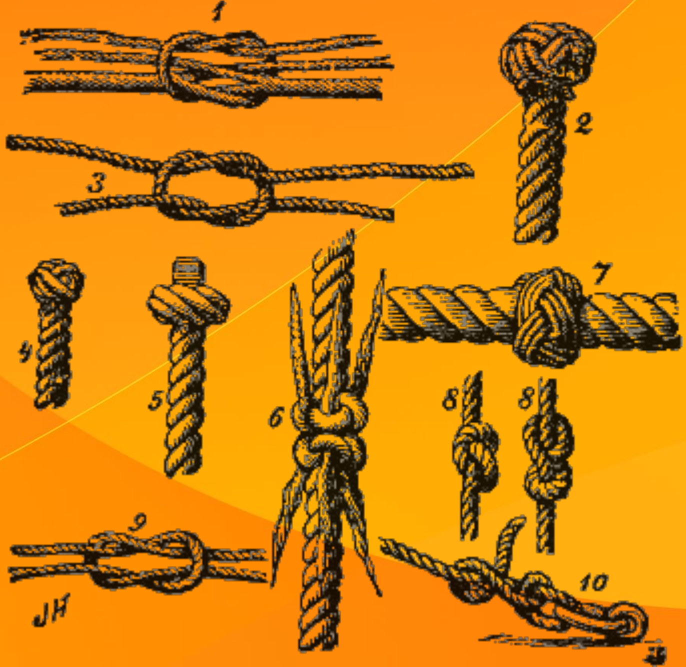


```
connect(sock_fd,
          { .family = AF_UNSPEC }, …)
```

See `connect(2)` man page

Section

Supported Socket Splicing Setups

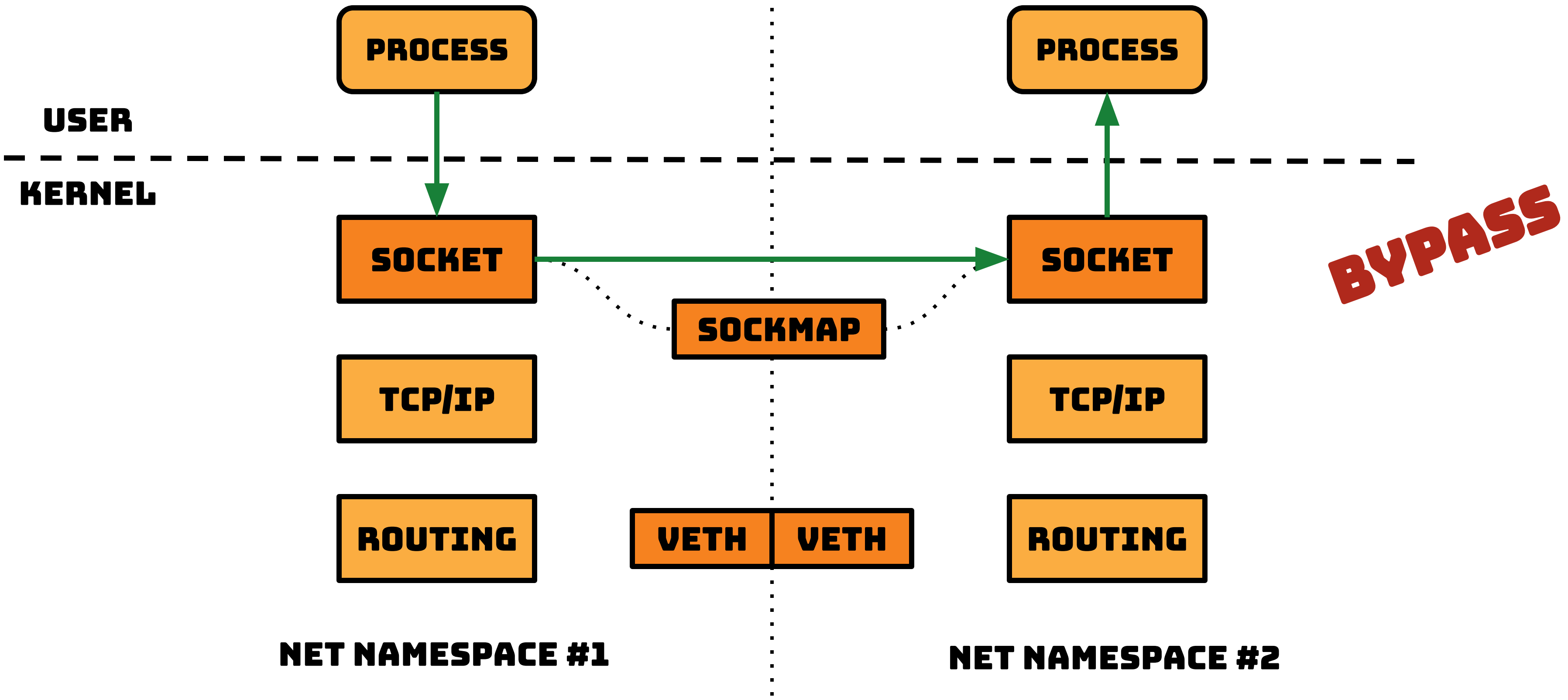


Redirect

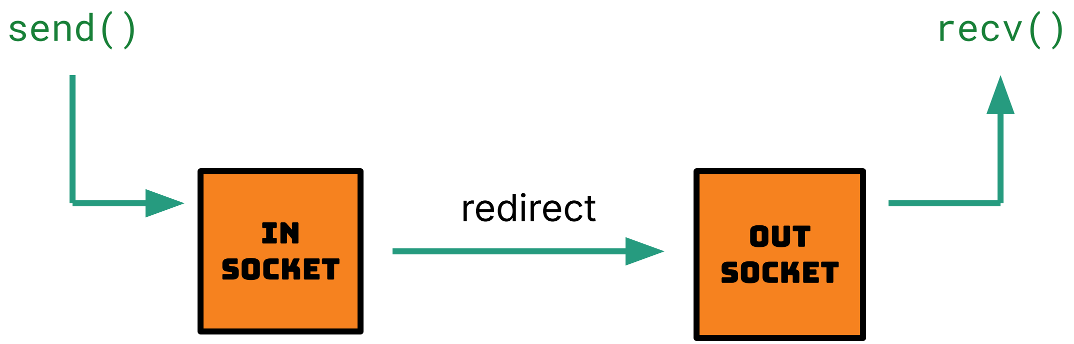
Redirect

send to local

Redirect use case → Bypass for containers

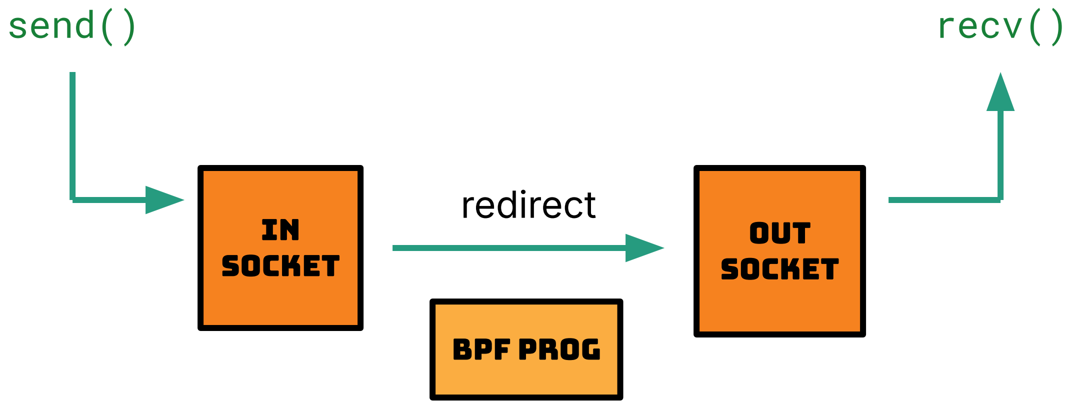


Redirect → send to local



Like `socketpair()` or `pipe()`

Redirect → send to local → How?



BPF_PROG_TYPE_**SK_MSG** program

→ attached to BPF_**SK_MSG_VERDICT** hook

→ calls bpf_**msg**_redirect_hash/map() with BPF_F_**INGRESS** flag

→ returns SK_**PASS**

selects target socket

Redirect → send to local → Example

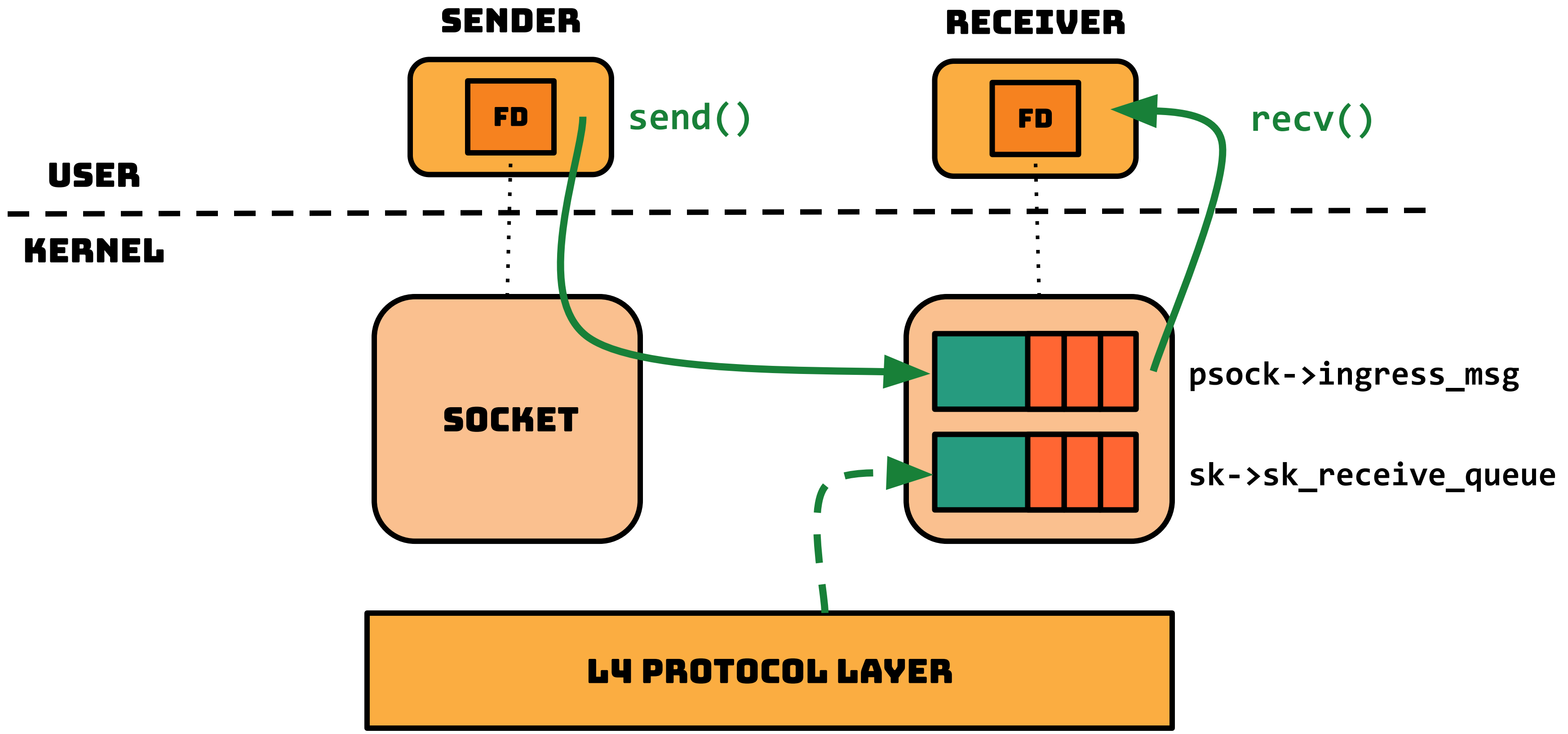
```
SEC("sk_msg")
int sk_msg_redir_ingress(struct sk_msg_md *msg)
{
    __u32 key = 0;

    if (msg->remote_port == bpf_htonl(53))
        key = 1;

     return bpf_msg_redirect_map(msg, &sockmap, key, BPF_F_INGRESS);
}
```

Redirect → send to local → What?

