Conception d'un EDR en Rust.

Création d'un moteur de détection de process anormaux en Rust.

Auteur : Deroubaix Sasha

22 March 2026

Table des matieres

1 Technologies utilisées

1.1 Rust

Rust est un langage de programmation système axé sur la sécurité mémoire et les performances. Il garantit l'absence de data races et de déréférencements invalides à la compilation, sans ramasse-miettes. Ces propriétés en font un choix naturel pour du code bas niveau comme un EDR, où les erreurs mémoire peuvent avoir des conséquences critiques.

1.2 eBPF

eBPF (*extended Berkeley Packet Filter*) est une technologie du noyau Linux permettant d'exécuter du bytecode sandboxé directement dans le kernel, sans modifier son code source ni charger de module noyau. Les programmes eBPF sont attachés à des points d'accroche (*tracepoints*, *kprobes*, etc.) et s'exécutent en réponse à des événements système. Le kernel vérifie statiquement le bytecode avant de l'exécuter pour garantir qu'il ne peut pas le crasher.

1.3 Aya

Aya est une librairie Rust pour écrire et charger des programmes eBPF entièrement en Rust, côté kernel comme côté user-land. Elle évite de devoir passer par du C pour la partie eBPF et s'intègre naturellement dans l'écosystème Rust (Cargo, traits, types).

2 Architecture du projet

2.1 Vue d'ensemble

Pour l'architecture du projet, il est nécessaire d'avoir 3 composants pour utiliser la librairie rust Aya:

1. Un package servant à créer un module eBPF pour observer les évènements.
2. Un package User-land servant à distribuer le binaire eBPF dans le noyau et à nous afficher ses résultats.
3. Une librairies pour interfacer le kernel-land et le User-land.

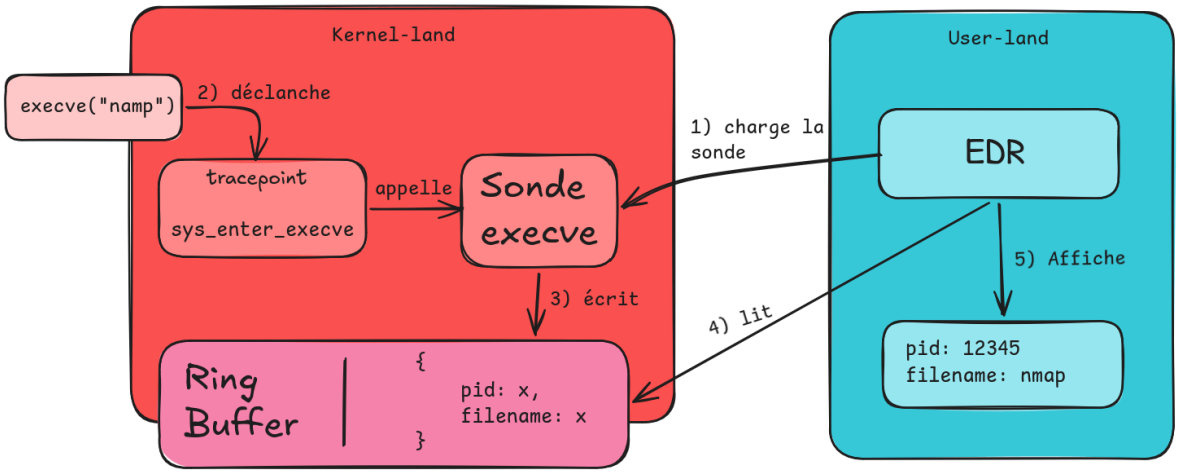

Fig. 1. – Architecture globale du projet EDR

2.2 Le workspace

A la racine de notre projet. nous allons donc créer un workspace. pour cela il suffit de créer un fichier `Cargo.toml` comme ceci:

```
[workspace]
resolver = "2"
members = [
    "edr",
    "edr-common",
    "edr-ebpf",
]
```

Nous déclarons les 3 packages qu'il y aura dans notre workspace.

> **Attention**
>
> Il est nécessaire d'utiliser le resolver `2` (2021) pour utiliser la librairie Aya sans erreurs.

