Atelier sur eBPF au Capitole du Libre

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 19 novembre 2025

https://www.bortzmeyer.org/ebpf-capitole-libre.html

Au Capitole du Libre <https://www.bortzmeyer.org/capitole-du-libre-2025.html> de novembre 2025, j'ai participé à un passionnant atelier de Maxime Chevallier et Alexis Lothoré sur la programmation réseau en eBPF. Le but était de mettre en œuvre un bloqueur de publicité (ou autre contenu qu'on ne voulait pas voir), agissant via les requêtes DNS, que le programme va intercepter et analyser.

eBPF désigne à la fois un jeu d'instructions machine, normalisé dans le RFC 9669[1], et un système permettant de produire des programmes utilisant ce jeu d'instructions, et des les installer dans le noyau Linux. eBPF permet ainsi d'ajouter des fonctions au noyau, sans modifier et recompiler celui-ci (ce qui est vraiment une tâche difficile) et sans redémarrer la machine. Je vous renvoie à mon article sur le RFC 9669 pour plus de détail. Un exemple d'utilisation de eBPF dans le contexte du DNS est dans le serveur NSD <https://blog.apnic.net/2025/08/04/experimental-support-for-af_xdp-sockets-in-nsd/>.

Le cahier des charges de l'atelier était d'écrire (en C) et de faire tourner un programme eBPF qui va regarder toutes les requêtes DNS de la machine, les comparer à une liste de blocage, et stopper ces requêtes si le nom de domaine est dans cette liste. Si vous avez un peu de temps, je recommande de faire l'exercice vous-même, téléchargez le support de l'atelier (`git clone https://github.com/bootlin/ebpf-workshop-cdl`), lisez le fichier `Readme.md` (il est en anglais) et essayez. Cet article que vous lisez est plutôt destiné aux gens qui veulent la solution.

Comme le système eBPF sur Linux n'est pas d'une clarté parfaite, il vaut mieux commencer par un programme trivial. Attention, comme indiqué dans le `Readme.md`, ça marchera…ou pas, selon la version exacte du système d'exploitation que vous utilisez (j'ai tout fait sur une Ubuntu "*stable*", la version 24.04). Quelques paquetages à installer :

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc9669.txt`

```
% sudo apt install linux-tools-common build-essential clang libbpf-dev
```

Après cela, si, quand vous essayez de lancer bpftool, vous avez :

```
% bpftool
WARNING: bpftool not found for kernel 6.14.0-112033

  You may need to install the following packages for this specific kernel:
    linux-tools-6.14.0-112033-tuxedo
    linux-cloud-tools-6.14.0-112033-tuxedo

  You may also want to install one of the following packages to keep up to date:
    linux-tools-tuxedo
    linux-cloud-tools-tuxedo
```

N'essayez pas d'installer les paquetages cités, ça ne servira sans doute pas. À la place, installez à la main :

```
% git clone --recurse-submodules https://github.com/libbpf/bpftool.git
% cd src
% make
% sudo make install
```

Maintenant, écrivons le programme trivial qui ne fait pas grand'chose :

```
% cat hello.bpf.c

#include <linux/bpf.h>
#include <linux/pkt_cls.h>
#include "bpf/bpf_helpers.h"
#include "bpf/bpf_endian.h"

SEC("tc")
int hello(struct __sk_buff *skb)
{
  bpf_printk("Hello, world !");
  return TC_ACT_OK;
}

char __license[] SEC("license") = "GPL";
```

Ce programme crée une fonction (qui sera donc exécutée par le noyau), qui sera appelée à chaque paquet réseau et affichera le classique « Bonjour, tout le monde », avant de laisser passer le paquet. Compilons-le en eBPF :

```
% clang -Wall  -g -O1 -target bpf -c hello.bpf.c -o hello.bpf.o
```

(Pour des raisons que je ne comprends pas bien, les options `-g` et `-On` avec n ¿ 1 semblent indispensables.) La compilation va produire du code eBPF :

```
% file hello.bpf.o
hello.bpf.o: ELF 64-bit LSB relocatable, eBPF, version 1 (SYSV), with debug_info, not stripped
```

Le désassembleur ne fonctionne pas, je ne sais pas pourquoi (normalement, il devrait) :

```
% objdump -d hello.bpf.o
hello.bpf.o:     file format elf64-little
objdump: can't disassemble for architecture UNKNOWN!
```

Bon, ce n'est pas tout de produire un binaire eBPF, il faut le charger dans le noyau, et dire au noyau de l'exécuter pour chaque paquet sortant. Il existe plusieurs méthodes pour cela (on peut aussi écrire son propre programme qui fait le chargement), on va utiliser tc, et attacher le code eBPF à l'interface réseau active, ici `wlp1s0` :

```
% sudo tc qdisc add dev wlp1s0 clsact
% sudo tc filter add dev wlp1s0 egress bpf direct-action object-file  hello.bpf.o sec tc
```

(La syntaxe de tc est merveilleuse.) Une fois que tout cela est fait sans erreur, on eut afficher les messages :

```
%  sudo bpftool prog tracelog
 irq/80-iwlwifi:-928     [011] b.s3. 200018.139564: bpf_trace_printk: Hello, world !
 irq/81-iwlwifi:-929     [008] b.s3. 200018.139564: bpf_trace_printk: Hello, world !
   Socket Thread-576739  [000] b..1. 200018.139832: bpf_trace_printk: Hello, world !
   Socket Thread-576739  [000] b..1. 200018.139976: bpf_trace_printk: Hello, world !
...
```

Et voilà, à chaque paquet sortant (la directive "*egress*" dans l'appel à tc), on a un message affiché.