IN → OUT	TCP	UDP	UNIX STR	UNIX DGR	VSOCK STR	VSOCK SEQ
TCP	🟢	🟢	🟢	🟢	●	●
UDP	●	●	●	●	●	●
UNIX STR	●	●	●	●	●	●
UNIX DGR	●	●	●	●	●	●
VSOCK STR	●	●	●	●	●	●
VSOCK SEQ	●	●	●	●	●	●

TCP to any but VSOCK

Redirect → send to local → Internals



Let's trace what happens in the sender process during send-to-local redirect

```
~ # echo 1 > /sys/kernel/tracing/options/funcgraph-retval
~ # perf ftrace -C 0 -G __sys_sendto --graph-opts noirqs
```

```
sh # perf ftrace -C 0 -G __sys_sendto --graph-opts noirqs
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
  0)               |  __sys_sendto() {
  0)               |    inet_sendmsg() {
  0)               |      tcp_bpf_sendmsg() {
  0) + 23.782 us   |        sk_msg_alloc();
  0) + 29.818 us   |        sk_msg_memcopy_from_iter();
  0)               |        sk_psock_msg_verdict() {
  0)               |          bpf_msg_redirect_hash() {
  0)   1.764 us    |            __sock_hash_lookup_elem();
  0)   2.599 us    |          } /* bpf_msg_redirect_hash = 0x1 */
  0)   6.048 us    |        } /* sk_psock_msg_verdict = 0x2 */
  0)   1.204 us    |        sk_msg_return(); /* = 0x1 */
  0)               |        tcp_bpf_sendmsg_redir() {
  0)   1.507 us    |          __sk_mem_schedule(); /* = 0x1 */
  0)               |          sock_def_readable() {
  0)               |            __wake_up_sync_key() {
  0)               |              __wake_up_common() {
  0)               |                pollwake() {
  0)               |                  default_wake_function() {
  0) + 12.872 us   |                    try_to_wake_up(); /* = 0x1 */
  0) + 13.207 us   |                  } /* default_wake_function = 0x1 */
  0) + 13.650 us   |                } /* pollwake = 0x1 */
  0) + 14.583 us   |              } /* __wake_up_common = 0x1 */
  0) + 16.905 us   |            } /* __wake_up_sync_key = 0x4b18 */
  0) + 19.696 us   |          } /* sock_def_readable = 0x0 */
  0) + 63.929 us   |        } /* tcp_bpf_sendmsg_redir = 0x0 */
  0) ! 168.534 us  |      } /* tcp_bpf_sendmsg = 0x7 */
  0) ! 172.112 us  |    } /* inet_sendmsg = 0x7 */
  0) ! 194.533 us  |  } /* __sys_sendto = 0x7 */
```

irrelevant bits omitted

```
sh # perf ftrace -C 0 -G __sys_sendto --graph-opts noirqs
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
  0)               |  __sys_sendto() {
  0)               |    inet_sendmsg() {
  0)               |      tcp_bpf_sendmsg() {
  0) + 23.782 us   |        sk_msg_alloc();
  0) + 29.818 us   |        sk_msg_memcopy_from_iter();
  0)               |        sk_psock_msg_verdict() {
  0)               |          bpf_msg_redirect_hash() {
  0)   1.764 us    |            __sock_hash_lookup_elem();
  0)   2.599 us    |          } /* bpf_msg_redirect_hash = 0x1 */
  0)   6.048 us    |        } /* sk_psock_msg_verdict = 0x2 */
  0)   1.204 us    |        sk_msg_return(); /* = 0x1 */
  0)               |        tcp_bpf_sendmsg_redir() {
  0)   1.507 us    |          __sk_mem_schedule(); /* = 0x1 */
  0)               |          sock_def_readable() {
  0)               |            __wake_up_sync_key() {
  0)               |              __wake_up_common() {
  0)               |                pollwake() {
  0)               |                  default_wake_function() {
  0) + 12.872 us   |                    try_to_wake_up(); /* = 0x1 */
  0) + 13.207 us   |                  } /* default_wake_function = 0x1 */
  0) + 13.650 us   |                } /* pollwake = 0x1 */
  0) + 14.583 us   |              } /* __wake_up_common = 0x1 */
  0) + 16.905 us   |            } /* __wake_up_sync_key = 0x4b18 */
  0) + 19.696 us   |          } /* sock_def_readable = 0x0 */
  0) + 63.929 us   |        } /* tcp_bpf_sendmsg_redir = 0x0 */
  0) ! 168.534 us  |      } /* tcp_bpf_sendmsg = 0x7 */
  0) ! 172.112 us  |    } /* inet_sendmsg = 0x7 */
  0) ! 194.533 us  |  } /* __sys_sendto = 0x7 */
```

Entry to syscall, allocate & initialize the message

```
sh # perf ftrace -C 0 -G __sys_sendto --graph-opts noirqs
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
  0)               |  __sys_sendto() {
  0)               |    inet_sendmsg() {
  0)               |      tcp_bpf_sendmsg() {
  0) + 23.782 us   |        sk_msg_alloc();
  0) + 29.818 us   |        sk_msg_memcopy_from_iter();
  0)               |        sk_psock_msg_verdict() {
  0)               |          bpf_msg_redirect_hash() {
  0)   1.764 us    |            __sock_hash_lookup_elem();
  0)   2.599 us    |          } /* bpf_msg_redirect_hash = 0x1 */
  0)   6.048 us    |        } /* sk_psock_msg_verdict = 0x2 */
  0)   1.204 us    |        sk_msg_return(); /* = 0x1 */
  0)               |        tcp_bpf_sendmsg_redir() {
  0)   1.507 us    |          __sk_mem_schedule(); /* = 0x1 */
  0)               |          sock_def_readable() {
  0)               |            __wake_up_sync_key() {
  0)               |              __wake_up_common() {
  0)               |                pollwake() {
  0)               |                  default_wake_function() {
  0) + 12.872 us   |                    try_to_wake_up(); /* = 0x1 */
  0) + 13.207 us   |                  } /* default_wake_function = 0x1 */
  0) + 13.650 us   |                } /* pollwake = 0x1 */
  0) + 14.583 us   |              } /* __wake_up_common = 0x1 */
  0) + 16.905 us   |            } /* __wake_up_sync_key = 0x4b18 */
  0) + 19.696 us   |          } /* sock_def_readable = 0x0 */
  0) + 63.929 us   |        } /* tcp_bpf_sendmsg_redir = 0x0 */
  0) ! 168.534 us  |      } /* tcp_bpf_sendmsg = 0x7 */
  0) ! 172.112 us  |    } /* inet_sendmsg = 0x7 */
  0) ! 194.533 us  |  } /* __sys_sendto = 0x7 */
```

Run BPF program, find the destination socket in the BPF map

```
sh # perf ftrace -C 0 -G __sys_sendto --graph-opts noirqs
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
  0)                   |  __sys_sendto() {
  0)                   |    inet_sendmsg() {
  0)                   |      tcp_bpf_sendmsg() {
  0) + 23.782 us       |        sk_msg_alloc();
  0) + 29.818 us       |        sk_msg_memcopy_from_iter();
  0)                   |        sk_psock_msg_verdict() {
  0)                   |          bpf_msg_redirect_hash() {
  0)   1.764 us        |            __sock_hash_lookup_elem();
  0)   2.599 us        |          } /* bpf_msg_redirect_hash = 0x1 */
  0)   6.048 us        |        } /* sk_psock_msg_verdict = 0x2 */
  0)   1.204 us        |        sk_msg_return(); /* = 0x1 */
  0)                   |        tcp_bpf_sendmsg_redir() {
  0)   1.507 us        |          __sk_mem_schedule(); /* = 0x1 */
  0)                   |          sock_def_readable() {
  0)                   |            __wake_up_sync_key() {
  0)                   |              __wake_up_common() {
  0)                   |                pollwake() {
  0)                   |                  default_wake_function() {
  0) + 12.872 us       |                    try_to_wake_up(); /* = 0x1 */
  0) + 13.207 us       |                  } /* default_wake_function = 0x1 */
  0) + 13.650 us       |                } /* pollwake = 0x1 */
  0) + 14.583 us       |              } /* __wake_up_common = 0x1 */
  0) + 16.905 us       |            } /* __wake_up_sync_key = 0x4b18 */
  0) + 19.696 us       |          } /* sock_def_readable = 0x0 */
  0) + 63.929 us       |        } /* tcp_bpf_sendmsg_redir = 0x0 */
  0) ! 168.534 us      |      } /* tcp_bpf_sendmsg = 0x7 */
  0) ! 172.112 us      |    } /* inet_sendmsg = 0x7 */
  0) ! 194.533 us      |  } /* __sys_sendto = 0x7 */
```

Deliver message to receiver’s queue, and notify the receiver

```
sh # perf ftrace -C 0 -G __sys_sendto --graph-opts noirqs
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
  0)               |  __sys_sendto() {
  0)               |    inet_sendmsg() {
  0)               |      tcp_bpf_sendmsg() {
  0) + 23.782 us   |        sk_msg_alloc();
  0) + 29.818 us   |        sk_msg_memcopy_from_iter();
  0)               |        sk_psock_msg_verdict() {
  0)               |          bpf_msg_redirect_hash() {
  0)   1.764 us    |            __sock_hash_lookup_elem();
  0)   2.599 us    |          } /* bpf_msg_redirect_hash = 0x1 */
  0)   6.048 us    |        } /* sk_psock_msg_verdict = 0x2 */
  0)   1.204 us    |        sk_msg_return(); /* = 0x1 */
  0)               |        tcp_bpf_sendmsg_redir() {
  0)   1.507 us    |          __sk_mem_schedule(); /* = 0x1 */
  0)               |          sock_def_readable() {
  0)               |            __wake_up_sync_key() {
  0)               |              __wake_up_common() {
  0)               |                pollwake() {
  0)               |                  default_wake_function() {
  0) + 12.872 us   |                    try_to_wake_up(); /* = 0x1 */
  0) + 13.207 us   |                  } /* default_wake_function = 0x1 */
  0) + 13.650 us   |                } /* pollwake = 0x1 */
  0) + 14.583 us   |              } /* __wake_up_common = 0x1 */
  0) + 16.905 us   |            } /* __wake_up_sync_key = 0x4b18 */
  0) + 19.696 us   |          } /* sock_def_readable = 0x0 */
  0) + 63.929 us   |        } /* tcp_bpf_sendmsg_redir = 0x0 */
  0) ! 168.534 us  |      } /* tcp_bpf_sendmsg = 0x7 */
  0) ! 172.112 us  |    } /* inet_sendmsg = 0x7 */
  0) ! 194.533 us  |  } /* __sys_sendto = 0x7 */
```

