HID-BPF

Benjamin Tissoires
Red Hat

benjamin.tissoires@{redhat|gmail}.com

Kernel Recipes, Paris, June 1-3, 2022

Foreword

- still a WIP (v5 is the latest)
- API mostly designed but still missing a few bits

HID-BPF == HID+BPF

Agenda

- HID
- BPF
- HID-BPF: why?
- HID-BPF: what?
- HID-BPF: how?

HID, a Plug & Play protocol

HID?

- Human Interface Devices
- Win 95 era protocol for handling plug and play USB devices (mice, keyboards)
  - now Bluetooth, BLE, I2C, Intel/AMD Sensors, (SPI in-progress)
- Most devices nowadays are working with generic drivers

HID report descriptor

- describes the device protocol in a "simple" language (no loops, conditionals, etc...)
- static for each device (in flash)

```
0x05, 0x01,       // Usage Page (Generic Desktop)
0x09, 0x02,       // Usage (Mouse)
0xa1, 0x01,       // Collection (Application)            <-- Application(Mouse)
0x09, 0x01,       //  Usage (Pointer)
0xa1, 0x00,       //  Collection (Physical)              <-- Physical(Pointer)
0x05, 0x09,       //   Usage Page (Button)
0x15, 0x00, 0x25, 0x01, 0x19, 0x01, 0x29, 0x05,  //   Logical Min/Max and Usage Min/Max
0x75, 0x01,       //   Report Size (1)                   <- each usage is 1 bit
0x95, 0x05,       //   Report Count (5)                  <- we got 5 of them
0x81, 0x02,       //   *Input* (Data,Var,Abs)      <--- 5 bits for 5 buttons
0x95, 0x03,       //   Report Count (3)
0x81, 0x01,       //   *Input* (Cnst,Arr,Abs)      <--- 3 bits of padding
0x05, 0x01,       //   Usage Page (Generic Desktop)
0x16, 0x01, 0x80, 0x26, 0xff, 0x7f, //   Logical Min/Max
0x09, 0x30,       //   Usage (X)
0x09, 0x31,       //   Usage (Y)
0x75, 0x10,       //   Report Size (16)
0x95, 0x02,       //   Report Count (2)
0x81, 0x06,       //   *Input* (Data,Var,Rel)      <--- X,Y of 16 bits
0x15, 0x81, 0x25, 0x7f,   //   Logical Min/Max (-127,127)
0x09, 0x38,       //   Usage (Wheel)
```

Documentation

- Device Class Definition
- HID Usage Tables

Device Class Definition

https://www.usb.org/document-library/device-class-definition-hid-111

- there are the equivalent files for I2C, Bluetooth, BLE, SPI
- last update: May 27, 2001
- defines generic protocol that every HID device must speak
  - operational model
  - descriptors (USB + HID report descriptor)
  - parser of report descriptors
  - requests
  - report protocol

The protocol is somewhat stable.

HID Usage Tables

https://www.usb.org/document-library/hid-usage-tables-13

- last update: April 5, 2021
- defines *meaning* of usages as defined in the report descriptor
  - X and Y are defined in the Generic Desktop page (0x01) as 0x30 and 0x31
- can be extended (and is) by companies
  - multitouch protocol
  - USI pens
  - HW sensors
- except for a few exceptions: an update means a new `#define` in the kernel if we care

HID

- Most devices nowadays are working with generic drivers

Except for a few of them that need:

- a fixup in the report descriptor (45 drivers out of 82)
  - `hid-sigmamicro.c` in v5.17
- 41 files are under 100 LoC (counted with cloc)
- some driver just change the input mapping (i.e. to enable a given key)
  - `hid-razer` in v5.17

After attending a few Kernel Recipes edition:

"Can eBPF help?"

BPF?

See Alexei's presentation tomorrow

https://www.kernel.org/doc/html/latest/bpf/index.html

https://docs.cilium.io/en/latest/bpf/

> BPF is a highly flexible and efficient virtual machine-like construct in the Linux kernel allowing to execute bytecode at various hook points in a **safe** manner. It is used in a number of Linux kernel subsystems, most prominently ~~networking~~ HID*, tracing and security (e.g. sandboxing).

Allows to add safe kernel space code from the user space (with root access).

