CVE-2022-0847 detected

This CVE uses a vulnerability that relies in a flag set on a pipe if a page if entirely filled.

This flag allows to overwrite the page, even if the document has been opened read only.

Defining a language to represent the graph

The goal is to create a simple and efficient language to represent the graph of conditions. If representing a graph is not hard, expressing the conditions is far more complicated.

Base of the language

The language describes the graph node by node.

```
<NODE_TYPE=PRIMARY|SECONDARY|TRIGGER> <NODE_ID> SATISFIES <CONDITION>
DEPENDS <LIST OF NODE_ID SEPARATED BY COMMA>
```

Conditions

We have to express a intuitive to represent a condition satisfied by a kernel function.

First, the condition depends on one kfunction, that must be specified.

We can enumerate different conditions based on one kfunction call:

- The function has been called more than N times
- The function's caller PID is *PID*
- The arguments of the function satisfy some other conditions

The two first conditions are easy to represent, the last one is challenging.

We can start defining the condition as

```
<KFUNCTION> [CALLED | PID] <VALUE>
```

For the two first conditions. The last one could be something like

```
<KFUNCTION> ARGS ...
```

Argument conditions

Knowing the condition an argument should satisfy means we know the prototype of the kfunction.

We can then assume we know the type and position of each argument.

In eBPF, the arguments are provided as a list, and can't be retrieved by their name. For this reason, knowing the position is necessary

However, the argument can be a pointer or structure, and it may be needed to access one specific field. To dereference a pointer, some kernel or user read have to be explicited with eBPF so such read must be expressed in the condition.

For example, let's take the `task_struct` structure, and assume we would like to get the pid of the parent. According to https://elixir.bootlin.com, `task_struct` has the fields

```
struct task_struct __rcu    *real_parent;
...
pid_t                       pid;
```

Given a `task_struct` *current*, we can access the parent's pid with

```
current->real_parent.pid;
```

The arrows **->** and dots **.** indicate if the field is a pointer or not. This syntax can be used as well in the condition to indicate when to use functions such as `bpf_probe_read_kernel` and `bpf_probe_read_user`.

`current` is just the nth arguments so we can replace it by ARG_N.

We can then define some operators to express the condition, reusing the common names (EQ, NEQ, GT, LE...). The type can be specified before the comparee (UINT8, STR, INT64...).

The condition is then

```
ARG_N->field1.field2 ... fieldn <OPERATOR> <TYPE> <VALUE>
```

We can extend the condition over multiple arguments using logical operators OR, AND, XOR... and parenthesis between each argument condition.

Parse the condition into eBPF function

The condition can be easily parsed, except from the argument conditions. Indeed, in eBPF structure fields cannot be accessed by their name and using arrows. We already solved the dereference issue, but now, having a raw data representing the C structure, we need to extract the interresting values for RUST.

We can use the *aya_tool* to convert C structures into Rust ones. However, the structure fields isn't as explicit as C one.

Example: the `sock_common` structure conversion

```
pub struct sock_common {
    pub __bindgen_anon_1: sock_common__bindgen_ty_1,
    pub __bindgen_anon_2: sock_common__bindgen_ty_2,
    pub __bindgen_anon_3: sock_common__bindgen_ty_3,
    pub skc_family: ::aya_bpf::cty::c_ushort,

    ...

    pub __bindgen_anon_8: sock_common__bindgen_ty_8,
}
#[repr(C)]
#[derive(Copy, Clone)]
pub union sock_common__bindgen_ty_1 {
    pub skc_addrpair: __addrpair,
    pub __bindgen_anon_1: sock_common__bindgen_ty_1__bindgen_ty_1,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sock_common__bindgen_ty_1__bindgen_ty_1 {
    pub skc_daddr: __be32,
    pub skc_rcv_saddr: __be32,
}

...
```

Compared to the C one

```
struct sock_common {
    union {
        __addrpair  skc_addrpair;
        struct {
            __be32  skc_daddr;
            __be32  skc_rcv_saddr;
        };
    };
    union  {
        unsigned int    skc_hash;
        __u16       skc_u16hashes[2];
    };
    /* skc_dport && skc_num must be grouped as well */
    union {
        __portpair  skc_portpair;
        struct {
            __be16  skc_dport;
            __u16   skc_num;
        };
    };

    unsigned short      skc_family;

    ...
    union {
        u32     skc_rxhash;
        u32     skc_window_clamp;
        u32     skc_tw_snd_nxt; /* struct tcp_timewait_sock */
    };
    /* public: */
};
```

Every union is replaced by another struct, making the conversion not straightforward.

I don't see other option that comparing with the C code of the structure in order to get the correct union. Knowing that the desire union is the xth one, it can be then matched with Rust.

For now, some basic examples will be made manually, but we can imagine another algorithm to realise this matching.