2.3 Création des packages

Une fois le `Cargo.toml` créé, il faut créer notre packages via la commande `cargo new ...` (`--lib` si nous ne voulons pas de `main()`).

```
cargo new edr
cargo new edr-common --lib
cargo new edr-ebpf --lib
```

nous obtenons cette structure de projet :

```
├── Cargo.toml
├── edr
│   ├── Cargo.toml
│   └── src
│       └── main.rs
├── edr-common
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── edr-ebpf
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
```

Attention

`edr-ebpf` doit être une lib car ce bytecode ebpf n'aura pas de fonction `main()`.

Maintenant il faut éditer chaque `Cargo.toml` de chaque packages.

2.3.1 edr-ebpf

```
[package]
name = "edr-ebpf"
version = "0.1.0"
edition = "2024"

[dependencies]
aya-ebpf = "0.1"
edr-common = { path = "../edr-common" }

[lib]
crate-type = ["cdylib"]
```

- Nous utiliserons la librairie `aya-ebpf` pour construire notre binaire eBPF.
- Pour eviter de compiler vers une lib rust (`.rlib`) il faut préciser le type de sortie en `cdylib` pour que ce soit compréhensible par le kernel.

pour compiler en ebpf il faut créer ce fichier `.cargo/config.toml` et mettre ceci:

```
[build]
target = "bpfel-unknown-none"

[unstable]
build-std = ["core"]
```

2.3.2 edr

```
[package]
name = "edr"
version = "0.1.0"
edition = "2024"

[dependencies]
aya = { version = "0.13", features = ["async_tokio"] }
aya-log = "0.2"
tokio = { version = "1", features = ["full"] }
edr-common = { path = "../edr-common" }
```

- La librairie `aya` nous permet d'interagir avec notre programme eBPF
- La librairie `tokio` nous permet de gérer l'asynchrone.

2.3.3 edr-common

```
[package]
name = "edr-common"
version = "0.1.0"
edition = "2024"

[dependencies]
```

Celui-ci reste vide pour l'instant.

2.4 Première compilation

1. il faut installer la toolchain nightly (requise par aya-ebpf)
2. installer bpf-linker

```
rustup toolchain install nightly --component rust-src
cargo install bpf-linker
```

Pour tester un build vers une target ebpf nous pouvons mettre ce code dans `edr-ebpf/src/lib.rs`

```
#![no_std]

#[panic_handler]
fn panic(_info: &core::panic::PanicInfo) -> ! {
    loop {}
}
```

1. Pas de librairie standard car elel n'existe pas dans le kernel.
2. Pas `panic_handler` obligatoire pour Aya.

```
cd edr-ebpf
cargo +nightly build
```

Nous pouvons vérifier que nous avons bien un binaire ebpf comme ceci:

```
file target/bpfel-unknown-none/debug/libedr_ebpf.so

ELF 64-bit LSB relocatable, eBPF, version 1 (SYSV), not stripped
```

3 Développement du binaire ebpf

Afin de capturer chaque exécution du syscall `execve`, nous allons devoir nous attacher à un tracepoint kernel ce qui permettra qu'a chaque execve exécuté, une fonction de notre bytecode eBPF s'executera (elle envera la structure du execve dans le userland pour la traiter et la remonté si besoin) .

```
sudo ls /sys/kernel/debug/tracing/events/syscalls/ | grep execve

sys_enter_execve
sys_enter_execveat
sys_exit_execve
sys_exit_execveat
```

Le tracepoint qui nous interesse est donc le `sys_enter_execve`.

Dans notre programme, nous pouvons donc d'abors créer une fonction pour récupérer les évènements `execve`.

```
#![no_std]
#![no_main]
#![feature(asm_experimental_arch)]

use aya_ebpf::{
    EbpfContext,
    macros::tracepoint,
    macros::map,
    maps::RingBuf,
    programs::TracePointContext,
    helpers::bpf_probe_read_user_str_bytes,
};
use edr_common::ExecveEvent;

#[map]
static EVENTS: RingBuf = RingBuf::with_byte_size(256 * 1024, 0);

#[tracepoint]
pub fn edr_execve(ctx: TracePointContext) -> u32
{
    match try_edr_execve(&ctx) {
        Ok(_) => 0,
        Err(_) => 1,
    }
}

fn try_edr_execve(ctx: &TracePointContext) -> Result<(), i64>
{
    // ...
    Ok(())
}
```

- `#![no_std]` et `#![no_main]` car nous n'avons pas de librairie standard ni de `main` dans le kernel.
- `#![feature(asm_experimental_arch)]` est requis pour le `panic_handler` que nous verrons plus tard.
- La macro `#[map]` déclare la map eBPF : le kernel peut y écrire, le user-land peut la lire.
- Nous séparons la logique dans `try_edr_execve` pour pouvoir retourner un `Result` et gérer les erreurs proprement, la fonction `edr_execve` étant uniquement le point d'entrée eBPF qui retourne un `u32`.
- `ctx` est la donnée que `sys_enter_execve` nous fournira et que nous parserons.