Comme eBPF inclut un vérificateur qui examine le code avant de l'exécuter (pour éviter qu'un programme eBPF bogué ne plante tout le noyau), des erreurs apparemment innocentes (comme d'aller visiter de la mémoire en dehors de nos variables) va se traduire par des erreurs au chargement, assez difficiles à déboguer :

```
libbpf: prog 'dns_filter': BPF program load failed: Permission denied
libbpf: prog 'dns_filter': -- BEGIN PROG LOAD LOG --
0: R1=ctx() R10=fp0
; int             dns_filter(struct __sk_buff *skb) @ dns-filter.bpf.c:107
0: (bf) r7 = r1                       ; R1=ctx() R7_w=ctx()
1: (b7) r6 = 0                        ; R6_w=0
...
124: (bf) r3 = r1                     ; R1_w=scalar(id=253) R3_w=scalar(id=253)
125: (0f) r3 += r2                    ; R2_w=0 R3_w=scalar(id=253+0)
126: (71) r3 = *(u8 *)(r3 +0)
R3 invalid mem access 'scalar'
processed 9460 insns (limit 1000000) max_states_per_insn 4 total_states 129 peak_states 129 mark_read 128
-- END PROG LOAD LOG --
libbpf: prog 'dns_filter': failed to load: -13
libbpf: failed to load object 'dns-filter.bpf.o'
Unable to load program
```

Si vous voyez cela, dites-vous que votre programme doit être trop laxiste et, pour reprendre une expression utilisée par les animateurs de l'atelier, qu'il jardine en dehors de ses plate-bandes.

Maintenant, revenons au vrai projet. L'atelier était prévu pour une découverte progressive, et c'est ce que vous ferez si vous suivez le support de l'atelier mentionné plus tôt (`git clone https://github.com/bootl`
Ici, je vais simplement présenter le résultat final. Mon code complet (légèrement différent de celui qui figure en correction dans les documents de l'atelier) est . Je ne vais mentionner ici que le plus important, des commentaires dans le source éclairent d'autres points.

La liste des domaines bloqués est un dictionnaire, stucture de données fournie par eBPF pour communiquer entre le programme dans le noyau, et l'extérieur :

```
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 2);
    __type(key, int);
                    __type(value, char[253]);
} blocklist     SEC(".maps");
```

On va utiliser beaucoup de structures de données TCP/IP dont la description est déjà fournie par Linux (comme `struct udphdr` pour un en-tête UDP). Il n'y en a apparemment pas pour le DNS, donc on la crée :

```
struct dnshdr {
    __be16          trans_id;
    __be16          flags;
    __be16          nr_quest;
    __be16          nr_answ;
    __be16          nr_auth_rr;
    __be16          add_rr;
    };
```

Pour l'analyse des paquets, on va charger les octets dans nos structures de données, à partir d'un index (la variable `offset`) avec la fonction BPF `bpf_skb_load_bytes`. Voici les étapes :

```
/* Ce paquet est-il un paquet IP ? Chargeons-le dans ethdhr avant de
regarder son champ Protocol. */
ret = bpf_skb_load_bytes(skb, offset, &ethhdr, sizeof(ethhdr));
if (ret) {
    return TC_ACT_OK;
}
if ((ethhdr.h_proto != __bpf_constant_htons(ETH_P_IPV6)) &&
        (ethhdr.h_proto != __bpf_constant_htons(ETH_P_IP))) {
        return TC_ACT_OK;
}
```

Si une fonction échoue (j'ai omis ce test par la suite mais dans le vrai code, il faut le mettre), ou bien si le paquet n'est pas de l'IP, on le laisse passer (on renvoie OK à tc). Ensuite, après avoir sauté l'en-tête IP (attention, il n'a pas la même taille en IPv4 et IPv6), on regarde si c'est bien de l'UDP, et s'il utilise bien le port 53 du DNS :

```
if (l4_proto != IPPROTO_UDP) {
        return TC_ACT_OK;
}
ret = bpf_skb_load_bytes(skb, offset, &udph, sizeof(udph));
if (__bpf_constant_ntohs(udph.dest) != 53) {
        return TC_ACT_OK;
}
```

On va ensuite récupérer le nom de domaine demandé (attention à l'analyse d'un paquet DNS, lisez bien le RFC 9267) :

```
offset += sizeof(dnsh);
ret = parse_query(skb, offset, query, 253);
ctx.query = query;
```