Wake up the receiver

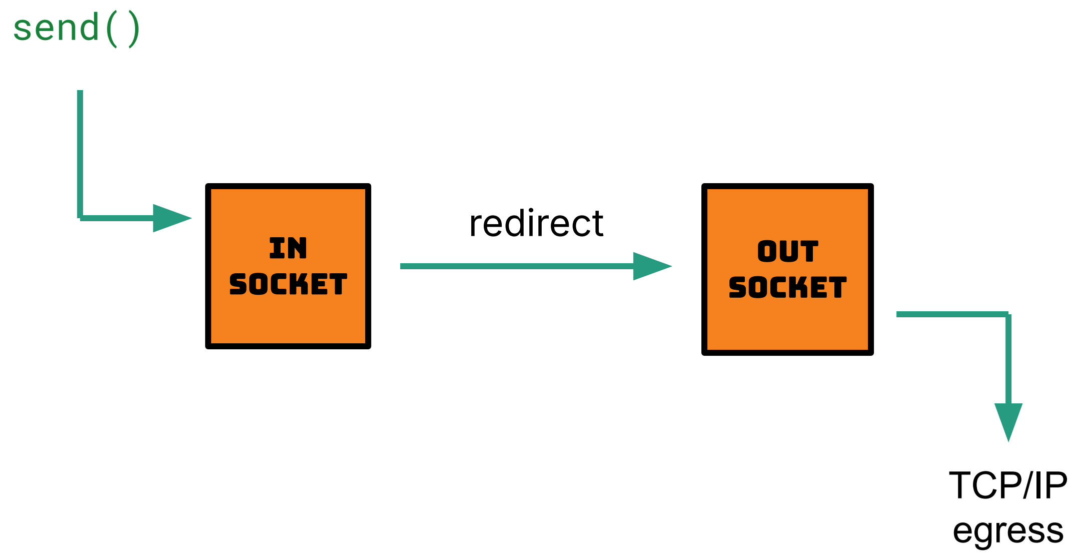
```
sh # perf ftrace -C 0 -G __sys_sendto --graph-opts noirqs
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
  0)               |  __sys_sendto() {
  0)               |    inet_sendmsg() {
  0)               |      tcp_bpf_sendmsg() {
  0) + 23.782 us   |        sk_msg_alloc();
  0) + 29.818 us   |        sk_msg_memcopy_from_iter();
  0)               |        sk_psock_msg_verdict() {
  0)               |          bpf_msg_redirect_hash() {
  0)   1.764 us    |            __sock_hash_lookup_elem();
  0)   2.599 us    |          } /* bpf_msg_redirect_hash = 0x1 */
  0)   6.048 us    |        } /* sk_psock_msg_verdict = 0x2 */
  0)   1.204 us    |        sk_msg_return(); /* = 0x1 */
  0)               |        tcp_bpf_sendmsg_redir() {
  0)   1.507 us    |          __sk_mem_schedule(); /* = 0x1 */
  0)               |          sock_def_readable() {
  0)               |            __wake_up_sync_key() {
  0)               |              __wake_up_common() {
  0)               |                pollwake() {
  0)               |                  default_wake_function() {
  0) + 12.872 us   |                    try_to_wake_up(); /* = 0x1 */
  0) + 13.207 us   |                  } /* default_wake_function = 0x1 */
  0) + 13.650 us   |                } /* pollwake = 0x1 */
  0) + 14.583 us   |              } /* __wake_up_common = 0x1 */
  0) + 16.905 us   |            } /* __wake_up_sync_key = 0x4b18 */
  0) + 19.696 us   |          } /* sock_def_readable = 0x0 */
  0) + 63.929 us   |        } /* tcp_bpf_sendmsg_redir = 0x0 */
  0) ! 168.534 us  |      } /* tcp_bpf_sendmsg = 0x7 */
  0) ! 172.112 us  |    } /* inet_sendmsg = 0x7 */
  0) ! 194.533 us  |  } /* __sys_sendto = 0x7 */
```

Sent message is delivered straight to the receiver's queue!

Redirect

send to egress

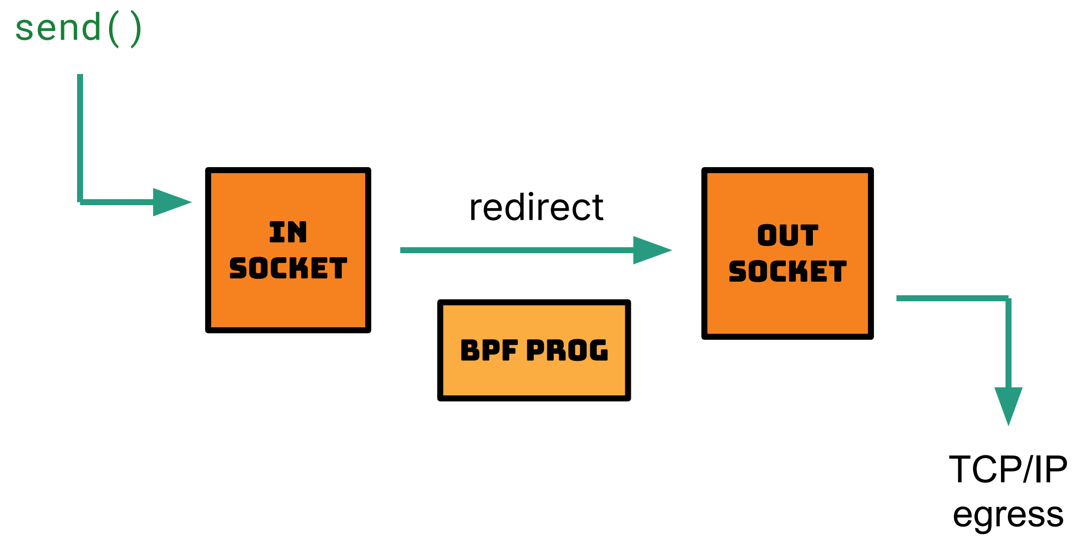
Redirect → send to egress



Sort of like `splice()`

(pipe → socket)

Redirect → send to egress → How?



`BPF_PROG_TYPE_`**`SK_MSG`** prog

→ attached to `BPF_`**`SK_MSG_VERDICT`** hook

→ calls `bpf_`**`msg`**`_redirect_hash/map()` without any flags

→ returns `SK_`**`PASS`**

Redirect → send to egress → What?

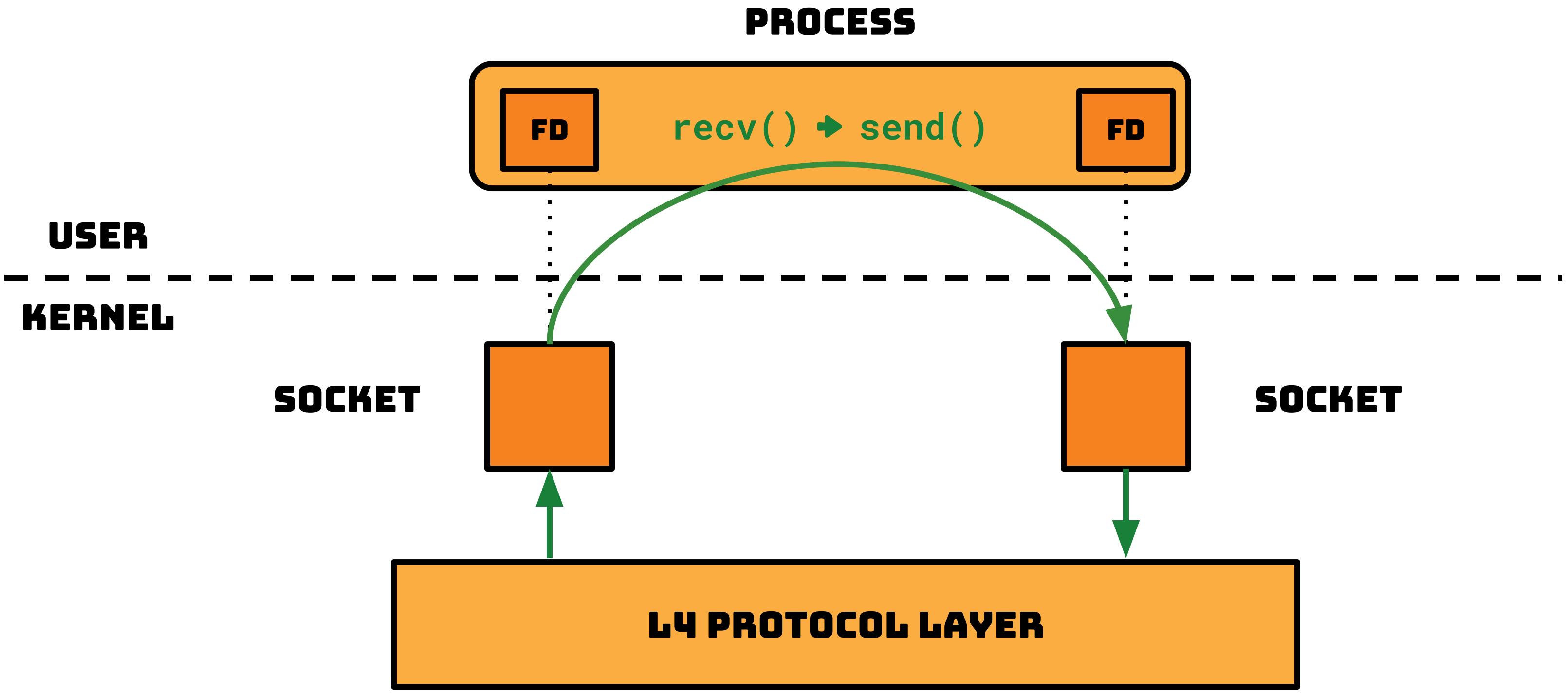
IN → OUT	TCP	UDP	UNIX STR	UNIX DGR	VSOCK STR	VSOCK SEQ
TCP	🟢	●	●	●	●	●
UDP	●	●	●	●	●	●
UNIX STR	●	●	●	●	●	●
UNIX DGR	●	●	●	●	●	●
VSOCK STR	●	●	●	●	●	●
VSOCK SEQ	●	●	●	●	●	●