Nous avons ensuite besoin d'une structure pour cet évènement. écrivons là dans `edr-common/src/lib.rs`

```
#![no_std]

#[repr(C)]
#[derive(Clone, Copy)]
pub struct ExecveEvent {
    pub pid: u32,
    pub filename: [u8; 256],
}
```

Note

- Vu que le kernel ne nous permet pas d'avoir un type `String`, nous devons faire un tableau de `u8` de taille fixe.
- Vu que cette structure est dans un ring buffer, il doit avoir le trait Clone et Copy.

Voici le format du tracepoint `execve` dans le kernel

```
sudo cat /sys/kernel/debug/tracing/events/syscalls/sys_enter_execve/format

name: sys_enter_execve
ID: 864
format:
        field:unsigned short common_type;       offset:0;       size:2; signed:0;
        field:unsigned char common_flags;       offset:2;       size:1; signed:0;
        field:unsigned char common_preempt_count;       offset:3;       size:1;
signed:0;
        field:int common_pid;   offset:4;       size:4; signed:1;

        field:int __syscall_nr; offset:8;       size:4; signed:1;
        field:const char * filename;    offset:16;      size:8; signed:0;
        field:const char *const * argv; offset:24;      size:8; signed:0;
        field:const char *const * envp; offset:32;      size:8; signed:0;
        ...
```

Ce qui nous interesse sont:

1. **pid**: offset 4
2. **filename**: offset 16

Voici l'implémentation complète de `try_edr_execve` :

Dans un premier temps, nous réservons une zone dans le ring buffer de la taille d'`ExecveEvent` :

```
let mut entry = EVENTS.reserve::<ExecveEvent>(0).ok_or(1i64)?;
// pointeur vers la struct ExecveEvent en ring buffer.
let event_ptr = entry.as_mut_ptr();
```

Nous récupérons ensuite le PID du process et le pointeur vers le `filename`. Si la lecture échoue, nous devons libérer l'entrée réservée via `entry.discard(0)` avant de retourner l'erreur, sinon le slot en ring buffer resterait bloqué :

```
let pid = ctx.pid();

// 16 = offset du filename dans la structure sys_enter_execve.
let filename_ptr = match unsafe { ctx.read_at::<u64>(16) } {
    Ok(ptr) => ptr as *const u8,
    Err(e) => {
        entry.discard(0);
        return Err(e);
    }
};
```

On copie ensuite le pid et le filename dans notre structure en ring buffer. `bpf_probe_read_user_str_bytes` lit une chaîne depuis l'espace utilisateur (le pointeur `filename` est un pointeur user-space) et la copie dans notre buffer kernel :

```
unsafe {
    (*event_ptr).pid = pid;
    if bpf_probe_read_user_str_bytes(filename_ptr, &mut
(*event_ptr).filename).is_err() {
        entry.discard(0);
        return Err(1);
    }
}
```

Une fois l'entrée remplie, on la soumet au ring buffer pour qu'elle soit visible du user-land :

```
entry.submit(0);
Ok(())
```

3.0.1 Panic handler

Le kernel n'a pas de mécanisme de panic Rust standard. Nous devons donc fournir un `panic_handler` personnalisé. Ici on sort simplement du programme eBPF via une instruction assembleur :

```
#[panic_handler]
fn panic(_info: &core::panic::PanicInfo) -> ! {
    unsafe {
        core::arch::asm!("exit", options(noreturn))
    }
}
```

C'est pour cela que `#![feature(asm_experimental_arch)]` est nécessaire en tête de fichier.

4 Developpement de la partie user-land

Voici la base pour notre user-land asynchrone:

```
use tokio::signal;

#[tokio::main]
async fn main() -> Result<(), anyhow::Error>
{
    signal::ctrl_c().await?;
    Ok(())
}
```

4.1 Chargement de l'ebpf compilé

La première étape pour notre programme user-land est de charger notre bytecode eBPF précédemment compilé dans le kernel. Une methode idiomatique utilisant aya serait plus aproprié pour ca avec la fonction `Ebpf::load()`.

```
use aya::Ebpf;

let mut bpf = Ebpf::load(include_bytes!("../../target/bpfel-unknown-none/debug/
libedr_ebpf.so"))?;
```

Nous récupérons directement les bytecode et les chargeons dans le kernel.

