Running Python in the Linux Kernel - Yonatan Goldschmidt - Medium

Yonatan Goldschmidt

Y

This article will talk about a cool project I've worked on recently — a full Python interpreter running inside the Linux kernel, allowing:

- Seamless kernel APIs calls
- Global variables access
- Python syntax sugar for reading & writing kernel structures
- Kernel-core-into-Python callbacks ("Python function pointers")
- Kernel function hooking (via kprobes & ftrace)
- Python-based kernel threads

And everything's dynamic, with a REPL running in the kernel.

If you just wanna try it, then you can jump to the end of the article for usage instructions. In the rest of this post I'll share how this idea came about, what were the challenges building it, etc.

But first, I'll give you a few examples of what it's capable of.

```
>>> printk("so.. %s %d %d %d\n", "hello", 123, None, True)
18
>>> # in dmesg: "so.. hello 123 0 1"
```

This small snippet will print all files opened on the system (by all processes and by the kernel itself):

```
# kernel_ffi has functions to interop with the kernel
from kernel_ffi import ftrace, str as s# filename_s is now a function that "casts" pointers to the kernel's "struct filename"
filename_s = partial_struct("filename")# "orig" is a callable pointer, to call the real do_filp_open.
def do_filp_open_hook(orig, dfd, pathname, op):
    fn = filename_s(pathname)
    # fn is a "struct filename" object, and we can access its fields
    # use "s" to read the name pointer to a Python string
    fn_str = s(int(fn.name))
    print("do_filp_open on {!r}".format(fn_str))
    # finally, call the original with the same arguments. we
    # could modify them if we wanted.
    return orig(dfd, pathname, op)ft = ftrace("do_filp_open", do_filp_open_hook)# remove when you're done. if you forget, it'll be removed when the
# object is garbage-collected.
ft.rm()
```

This snippet will "change" the contents of `/dev/null` :

```
file_operations = partial_struct("file_operations")
# I can reference null_fops without previously defining it -
# all missing globals are resolved using the kernel symbols.
null_fops = file_operations(null_fops)from kernel_ffi import callbackdef my_read_null(file, buf, count, ppos):
    pos = p64(ppos)
    b = b"who said /dev/null must be empty?\n"[pos:]
    l = min(len(b), count)
    memcpy(buf, b, l)
    p64(ppos, pos + l)
    return lc = callback(my_read_null)
# calls to null_fops.read will call our callback instead.
null_fops.read = c.ptr()
```

Why?

I've had experience developing both user-mode and kernel-mode software. User-mode environments have a great advantage when we talk about the ease of development; The development community is huge, we have tons of examples online, and many dev tools are ready at hand. You also have many tools and scripting languages to help you with prototyping. Even when your code-base is written in a low level language like C, you can experiment with user-mode ideas in a faster prototyping environment, like Python.

You don't get that in kernel development. There are tons of APIs, much less documentation and no feasible way of prototyping besides recompiling your code, reloading it and trying to measure (let's face it, we all just `printk`) the differences. Needless to say that you pay badly for mistaking, since many of them will crash your kernel.

I got the idea that being able to *easily* prototype API calls, access variables and monitor kernel functions behavior, might be useful. A more dynamic language will do well, and a REPL will be very nice.

There are tons[1] of dynamic-patching kernel tools, but I didn't know of one providing a dynamic REPL, and in a convenient manner. And what would be more convenient than Python?

Basically what I had in mind is a fusion of [Frida](https://frida.re) (providing hooks, because I realized hooks will also be useful) and a Python REPL (providing easy inspection and interaction with objects and functions).

Here's a real-life example from this week, showing this need of a REPL: I wrote something based on the kernel's `rw_semaphore`, in one function I had the read lock taken and I needed to write. The concept of "upgradable RW locks" — *atomically converting a read lock to a writer lock* — is well known, so I thought a quick search online would yield a result as for whether it's possible or not. I couldn't find anything, neither in the docs (the .h file).