TCP to TCP only

Redirect

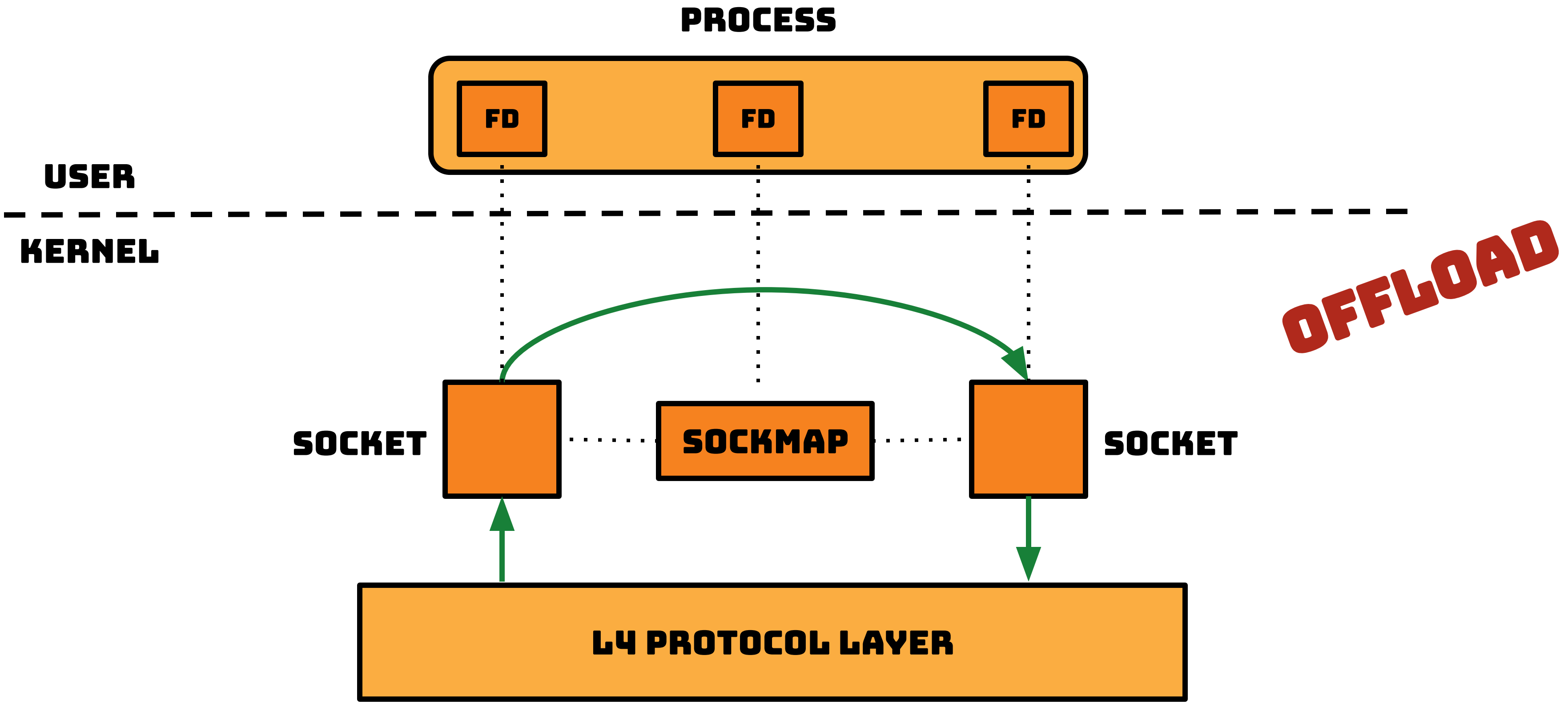
ingress to egress

Redirect use case → L7 network proxy

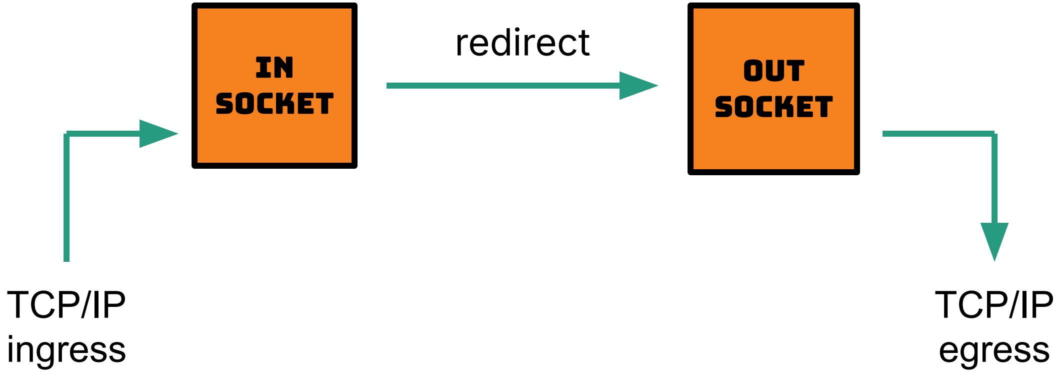


Examples: `socat`, `systemd-socket-proxyd`

Redirect use case → L7 network proxy



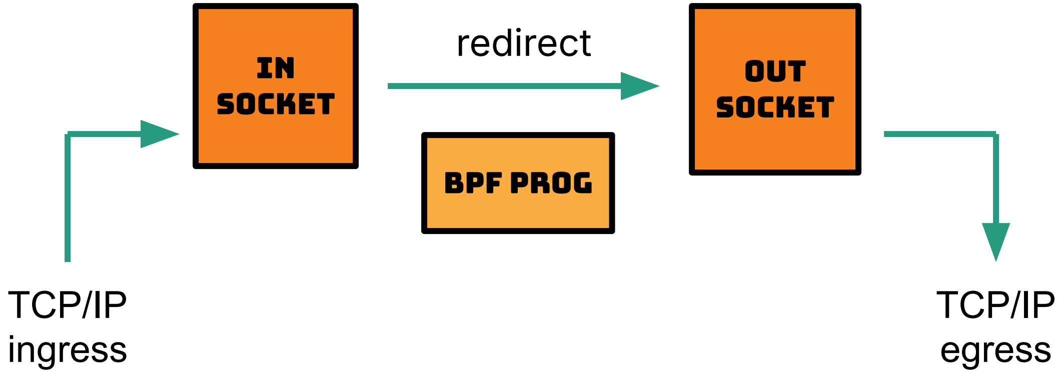
Redirect → ingress to egress



Like double `splice()`

(socket → pipe → socket)

Redirect → ingress to egress → How?



`BPF_PROG_TYPE_`**`SK_SKB`** prog

→ attached to `BPF_`**`SK_SKB_VERDICT`** hook

→ calls `bpf_sk_redirect_hash/map()` without any flags

→ returns `SK_`**`PASS`**

Redirect → ingress to egress → What?

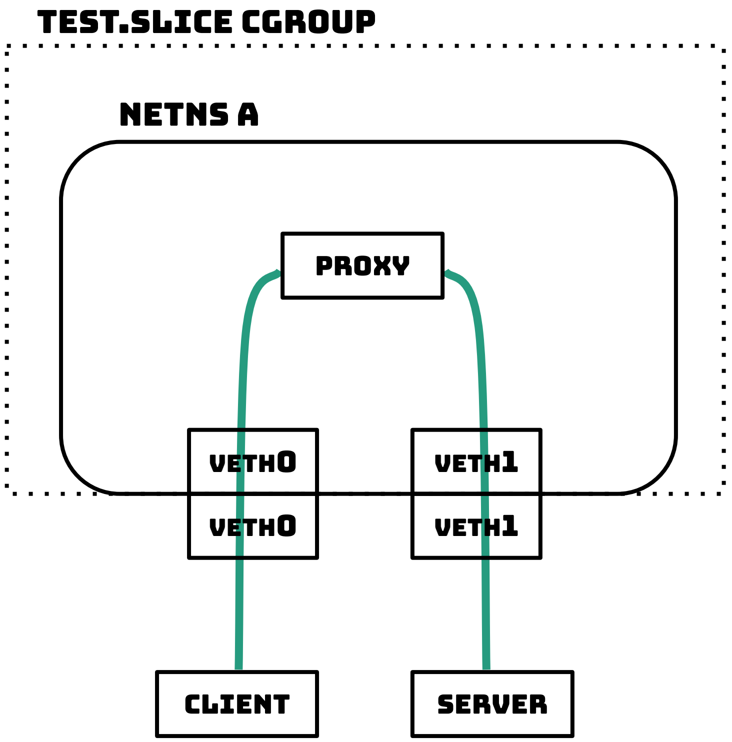
IN → OUT	TCP	UDP	UNIX STR	UNIX DGR	VSOCK STR	VSOCK SEQ
TCP	●	●	●	●	●	●
UDP	●	●	●	●	●	●
UNIX STR	●	●	●	●	●	●
UNIX DGR	●	●	●	●	●	●
VSOCK STR	●	●	●	●	●	●
VSOCK SEQ	●	●	●	●	●	●

any to any

Another benchmark time!

Image credit: Stopwatch Timer Chronometer - Free photo on Pixabay

Ping-pong test through a TCP proxy



https://github.com/jsitnicki/sockmap-project/blob/main/examples/ingress-to-egress/tcp_proxy.go

Ping-pong test through a TCP proxy

Spawn proxy inside the test.slice cgroup and in netns A

```
# (
        echo $BASHPID > /sys/fs/cgroup/test.slice/cgroup.procs
        taskset -c 0 ip netns exec A \
                     ./tcp_proxy -proxy="10.100.0.10:1111" -target="10.200.0.1:2222" &
)
```

Start TCP server in main netns

```
# taskset -c 2 sockperf server -i 10.200.0.1 -p 2222 --tcp &
```

Run TCP client in main netns

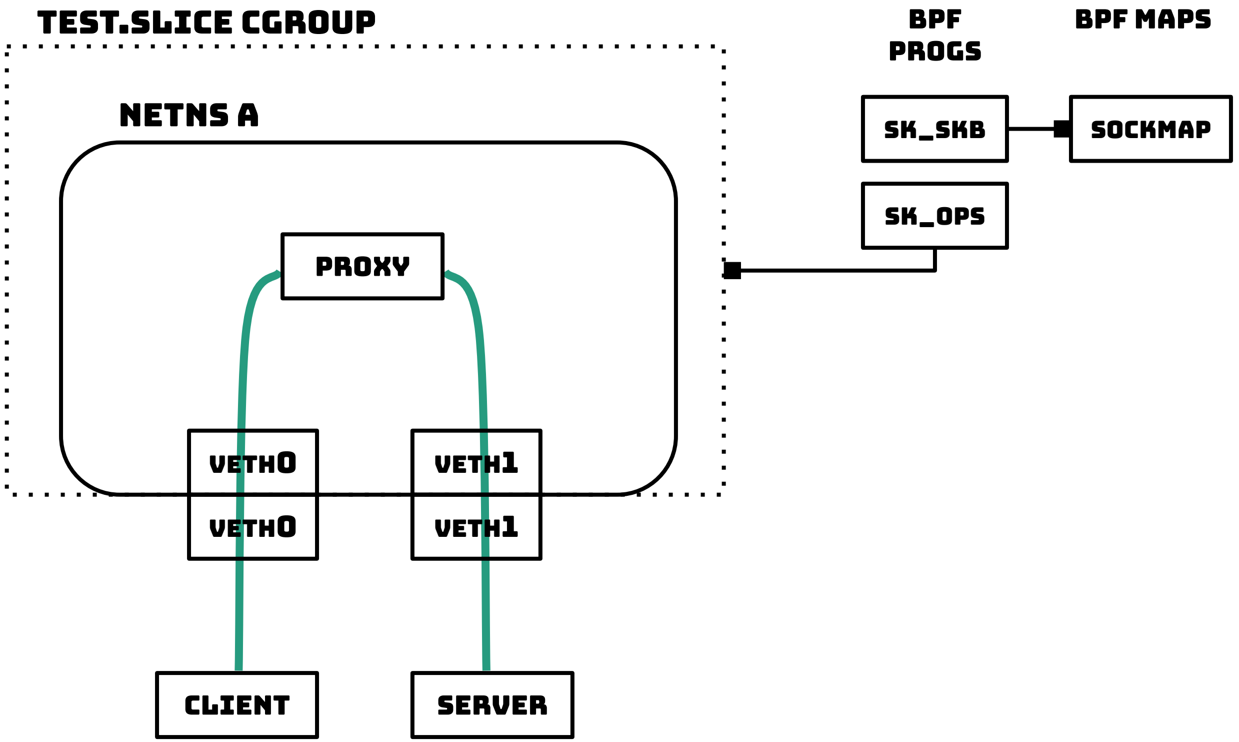
```
# taskset -c 4 sockperf ping-pong -i 10.100.0.10 -p 1111 --tcp --time 30
```

Ping-pong test through a TCP proxy

```
# taskset -c 4 sockperf ping-pong -i 10.100.0.10 -p 1111 --tcp --time 30
sockperf: Starting test...
sockperf: Test end (interrupted by timer)
sockperf: Test ended
sockperf: [Total Run] RunTime=30.001 sec; Warm up time=400 msec; …
sockperf: ========= Printing statistics for Server No: 0
sockperf: [Valid Duration] RunTime=29.550 sec; SentMessages=619300; ReceivedMessages=619300
sockperf: ====> avg-latency=23.827 (std-dev=16.129)
sockperf: # dropped messages = 0; # duplicated messages = 0; # out-of-order messages = 0
sockperf: Summary: Latency is 23.827 usec
sockperf: Total 619300 observations; each percentile contains 6193.00 observations
sockperf: ---> <MAX> observation = 1449.742
sockperf: ---> percentile 99.999 = 1382.814
sockperf: ---> percentile 99.990 = 1034.076
sockperf: ---> percentile 99.900 =   82.949
sockperf: ---> percentile 99.000 =   33.360
sockperf: ---> percentile 90.000 =   25.817
sockperf: ---> percentile 75.000 =   24.732
sockperf: ---> percentile 50.000 =   23.810
sockperf: ---> percentile 25.000 =   22.749
sockperf: ---> <MIN> observation =   15.487
```