4.2 Attacher le Tracepoint

On fait maintenant en sorte que chaque syscall `execve` déclanche notre fonction `edr_execve`.

```
let program: &mut TracePoint =
bpf.program_mut("edr_execve").unwrap().try_into()?;
program.load()?;
program.attach("syscalls", "sys_enter_execve")?;
```

4.3 Lecture du ring buffer

Aya ne fournit pas directement un ring buffer asynchrone prêt à l'emploi. On utilise donc la combinaison `RingBuf` + `AsyncFd` de tokio. `AsyncFd` permet d'attendre de façon asynchrone que le file descriptor du ring buffer soit lisible sans bloquer le thread :

```
use aya::maps::RingBuf;
use tokio::io::unix::AsyncFd;

let ring = RingBuf::try_from(bpf.map_mut("EVENTS").unwrap())?;
let mut async_fd = AsyncFd::new(ring)?;
```

4.4 Boucle d'événements

On peut ensuite faire une boucle qui attend soit un `Ctrl+C` soit que le ring buffer soit prêt à lire. On utilise `tokio::select!` pour ça :

```
loop {
    tokio::select! {
        _ = signal::ctrl_c() => break,
        result = async_fd.readable_mut() => {
            let mut guard = result?;
            let rb = guard.get_inner_mut();
            while let Some(item) = rb.next() {
                let event = unsafe { &*(item.as_ptr() as *const ExecveEvent) };
                let filename = std::str::from_utf8(&event.filename)
                    .unwrap_or("?")
                    .trim_end_matches('\0');
                println!("execve: pid={} filename={}", event.pid, filename);
            }
            guard.clear_recvy();
        }
    }
}
```

- `async_fd.readable_mut()` attend que le ring buffer contienne des données.
- `guard.get_inner_mut()` donne accès au `RingBuf`.
- `rb.next()` retourne chaque entrée en attente : on reinterprète les octets bruts en `&ExecveEvent` via un cast.
- Le `filename` est un tableau de `u8` de taille fixe : on le convertit en `&str` UTF-8 et on supprime les octets nuls de fin.
- `guard.clear_ready()` réinitialise le flag de lisibilité pour que `readable_mut()` puisse se déclencher à nouveau au prochain lot d'événements.

> **Note**
>
> Il faut s'assurer que le programme est lancé avec les droits root (ou `CAP_BPF` + `CAP_PERFMON`) pour charger et attacher un programme eBPF dans le kernel.

5 Résultat

Voici le résultat de l'exécution du binaire avec les droits root :

```
sudo ./target/debug/edr
```

```
execve: pid=89585 filename=/usr/bin/ls
execve: pid=89587 filename=/usr/bin/git
execve: pid=89591 filename=/usr/libexec/localsearch-extractor-3
execve: pid=89595 filename=/usr/bin/fedora-third-party
execve: pid=89600 filename=/proc/self/fd/16
execve: pid=89600 filename=/usr/libexec/packagekitd
execve: pid=89609 filename=/usr/bin/pkla-check-authorization
execve: pid=89634 filename=/usr/bin/nmap
execve: pid=89636 filename=/usr/bin/git
execve: pid=89643 filename=/usr/bin/git
execve: pid=89647 filename=/usr/libexec/localsearch-extractor-3
execve: pid=89654 filename=/usr/bin/clear
execve: pid=89662 filename=/usr/bin/pkla-check-authorization
execve: pid=89665 filename=/usr/bin/pkla-check-authorization
^C
```

On observe que l'EDR capture bien en temps réel tous les appels `execve` du système : aussi bien des commandes utilisateur (`ls`, `git`, `nmap`, `clear`) que des processus système (`packagekitd`, `localsearch-extractor-3`, `pkla-check-authorization`, `fedora-third-party`). Le programme s'arrête proprement à la réception du signal `Ctrl+C`.

6 References

1. Le Rust-Book : https://doc.rust-lang.org/stable/book/index.html
2. La libraire Aya : https://aya-rs.dev/index.html
3. Documentation tracepoint : https://docs.rs/aya/latest/aya/programs/trace_point/struct.TracePoint.html