* Changed by me :)

HID+BPF

Use BPF in HID drivers to have user-space drivers fixes in the kernel

HID-BPF: base principles

- works only on **arrays of bytes** and talks HID
  - no access to input, or any other subsystems (LEDs, force feedback, ...)
- any *smart* processing needs to be done in userspace or at programming time:
  - parse HID report descriptor
  - compute location of various fields
- targets a specific device for a given program
- enforces GPL programs
  - simple fixes should be shipped in-tree
- programs needs to be CORE (like)
  - users should not be required to have LLVM

HID-BPF: why?

- more convenient to do simple fix and user testing
- HID firewall
- change the device based on the user context
- tracing

HID-BPF: why?

- **more convenient to do simple fix and user testing**
- HID firewall
- change the device based on the user context
- tracing

HID: what it means to add a new quirk?

Device *x* is somewhat broken: a key is not properly reported:

- identification of the issue
- new patch created + tests
- user needs to recompile the kernel
- submission on the LKML
- review of the patch
- inclusion in branch:
  - either scheduled for this cycle
  - either for the next (if big changes, like new driver)
- patch goes into Linus' tree
- kernel marked stable or patch backported in stable
- distributions take the new kernel
- user can drop the custom kernel build

HID: Adding a new quirk with BPF

Device *x* is somewhat broken: a key is not properly reported:

- identification of the issue
- new ~~patch~~ *BPF program* created + tests
- user ~~needs to recompile the kernel~~ drops the bpf program into the filesystem

```
1   SEC("fmod_ret/hid_bpf_rdesc_fixup")
2   int BPF_PROG(rdesc_fixup, struct hid_bpf_ctx *hid_ctx)
3   {
4         __u8 *data = hid_bpf_get_data(hid_ctx, 0, 4096 /* size */);
5
6         /* Convert Input item from Const into Var */
7         data[40] = 0x02;
8
9         return 0;
10  }
```

`data` contains the report descriptor of the device.

`hid_bpf_rdesc_fixup()` is executed once, once the device is exported to userspace.

HID: Adding a new quirk with BPF

Device *x* is somewhat broken: a key is not properly reported:

- identification of the issue
- new ~~patch~~ *BPF program* created + tests
- user ~~needs to recompile the kernel~~ drops the bpf program into the filesystem

User implication stops here once the BPF program is accepted.

Developers continue to *include and ship* the fix in the kernel:

- submission on the LKML
- review of the patch with *the bpf program*
- inclusion in branch
- patch goes into Linus' tree
- kernel marked stable or patch backported in stable
- distributions take the new kernel

HID-BPF: why?

- more convenient to do simple fix and user testing
- **HID firewall**
  - Steam opens up game controllers to the world (with `uaccess`)
  - SDL is happy with that
  - What prevents a Chrome plugin to initiate a controller firmware upgrade over the network?
- change the device based on the user context
- tracing

HID-BPF: why?

- more convenient to do simple fix and user testing
- HID firewall
  - Steam opens up game controllers to the world (with `uaccess`)
  - SDL is happy with that
  - What prevents a Chrome plugin to initiate a controller firmware upgrade over the network?
- **change the device based on the user context**
  - Microsoft Surface Dial example
- tracing

HID-BPF: why?

- more convenient to do simple fix and user testing
- HID firewall
  - Steam opens up game controllers to the world (with `uaccess`)
  - SDL is happy with that
  - What prevents a Chrome plugin to initiate a controller firmware upgrade over the network?
- change the device based on the user context
  - Microsoft Surface Dial example
- **tracing**
  - hidraw is good, but not enough
  - we can trace external requests with eBPF

HID-BPF: what?

HID-BPF: the net-like capability

Change the incoming data flow

BPF program, compiled by clang:

```
SEC("fmod_ret/hid_bpf_device_event")
int BPF_PROG(invert_x, struct hid_bpf_ctx *hid_ctx)
{
	__s16 *x = (__s16*)hid_bpf_get_data(hid_ctx, 1 /* offset */, 2 /* size */);

	/* invert X coordinate */
	*x *= -1;

	return 0;
}
```

Yes, this is a *tracing* BPF program.

Note: this is executed *before* `hidraw` or any driver processing.