24 ± 16 μsec

Same with SOCKMAP bypass

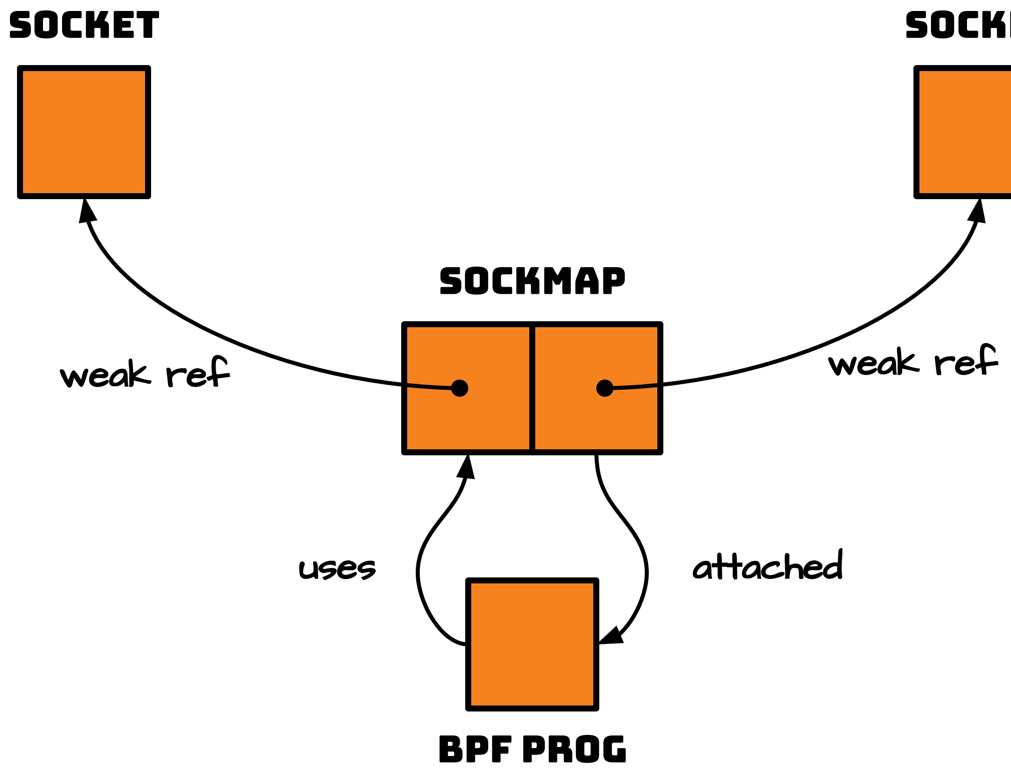


https://github.com/jsitnicki/sockmap-project/blob/main/examples/ingress-to-egress/redir_ingress_egress.bpf.c

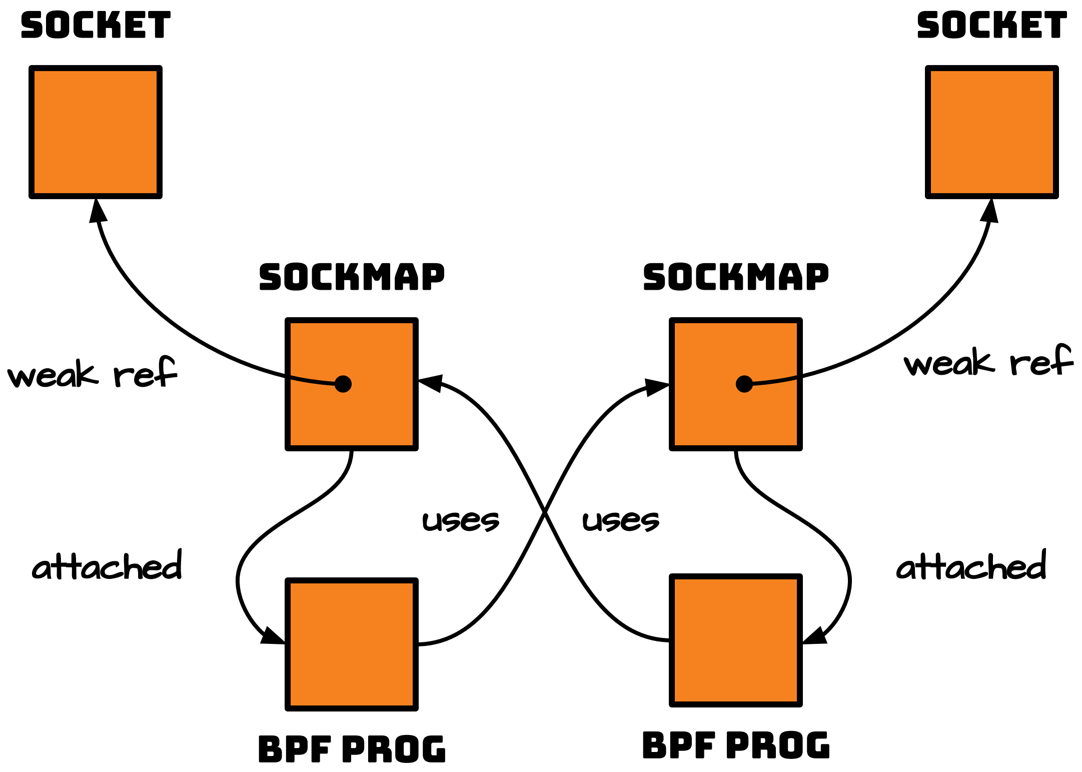
```c
SEC("sockops")
int sockops_prog(struct bpf_sock_ops *ctx)
{
    enum conn_dir dir;
    __u64 cookie;

    switch (ctx->op) {
    case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:
        dir = OUTGOING;
        break;
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
        dir = INCOMING;
        break;
    default:
        goto out;
    }

    cookie = bpf_get_socket_cookie(ctx);
    bpf_sock_map_update(ctx, &sock_map, &dir, /* flags= */ 0);
    bpf_map_update_elem(&conn_map, &cookie, &dir, BPF_ANY);
out:
    return SK_PASS;
}
```

```
SEC("sk_skb")
int redir_skb_prog(struct __sk_buff *skb)
{
    __u64 cookie = bpf_get_socket_cookie(skb);
    enum conn_dir *v, target;

    v = bpf_map_lookup_elem(&conn_map, &cookie);
    if (!v)
        goto err;

    switch (*v) {
    case INCOMING:
        target = OUTGOING;
        break;
    case OUTGOING:
        target = INCOMING;
        break;
    default:
        goto err;
    }

    return bpf_sk_redirect_map(skb, &sock_map, target, /* flags= */ 0);
err:
    __sync_fetch_and_add(&redir_errors, 1);
    return SK_DROP;
}
```

Two sockets in one sockmap



Two sockets in two sockmaps



Same with SOCKMAP bypass

```
# taskset -c 4 sockperf ping-pong -i 10.100.0.10 -p 1111 --tcp --time 30
sockperf: Starting test...
sockperf: Test end (interrupted by timer)
sockperf: Test ended
sockperf: [Total Run] RunTime=30.000 sec; Warm up time=400 msec; …
sockperf: ========= Printing statistics for Server No: 0
sockperf: [Valid Duration] RunTime=29.552 sec; SentMessages=738402; ReceivedMessages=738402
sockperf: ====> avg-latency=19.973 (std-dev=5.672)
sockperf: # dropped messages = 0; # duplicated messages = 0; # out-of-order messages = 0
sockperf: Summary: Latency is 19.973 usec
sockperf: Total 738402 observations; each percentile contains 7384.02 observations
sockperf: ---> <MAX> observation = 1238.486
sockperf: ---> percentile 99.999 =  696.232
sockperf: ---> percentile 99.990 =  147.685
sockperf: ---> percentile 99.900 =   64.345
sockperf: ---> percentile 99.000 =   38.314
sockperf: ---> percentile 90.000 =   22.524
sockperf: ---> percentile 75.000 =   21.908
sockperf: ---> percentile 50.000 =   18.355
sockperf: ---> percentile 25.000 =   17.427
sockperf: ---> <MIN> observation =   13.225
```

$\boxed{20 \pm 6.0\ \mu\text{sec}}$

Ping-pong test through a TCP proxy with and without SOCKMAP bypass

before: 24 ± 16 µsec

↓ - 17%

after: 20 ± 6.0 µsec

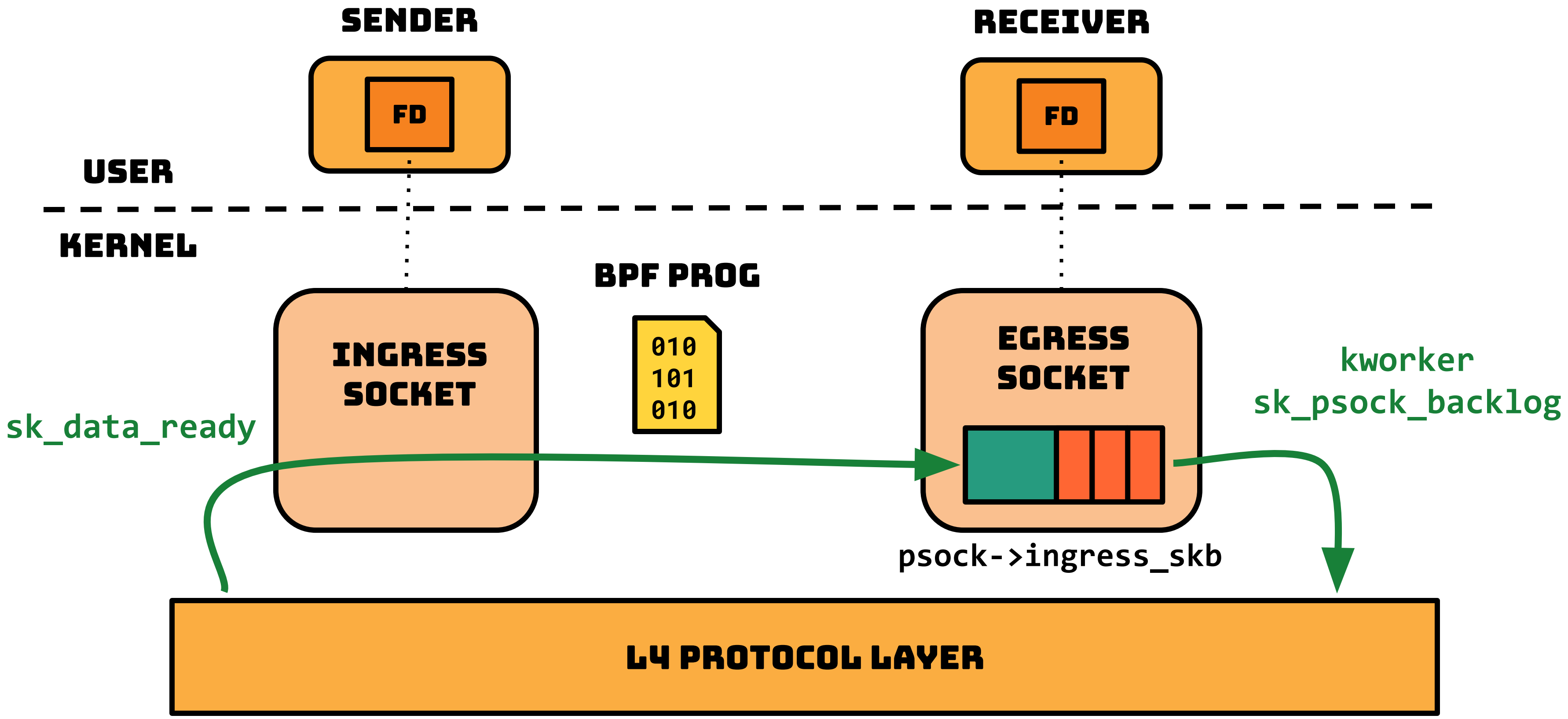
Run the benchmark yourself:
https://github.com/jsitnicki/sockmap-project/tree/main/examples/ingress-to-egress