On va ensuite tester ce nom `query` pour voir s'il est dans la liste de blocage, avec `bpf_for_each_map_elem`. Notez le code de retour `TC_ACT_SHOT`, qui dit à tc de jeter le paquet sans autre forme de procès.

```
bpf_for_each_map_elem(&blocklist, dns_check, &ctx, 0);
if (ctx.match) {
        bpf_printk("*REJECTED* DNS query of %s", query);
        return TC_ACT_SHOT;
}
bpf_printk("Accepted DNS query of %s", query);
return TC_ACT_OK;
```

Reste un petit détail : il faut peupler cette liste de blocage (la variable `blocklist`, de type dictionnaire). Une méthode possible est d'utiliser la commande bpftool mais, évidemment, en vrai, on écrirait un programme avec une interface utilisateur plus agréable :

```
% sudo bpftool map update name blocklist key 0 0 0 0 value $(printf '%-253s' x.com | tr ' ' '\0' | xxd -i -c 253
```

Armé de tout cela, on peut utiliser cette suite de commandes pour compiler le programme, attacher l'eBPF via tc, remplir la liste de blocage et regarder le résultat :

```
#!/bin/sh

sudo tc qdisc del dev wlp1s0 clsact
sudo tc qdisc add dev wlp1s0 clsact
clang -Wall -g -O1 -target bpf -c dns-filter.bpf.c -o dns-filter.bpf.o
sudo tc filter add dev wlp1s0 egress bpf direct-action object-file dns-filter.bpf.o sec tc
sudo bpftool map update name blocklist key 0 0 0 0 value $(printf '%-253s' x.com | tr ' ' '\0' | xxd -i -c 253|
sudo bpftool map update name blocklist key 1 0 0 0 value $(printf '%-253s' facebook.com | tr ' ' '\0' | xxd -i -
sudo bpftool prog tracelog
```

Testons-le :

```
% ping facebook.com
ping: facebook.com: Temporary failure in name resolution

% ping bootlin.com
PING bootlin.com (87.98.181.233) 56(84) bytes of data.
64 bytes from bootlin.com (87.98.181.233): icmp_seq=1 ttl=49 time=15.8 ms
```

Et, affiché par bpftool :

```
systemd-resolve-1112    [000] b..1. 208052.900167: bpf_trace_printk: *REJECTED* DNS query of facebook.com
systemd-resolve-1112    [000] b..1. 208052.900183: bpf_trace_printk: *REJECTED* DNS query of facebook.com
systemd-resolve-1112    [005] b..1. 208264.967434: bpf_trace_printk: Accepted DNS query of bootlin.com
```

Objectif atteint. On peut aller boire une bière. Notez que vous pouvez afficher la liste de blocage :

```
% sudo bpftool map dump name blocklist
[{
        "key": 0,
        "value": "x.com"
    },{
        "key": 1,
        "value": "facebook.com"
    }
]
```

Quelques détail toutefois. D'abord, le programme de filtrage étant sommaire, il ne bloque que le nom exact dans la liste, pas ses sous-domaines :

```
 % dig facebook.com
;; communications error to 127.0.0.53#53: timed out
...
;; no servers could be reached

% dig www.facebook.com
...
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 54041
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
...
;; ANSWER SECTION:
www.facebook.com. 2724 IN CNAME star-mini.c10r.facebook.com.
star-mini.c10r.facebook.com. 29 IN A 157.240.253.35

;; Query time: 22 msec
;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
;; WHEN: Mon Nov 17 16:08:00 CET 2025
;; MSG SIZE  rcvd: 90
```

Corriger cette limite est laissé au lecteur ou à la lectrice (notez toutefois que dans l'atelier, il y avait environ vingt participants et pas de participante).

Ensuite, puisque tc va purement et simplement jeter le paquet, le client DNS n'aura aucune information, réessaiera et finira par laisser tomber mais au bout d'un délai qui est certainement pénible pour l'utilisateur. Il vaudrait mieux fabriquer une réponse mensongère, genre NXDOMAIN. Je ne suis pas sûr que cela soit facilement faisable avec tc mais il existe d'autres façons d'exécuter de l'eBPF donc le problème est certainement soluble.

Enfin, le DNS ne marche pas que sur UDP, il fonctionne aussi sur TCP (RFC 7766). Le programme ci-dessus peut donc être facilement contourné :

```
% dig @9.9.9.9 facebook.com
;; communications error to 9.9.9.9#53: timed out
...;; no servers could be reached

% dig +tcp @9.9.9.9 facebook.com
...
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 24690
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
...
;; ANSWER SECTION:
facebook.com.  30 IN A 57.144.222.1

;; Query time: 7 msec
;; SERVER: 9.9.9.9#53(9.9.9.9) (TCP)
;; WHEN: Mon Nov 17 16:16:59 CET 2025
;; MSG SIZE  rcvd: 63
```

Là encore, si vous voulez perfectionner ce progrmme, n'hésitez pas (mais c'est plus difficile, une requête DNS peut se retrouver répartie dans deux paquets IP différents).