HID-BPF: attach our program to a device

A program is attached to a `struct hid_device` in the kernel, by using the system unique id to attach to it (to be triggered by udev):

```
struct attach_prog_args {
      int prog_fd;
      unsigned int hid;
      unsigned int flags;
      int retval;
};

SEC("syscall")
int attach_prog(struct attach_prog_args *ctx)
{
      ctx->retval = hid_bpf_attach_prog(ctx->hid,
                                        ctx->prog_fd,
                                        ctx->flags);
      return 0;
}
```

```
sudo ./hid_mouse /sys/bus/hid/devices/0018:06CB:CD7A.000A
```

HID-BPF: Load more than 1 program for `device_event`

```c
SEC("fmod_ret/hid_bpf_device_event")
int BPF_PROG(invert_x, struct hid_bpf_ctx *hid_ctx)
{
	__s16 *x = (__s16*)hid_bpf_get_data(hid_ctx, 1 /* offset */, 2 /* size */);

	/* invert X coordinate */
	*x *= -1;

	return 0;
}

SEC("fmod_ret/hid_bpf_device_event")
int BPF_PROG(invert_y, struct hid_bpf_ctx *hid_ctx)
{
	__s16 *y = (__s16*)hid_bpf_get_data(hid_ctx, 3 /* offset */, 2 /* size */);

	/* invert Y coordinate */
	*y *= -1;

	return 0;
}
```

HID-BPF: `device_event`

Benefits/Use cases:

- Useful for neutral zone of a joystick
- Filter out unwanted fields in a stream
- Fix the report when something should not happen

HID-BPF: changing how the device looks and talks

```c
SEC("fmod_ret/hid_bpf_rdesc_fixup")
int BPF_PROG(rdesc_fixup, struct hid_bpf_ctx *hid_ctx)
{
	__u8 *data = hid_bpf_get_data(hid_ctx, 0, 4096 /* size */);

	/* invert X and Y definitions in the event stream interpretation */
	data[39] = 0x31;
	data[41] = 0x30;

	return 0;
}
```

`data` now contains the report descriptor of the device.

(Un)attaching this program triggers a disconnect/reconnect of the device.

Only 1 program of this type per HID device.

HID-BPF: `rdesc_fixup`

Benefits/Use cases:

- Fix a bogus report descriptor (key not properly mapped)
- Morph a device into something else (Surface Dial into a mouse)
- Change the device language (in conjunction with `device_event`)

HID-BPF: communicate with the device

```c
struct hid_send_haptics_args {
      /* data needs to come at offset 0 so we can use ctx as an argument */
      __u8 data[10];
      unsigned int hid;
};

SEC("syscall")
int send_haptic(struct hid_send_haptics_args *args)
{
        struct hid_bpf_ctx *ctx;
        int i, ret = 0;

        ctx = hid_bpf_allocate_context(args->hid);
        if (!ctx)
                return -1; /* EPERM check */

        ret = hid_bpf_hw_request(ctx, args->data, 10, HID_FEATURE_REPORT,
                                 HID_REQ_GET_REPORT);
        args->retval = ret;

        hid_bpf_release_context(ctx);

        return 0;
}
```

HID-BPF: communicate with the device

`hid_bpf_hw_request()`

Same behavior than the in-kernel function `hid_hw_raw_request()`.

Can not be used in interrupt context.

Allows:

- query device information
- put the device into a specific mode

HID-BPF: how?

Architecture

HID-BPF is built on top of BPF, but outside of it:

Existing BPF features:

- relies on `ALLOW_ERROR_INJECTION` API to add tracepoints
- relies on kfunc API for HID-BPF custom BPF API

Missing BPF features (addressed in the patch series):

- custom implementation for attaching to a given HID device
- (couple of BPF-core changes for accessing arrays of bytes)

`ALLOW_ERROR_INJECTION`

- Introduce a tracepoint in kernel code that can be tweaked by eBPF
- Introduced by programmer at a given place in the code