But…

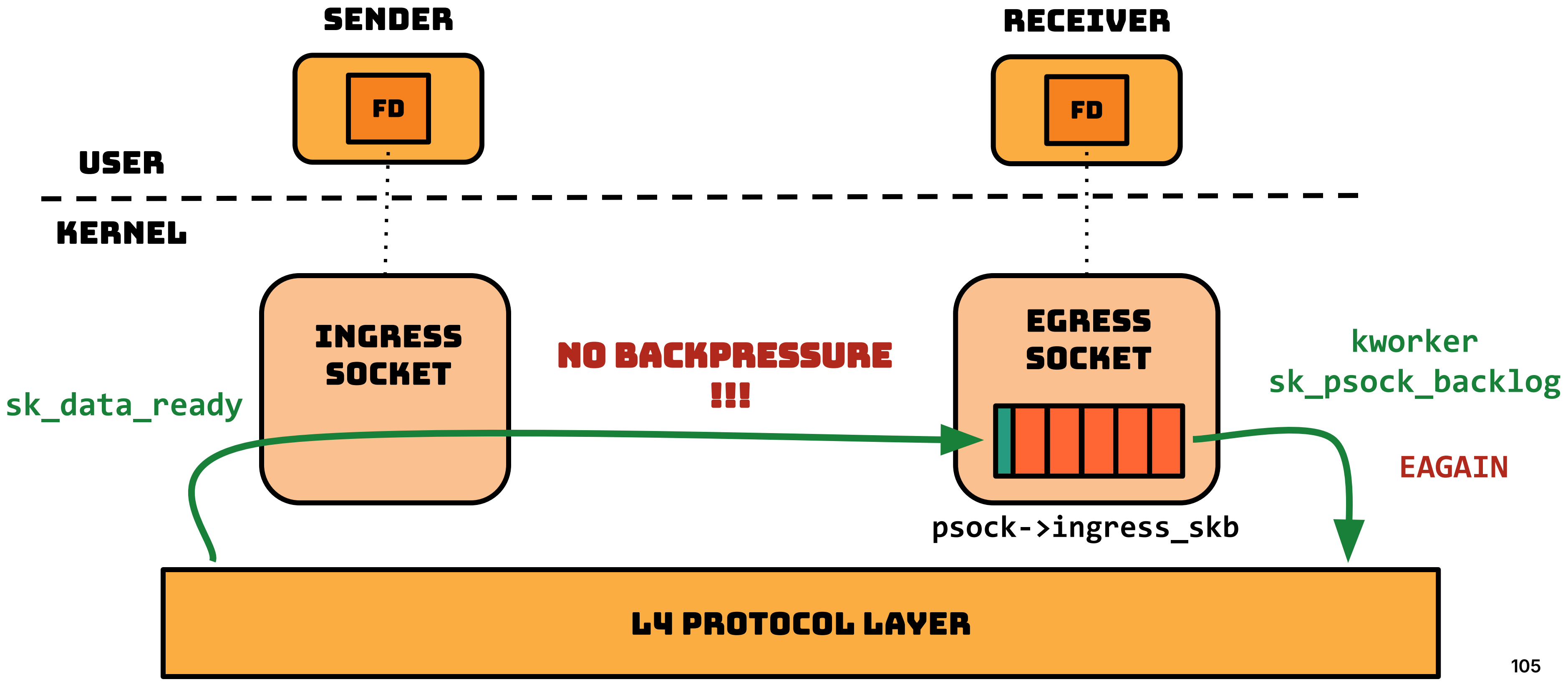
“there is always a but in this imperfect world!”

— Anne Brontë, The Tenant of Wildfell Hall

Redirect → ingress to egress → Internals



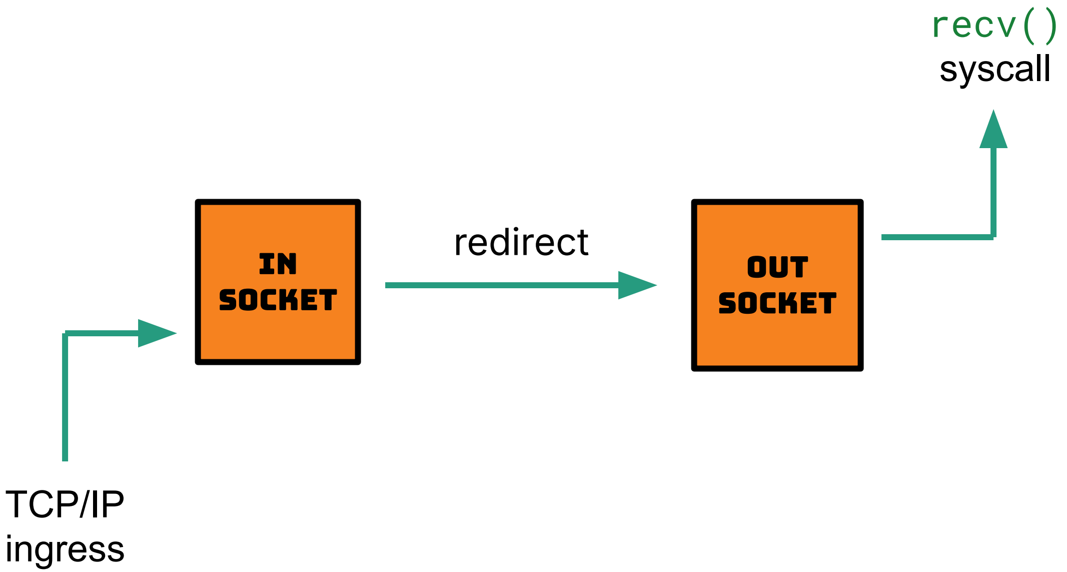
Redirect → ingress to egress → Internals



Redirect

ingress to local

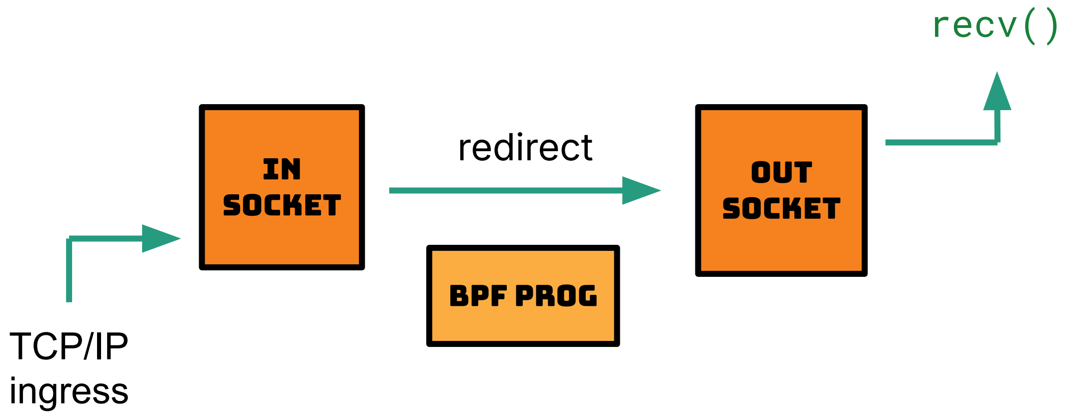
Redirect → ingress to local



Sort of like splice()

(socket → pipe)

Redirect → ingress to local → How?



BPF_PROG_TYPE_**SK_SKB** prog

→ attached to BPF_**SK_SKB_VERDICT** hook

→ calls bpf_**sk**_redirect_hash/map() with BPF_F_**INGRESS** flag

→ returns SK_**PASS**

Redirect → ingress to local → What?

IN → OUT	TCP	UDP	UNIX STR	UNIX DGR	VSOCK STR	VSOCK SEQ
TCP	🟢	🟢	🟢	🟢	⚪	⚪
UDP	🟢	🟢	🟢	🟢	⚪	⚪
UNIX STR	🟢	🟢	🟢	🟢	⚪	⚪
UNIX DGR	🟢	🟢	🟢	🟢	⚪	⚪
VSOCK STR	🟢	🟢	🟢	🟢	⚪	⚪
VSOCK SEQ	🟢	🟢	🟢	🟢	⚪	⚪

any to any but VSOCK

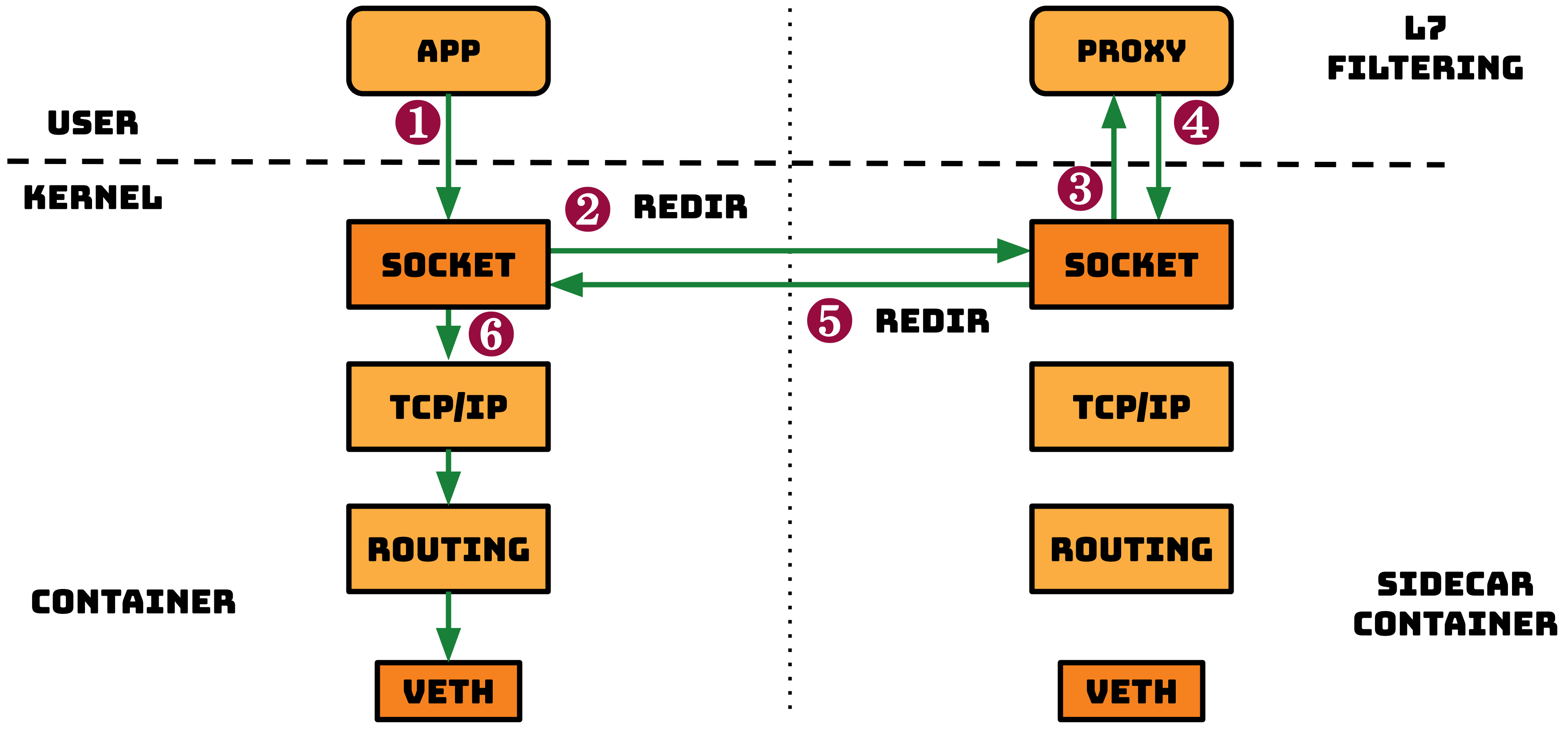
Cheatsheet - Redirect with SOCKMAP

redirect scenario	program type BPF_PROG_TYPE_*	attach type BPF_*	redirect helper function	redirect helper flags	in socket type	out socket type
send to local	SK_**MSG**	SK_**MSG_VERDICT**	bpf_**msg_redirect_***()	BPF_F_**INGRESS**	TCP	any but VSOCK
send to egress	SK_**MSG**	SK_**MSG_VERDICT**	bpf_**msg_redirect_***()	none	TCP	TCP
ingress to egress	SK_**SKB**	SK_**SKB_VERDICT**	bpf_**sk_redirect_***()	none	any	any
ingress to local	SK_**SKB**	SK_**SKB_VERDICT**	bpf_**sk_redirect_***()	BPF_F_**INGRESS**	any	any but VSOCK