A quick check in the REPL would be enough! So I headed to the terminal tab I keep open with a REPL on some QEMU VM, and typed the following:

```
>>> from kernel_ffi import kmalloc
>>> from struct_access import sizeof
>>>
>>> p = kmalloc(sizeof("rw_semaphore"))
>>> __init_rwsem(p)
>>>
>>> down_read(p)
>>> # yeah, can lock twice
>>> down_read(p)
>>>
>>> down_write_trylock(p)
0  # makes sense, i guess
>>> down_write(p)
# hangs forever!
```

Back to the story. I've had some experience with [MicroPython](https://micropython.org), which is a complete Python 3 interpreter intended for microcontrollers. I've been using it on various chips like the ESP32 and ESP8266 for a while, and I decided it won't be too troublesome to port it to the Linux kernel, since unlike CPython, MicroPython wasn't designed to be run (only) in usermode. It wasn't designed to run in the Linux kernel either, but it makes much less assumptions about its environment, and that's why it is easier to get it running on a new "platform".

Some challenges

Some issues and designs I have encountered during development.

Struct Access

The kernel code defines and uses thousands of complex structures. I wanted this tool to provide human-friendly struct accessing, not address-based like "read an integer at address XX", "write a byte at address YY".

So, for a struct like `struct net_device *dev`, what's required to provide accessors like `dev->stats.rx_dropped`?

The first thing is some Python syntax magic for the dereferences, array accesses, etc. It's quite cool behind the scenes, but I won't talk about that.

The second thing, and the more interesting one, is how does the Python get to know the structure layout?

When compiling a kernel module (that uses structures), you are required to have the kernel headers and configuration. The compiler reads the definitions from them. Can we parse those headers and extract structure definitions as well?

There are a few C structure parsing libraries in Python, for example [cstruct](#) and [dissect.cstruct](#). But if you have had a look at a complex kernel structure, you'd know these approaches won't "just work" — the struct definitions make very extensive use of `#ifdef`s based on configurations, specific alignment requirements like `__cacheline_aligned_in_smp`,[2] not to mention `__randomize_layout` ...[3] My point is, it will be very hard to parse it correctly for someone who's not the compiler actually building it.

Who's else doing it, though? Debuggers. GDB allows you to print structs and access structs field. How do debuggers do it?

The DWARF debugging format (embedded in ELF files when you compile with `-g`) can encode struct definitions. That's what debuggers use, and also how the useful [pahole](#) tool does its tricks. There's even also [a Python](#) implementation that does more or less the same.

*At this point, I thought that relying of DWARF requires the target **code** to be recompiled with debug info. Now I can say it's true. Anyway, I went ahead with the solution I had mind…*

I decided I had to follow the way GCC extracts structs, and what's the best way to do so if not from the inside of GCC itself? Time to write a GCC plugin.

So I found a [simple example of a GCC plugin](#) and did *what developers do the best* — take existing code they don't fully understand, run it, modify it a little, run it again and learn from what's changed. Shortly after I had a fully functioning plugin that you can load during your compilation, and it'll dump all defined structs in as nice Pythonic objects. I named it struct_layout (just like the DRAWF parsing Python project I mentioned earlier) and you can find it [on github](#).

So all you have to do is to include the headers you want into an empty C file, and compile with the plugin.

In hindsight, I could do similarly with DWARF… ;) Just compile an empty C file including the relevant headers, and read the dwarf output in the `.o` file.

But I'm happy with my choice. Writing a GCC plugin was a great experience I encourage all developers to take on themselves. It was sure more fun than parsing the DWARF section.

The rest was simple. As I said earlier, with a bit of Python magic you can easily wrap those definitions with objects that'll allow the nice syntax sugar mimicking how it looks in C code.

```
>>> d = net_device(dev_get_by_name(init_net, "eth0"))
>>> # so this accesses the net_device's
>>> # "struct net_device_stats stats" field, and
>>> # then accesses "unsigned long rx_bytes"
>>> d.stats.rx_bytes
15158
```

And since it's Python and everything's dynamic, it also comes with a few cool features, like tab completion for fields and basic protection from NULL dereferences, array out-of-bounds access, unsigned/signed overflow detection and more. Neat!

Multi-threading

MicroPython supports multi-threading out-of-the-box, you just have to give it some basic primitives (like locks) and a TLS (thread-local storage), and you need to hack a GC implementation that takes all threads into account.

TLS implementations in usermode vary. For `x86`, for example, you can `clone` a new thread with `CLONE_SETTLS` and the kernel will make sure that whenever your thread runs, it'll have the `fs` register pointing to your specified data structure. Other basic possible implementation can be to use the bottom/top of the stack for storage. That's the kernel implementation for `current_thread_info` in many architectures — for example, for ARM:

```
static inline struct thread_info *current_thread_info(void)
{
    return (struct thread_info *) (current_stack_pointer & ~(THREAD_SIZE - 1));
}
```

These methods are useful when you control the thread — that is, you create it, you control the entry point, and so on. But what if you're a callback in another thread, or just hooking onto some code which is to be run by threads you don't control? You can't use these tricks — you might interfere with the owner of the thread who's using the same method.