Define a tracepoint with side effect

in the kernel module itself:

```
__weak noinline int
my_tracepoint(struct my_kfunc_data *data)
      return 0;
}
ALLOW_ERROR_INJECTION(my_tracepoint, ERRNO);

int
regular_processing_fn(struct my_kfunc_data *data)
{
      int ret;

      ret = my_tracepoint(data)
      if (ret)
            return ret;

      /* do some other normal processing */

      return 0;
}
```

in the eBPF program:

```
SEC("fmod_ret/my_tracepoint")
int BPF_PROG(tracepoint_fixup,
             struct my_kfunc_data *data)
{
      if (something)
            return -1;

      return 0;
}
```

Kfuncs

- export a kernel function as eBPF dynamic API
  - no need to update libbpf
- care needs to be taken (it's like a syscall in the end), but eBPF takes all of the cumbersome part away:
  - argument checking
  - availability of the call
  - *versioning*

KFuncs? 1/2

in the module itself:

```
noinline int my_kfunc(struct my_kfunc_data *ctx) {
      return ctx->a + ctx->b;
}

BTF_SET_START(my_kfunc_ids)
BTF_ID(func, my_kfunc)
BTF_SET_END(hid_bpf_kfunc_ids)

static const struct btf_kfunc_id_set my_kfunc_set = {
      .owner         = THIS_MODULE,
      .check_set     = &hid_bpf_kfunc_ids,
};

int __init my_module_init(void)
{
      return register_btf_kfunc_id_set(BPF_PROG_TYPE_TRACING, &my_kfunc_set);
}

late_initcall(my_module_init);
```

KFuncs? 2/2

in the BPF program:

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

char _license[] SEC("license") = "GPL";

extern int my_kfunc(struct my_kfunc_data *ctx) __ksym;

SEC("fentry/another_function")
int BPF_PROG(bpf_something, struct my_kfunc_data *data)
{
      return my_kfunc(data);
}
```

Wrap-up

HID-BPF: Summary

- should simplify easy fixes in the future
- allow to add user-space defined behavior depending on the context
- can add traces in the events
- will allow to live-fix devices without having to update the kernel
- no more custom kernel API (sysfs, module parameters)
- will **not** replace in-kernel drivers for devices broken at boot time (keyboards) or for devices that need an actual driver (hid-rmi.ko)

END

HID-BPF: Summary

- should simplify easy fixes in the future
- allow to add user-space defined behavior depending on the context
- can add traces in the events
- will allow to live-fix devices without having to update the kernel
- no more custom kernel API (sysfs, module parameters)
- will **not** replace in-kernel drivers for devices broken at boot time (keyboards) or for devices that need an actual driver (hid-rmi.ko)

Extra slides

Current patch series (v5)

- couple of BPF core refinments need merging/discussion:
  - extend kfunc to return read/write char buffers
  - extend BPF map kernel API
- HID-BPF built outside of BPF-core
  - use of tracing BPF programs
  - API built using eBPF kfuncs (kernel functions called from BPF programs)
  - handling of dispatcher fully in HID-BPF thanks to a preloaded BPF program
- access to data through `` `hid_bpf_get_data()` ``
- `` `SEC("fmod_ret/hid_bpf_device_event")` `` done IMO
- `` `SEC("fmod_ret/hid_bpf_rdesc_fixup")` `` done IMO
- `` `SEC("syscall")` `` probably needs more HID kfuncs

HID-BPF: future

- finish various entrypoints to be able to handle all use cases
  - `SEC("fmod_ret/hid_bpf_request") called when a request is emitted to the device
  - `SEC("fmod_ret/hid_bpf_resume")
  - ...
  - to implement firewall-like capabilities
- might need a `hid_bpf_inject_event()` at some point
  - useful for macro keys
- add autoloading mechanism of in-kernel BPF programs
  - just drop the bpf source in the tree and it gets automagically included in a new module

HIDRAW: Other implementation of ioctl `HIDRAW_EVIOC_REVOKE`

- initial (non-BPF) patch submitted on LKML:
  - https://lore.kernel.org/linux-input/YmEAPZKDisM2HAsG@quokka/
- suggestion to use `ALLOW_ERROR_INJECTION`
- logind can revoke **any** hidraw fd without code change
- https://gitlab.freedesktop.org/bentiss/logind-hidraw

Something similar for USB devices is in the work:

- https://lore.kernel.org/linux-usb/20220425132315.924477-1-hadess@hadess.net/