Section

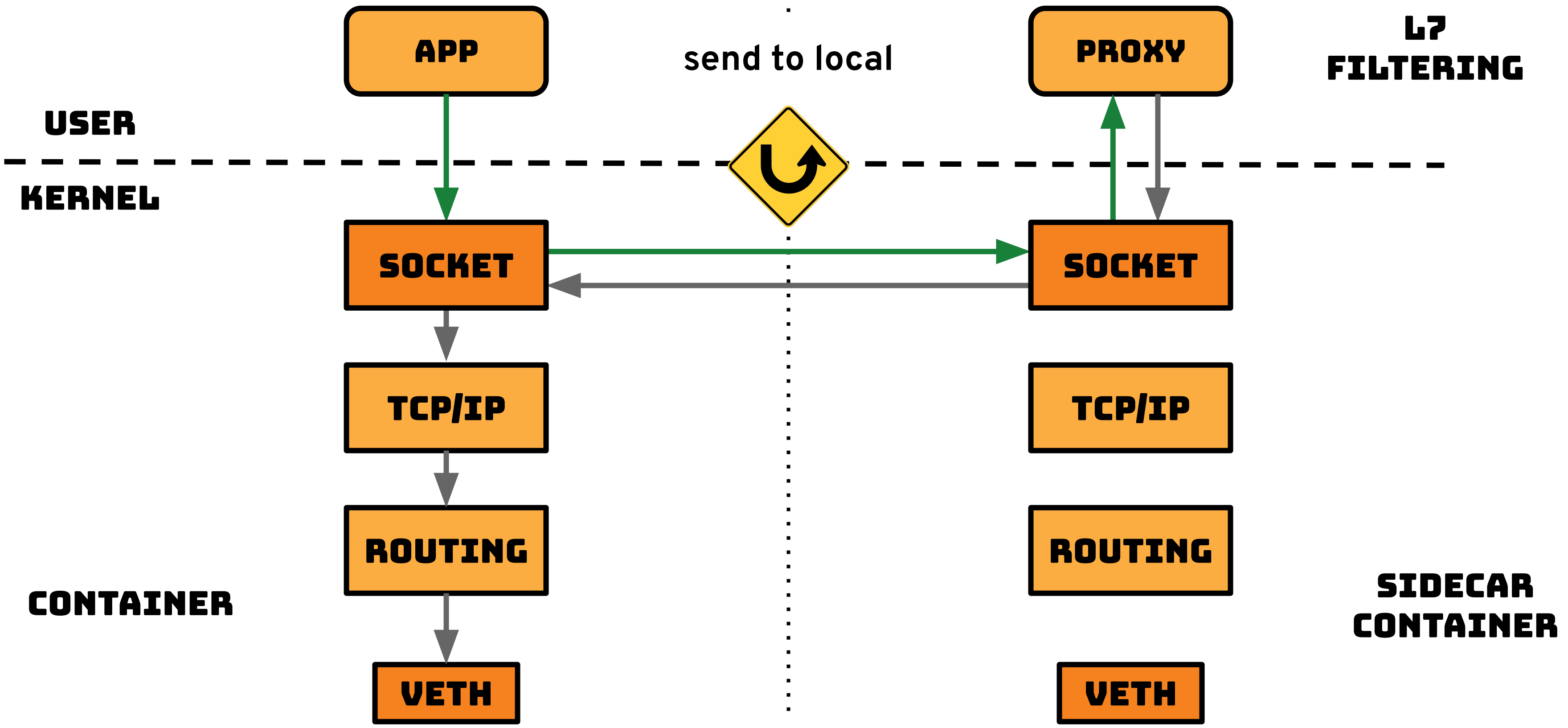
Real life use-cases

Cilium project (CNI for K8S)



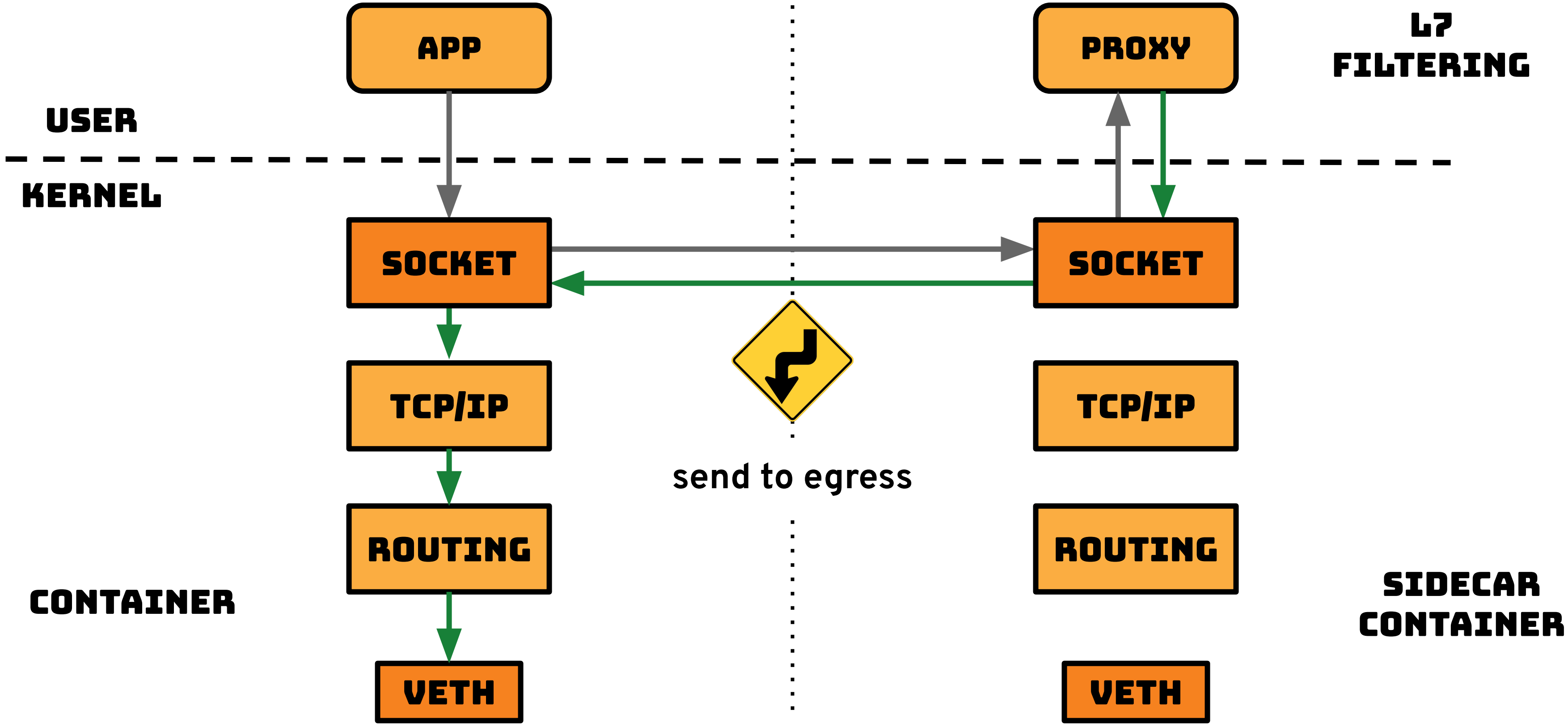
https://cilium.io/blog/2019/02/12/cilium-14/

Cilium project (CNI for K8S)



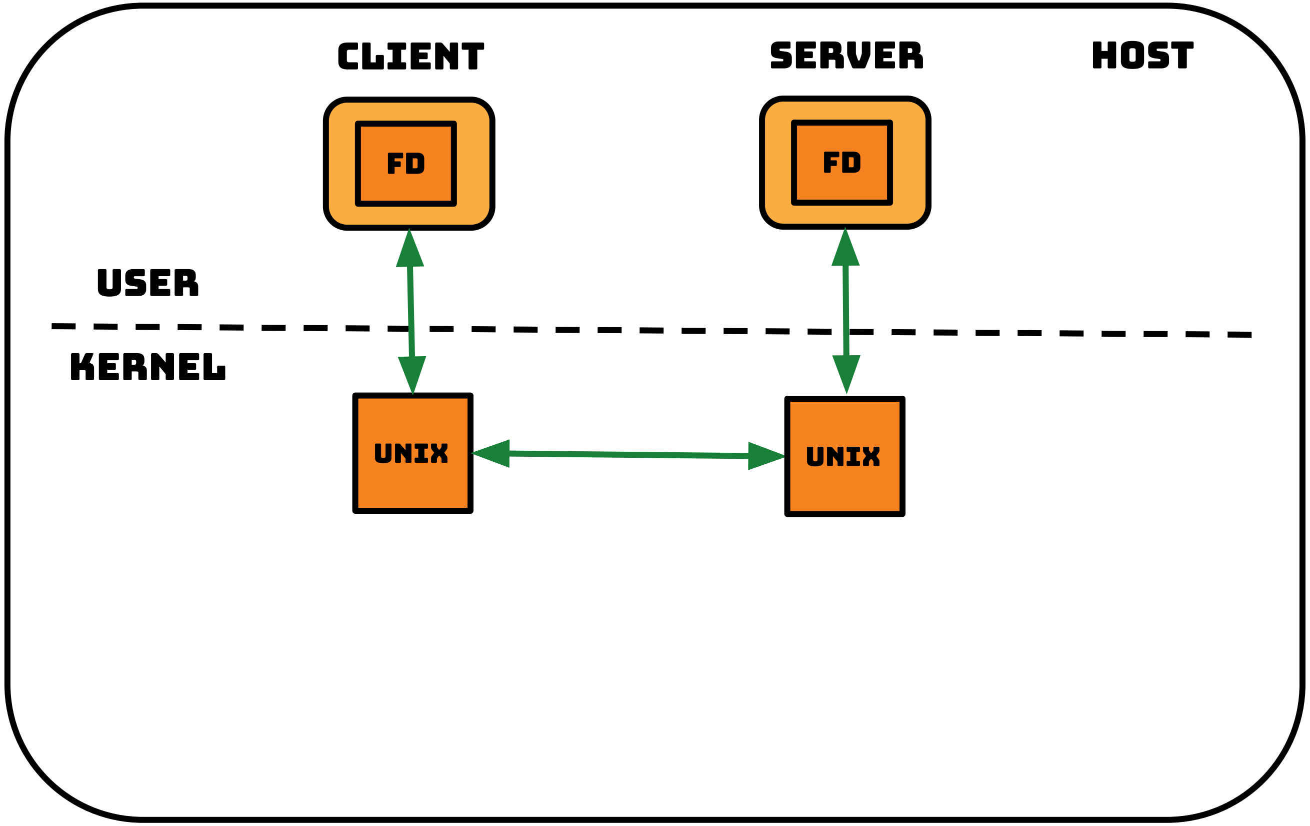
https://cilium.io/blog/2019/02/12/cilium-14/

Cilium project (CNI for K8S)