So, you can just select one of the unused methods (`x86` doesn't use the top of the stack — or the bottom, I can't recall — so it's free for your use) but since I needed to maintain the list of threads currently executing Python (we'll see later why) I went with a more naive implementation — making use of the existing thread-based struct `current`, I keep a list of `task_struct` s to their TLS info. Then you can always get the TLS for `current`.

The next major component that needs special treatment when multi-threading is the garbage collector— MicroPython uses a mark & sweep garbage collector:

In mark & sweep, no object references are maintained, and when the GC runs it needs to scan the entire program memory (or, at least the relevant areas) for pointers into the heap. Heap blocks that are referenced are marked, and are added to the scan queue, recursively. All blocks still unmarked when the scan queue is empty are considered garbage and are freed.

Mark & sweep has the advantage of simpler code (no reference counting, no freeing, you just `malloc` stuff and everything works). But the requirement to "scan the entire program memory" is… impossible in the kernel. For example, the Python makes partial of the kernel heap. Do we need to scan the heap as well? It might be huge!

But we just need to make sure all the pointers into the heap are scanned… If we never place root pointers — pointers that no other object refers to — globally, but only on stacks of threads executing Python, then this scan can be narrowed down to the stacks of these threads, which is simple enough to do.

Using the list of threads we kept for TLS, we can get the stack start & end pointers of the relevant threads. The stack start can be narrowed to the "stack start when Python started" — if there's a kernel stack containing 10 frames, and then a Python hook was called, we don't need to scan those 10 previous frames. Care must be taken to use the right stack start, even for a thread with nested Python calls (that is, a Python hook calling kernel code that's again hooked by Python). And since there's no feasible way to get the current stack pointer for a thread currently running on another CPU, we'll just use the real stack end.

Combining these, the GC is deterministic enough and safe enough to run, even from kernel hooks, which is quite cool.

Threading support is not complete and I still get crashes from time to time, but it's not something that can't be debugged and eliminated.

And's that's enough talking for this post. Onward to the real thing!

MicroPython for the Linux Kernel

Use it on your own risk! *This is definitely not production ready. Whatever you try using it should be tried using a VM beforehand. And if you need such a dynamic tool for production use, I suggest you to use one of the tools mentioned in the footnotes[1].*

(Though it mostly works and I managed to happily run it on my physical PC… :))

Compiling

```
$ # point KDIR to the kernel headers you're compiling for.
$ # if building for the locally running kernel, you can skip this step.
$ export KDIR=/path/to/the/kernel/
$ # get structs_layout, it'll be built as a part of micropython.
$ git clone https://github.com/Jongy/struct_layout.git
$ # build python:
$ git clone --depth 1 -b linux-kernel https://github.com/Jongy/micropython.git
$ cd micropython
$ make -C mpy-cross
$ make -C ports/linux-kernel STRUCT_LAYOUT=1
```

The resulting module `ports/linux-kernel/build/mpy.ko` can be loaded with a plain `insmod` / `modprobe` .

You can get a REPL with socat file:`tty`,raw,echo=0,escape=0xc tcp:<IP>:9999 (use Ctrl+D/Ctrl+L to break out from this shell).

For more examples and general help, you can refer to the [README](#).

I've tested it on QEMU+KVM with kernels 4.4, 5.0, 5.2. 5.3, 5.4. I've also tested it on my physical laptop (Ubuntu 18.04, kernel 5.0) and my Arch Linux (kernel 5.4).

That's all. I hope you enjoyed the reading, and that you find this tool useful :)

[1]: To name a few: SystemTap, eBPF, perf, kprobe events, ftrace, kplugs (which is somewhat similar to this project…)

[2]: `__cacheline_aligned` attributes ensure data items are aligned to the size of a cacheline. This is usually used when different items in a struct tend to be written concurrently by different CPUs. By ensuring "items of different CPUs" don't cross into the same cacheline, cache flushes can be avoided. A classic example is the `ptr_ring` struct which has separate cachelines for "producer" items, "consumer" items and shared items.

[3]:`__randomize_layout` attributes marks a struct definition as applicable for struct layout randomization, which is a very cool obfuscation technique employed to make kernel exploits harder. Read this lwn article to learn more about it.