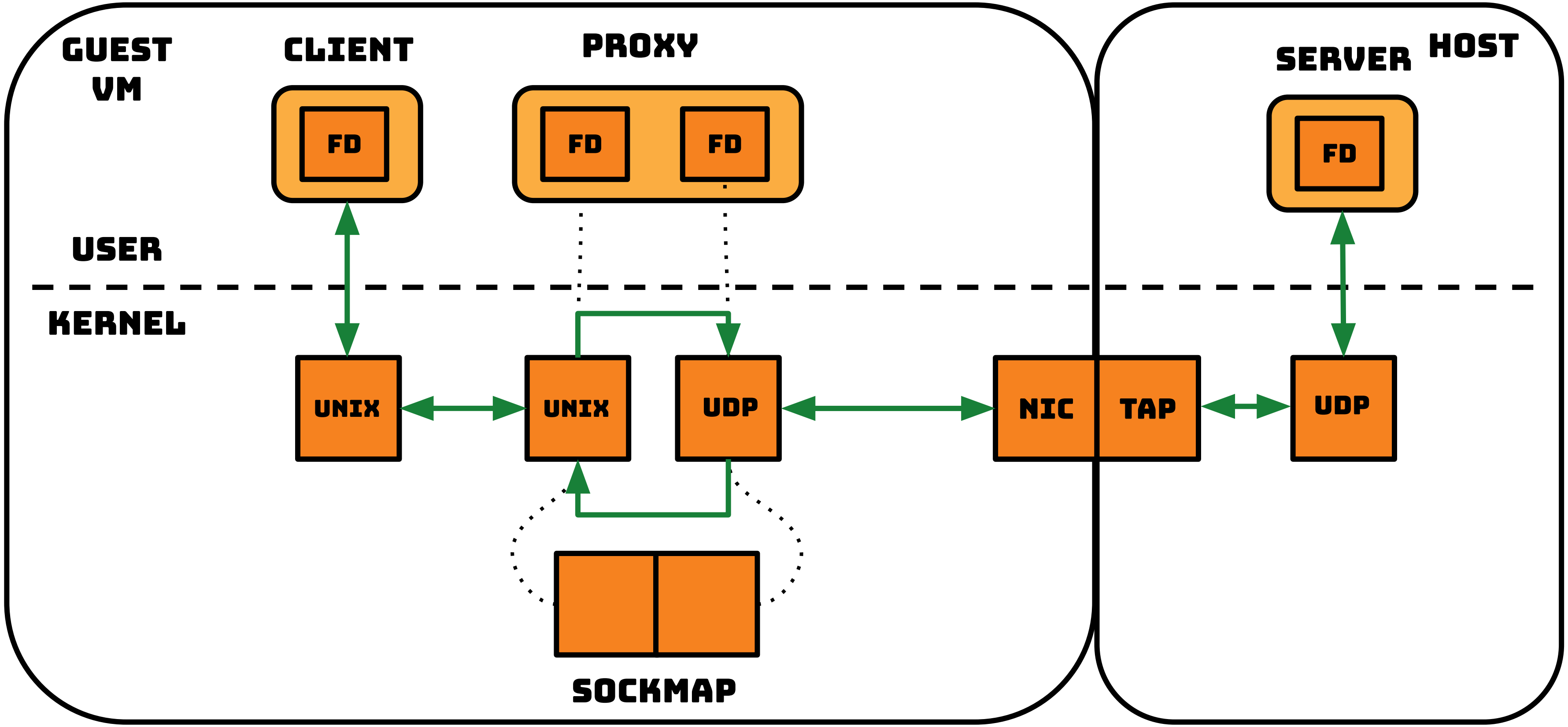
https://cilium.io/blog/2019/02/12/cilium-14/

Bytedance (TikTok)



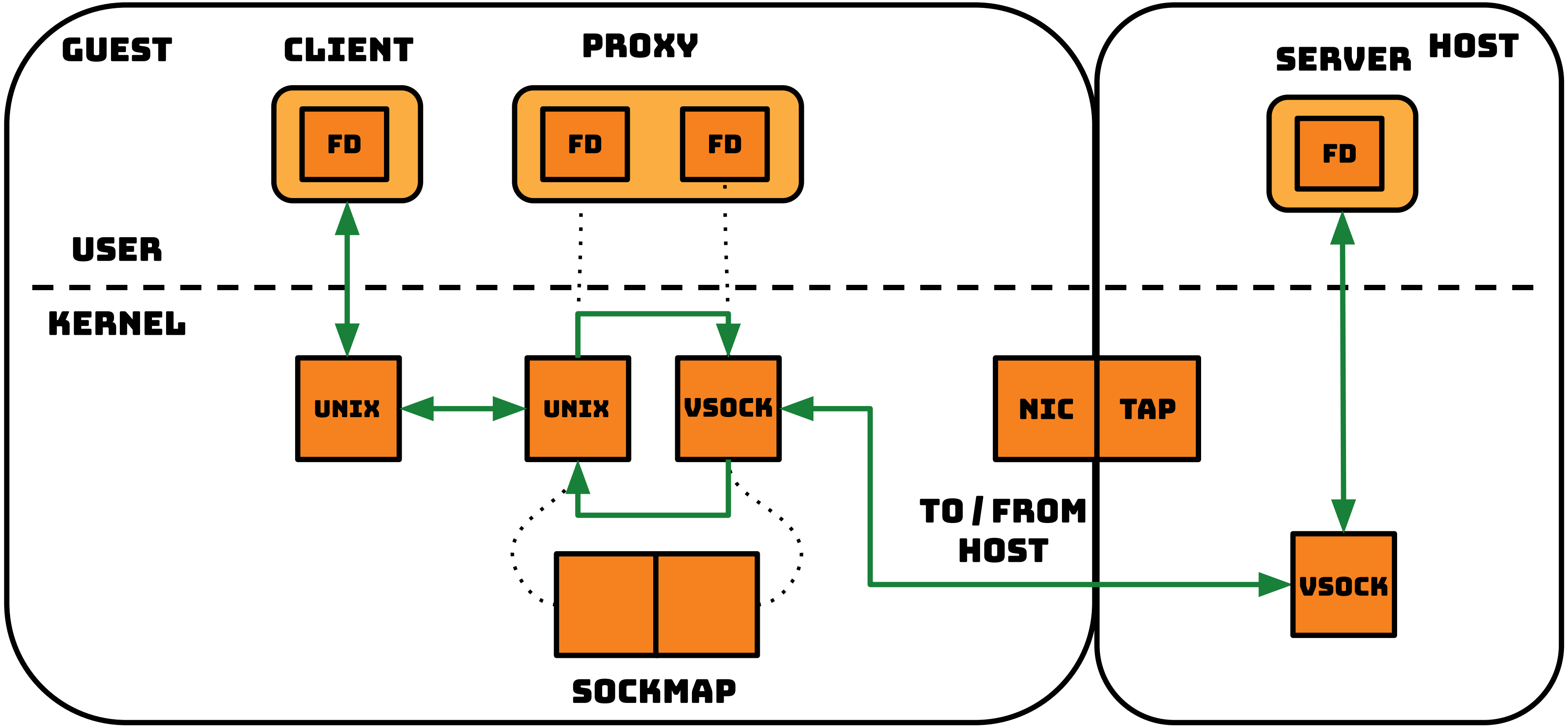
https://lore.kernel.org/bpf/20210331023237.41094-1-xiyou.wangcong@gmail.com/

Bytedance (TikTok)



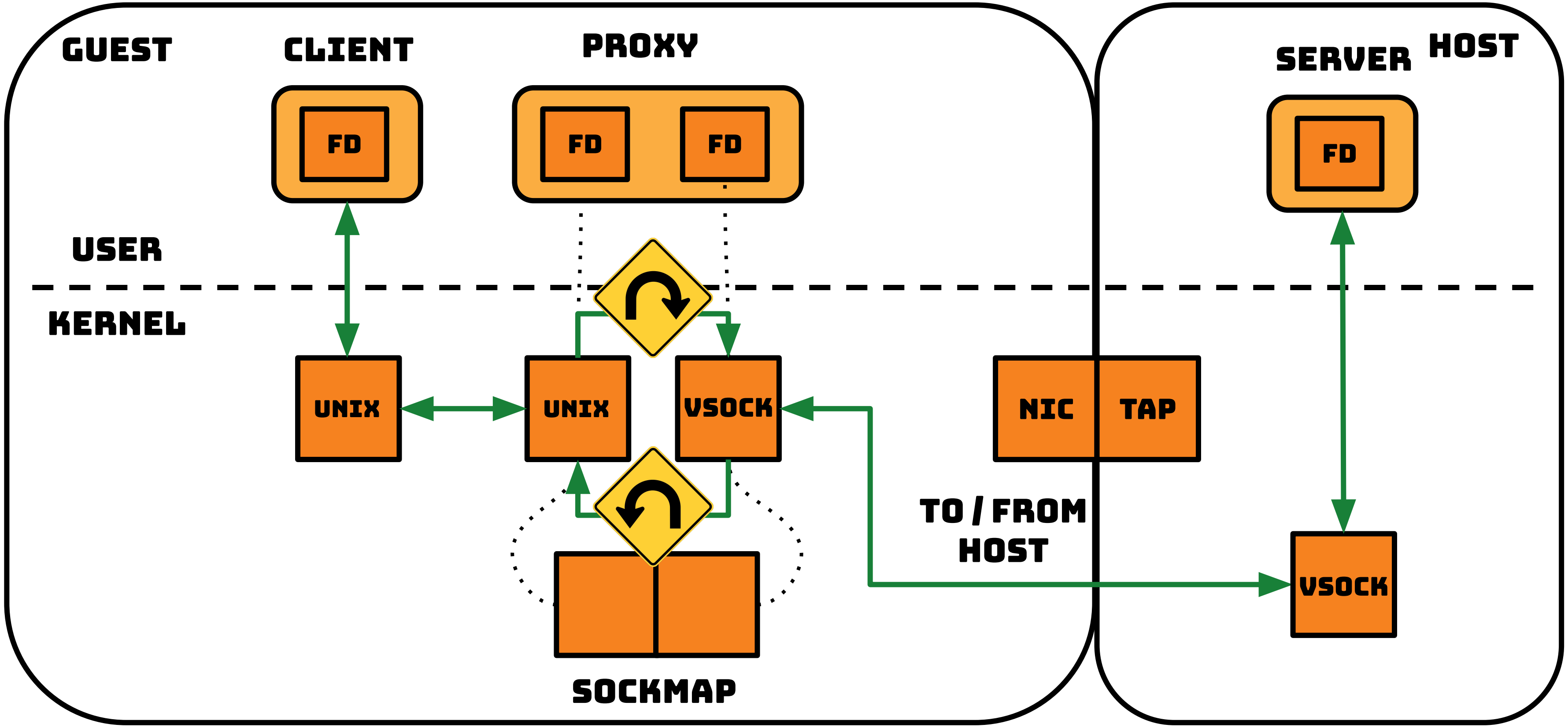
https://lore.kernel.org/bpf/20210331023237.41094-1-xiyou.wangcong@gmail.com/

Bytedance (TikTok) → Improved



https://lore.kernel.org/all/20230327-vsock-sockmap-v4-0-c62b7cd92a85@bytedance.com/

Bytedance (TikTok) → Improved v2



https://lore.kernel.org/all/20230327-vsock-sockmap-v4-0-c62b7cd92a85@bytedance.com/

Section

Where to learn more?

RESOURCES

1) **Linux Kernel → BPF Documentation → SOCKMAP and SOCKHASH map**

   See unit tests with API usage examples

2) **LPC 2018: Combining kTLS and BPF for Introspection and Policy Enforcement**

   See Daniel & John talk about Cilium SOCKMAP + kTLS use case (video, slides, paper)

3) **Cloudflare Blog: SOCKMAP - TCP splicing of the future**

   Read Marek review SOCKMAP from L7 proxy perspective

4) **eBPF Summit 2020: Steering connections to sockets with BPF socket lookup hook**

   Another use case for SOCKMAP as a container (video, slides, code)

THANK YOU

Reach out:

jakub@cloudflare.com

jsitnicki @ LinkedIn

Mailing lists:

bpf@vger.kernel.org

netdev@vger.kernel.org

code & slides repo

https://github.com/jsitnicki/sockmap-project

Attribution

- Uncle Sam Wants You, Public Domain Pictures, CC0
- A bunch of question marks, Wikimedia Commons, CC BY-SA 4.0 DEED
- Nf knots, getarchive.net, CC0
- Construction blocks, vectorportal.com, CC BY 4.0 DEED
- Container crane image, vectorportal.com, CC BY 4.0 DEED
- Golden Gate bridge silhouette, rawpixel.com, CC0
- Evolution-des-wissens.jpg, Wikimedia Commons, CC BY-SA 3.0 DEED
- Leather pouch, rawpixel.com, CC0
- Redirection.svg, Wikimedia Commons, CC BY 3.0 DEED