Veit's Blog

Discovering ptrace Is Discovering Pain

9/17/2017

One of my latest projects is writing a debugger. How hard can it be, I thought, I'm basically just wrapping system calls. So, to make things a bit more interesting, I decided to support both Linux and OS X. Some of you —the poor souls who've tried writing a debugger yourselves—might know where this is going. I really have no excuse: I knew about the incompatibilities and, like an idiot, decided to go for it anyway. Here is my story.

Elegy to a system call

For those of you who are lucky enough not to know about `ptrace`, let me introduce you: it is the mother of all system calls. The UNIX wizards of the 70s, in their infinite wisdom, decided to put all the facilities for inspecting and debugging a program into one single function call that has about a gazillion possible argument configurations, depending on what you want to do. Want to single-step through a program? Want to get the register contents, or modify them? Want to get or set memory? Want to work with breakpoints? `ptrace` is the answer to all of those questions. Its first argument, the `ptrace_request`, specifies what you want to do, and the arguments that come after that change according to that. Sounds like a job for multiple dedicated functions to me, but who am I to challenge the wisdom that was handed to us by the sages of Bell Labs?

```
uint64_t get_instruction_pointer(pid_
  struct user_regs_struct regs;
  ptrace(PTRACE_GETREGS, pid, NULL, &
  /* rip is the instruction pointer r
  return regs.rip;
}
```

Fig. 1: An example of how to get the current instruction pointer using `ptrace`.

To my surprise, working with `ptrace` actually isn't as painful as I made it sound. After understanding its idiosyncracies, it's actually quite pleasurable. Sure, the names of the requests are different on BSD and Linux, but that's nothing a few cleverly placed `define` statements can't fix, right? At least the rest of the interface is the same.

Enter OS X. Someone—or, more likely, some committee—came to a conclusion that mirrors my complaints from earlier: let's gut this hellish mutation of a system call and split it up into a bunch of different functions instead. Sounds like a good idea at first, until you realize that that means that you have to implement integral parts of your debugger again, and then hide all of that behind several massive `ifdef` statements.

I'd happily pay that price if it meant that the interface was any better. Sadly, this is not the case. Let's consider the example I provided in Figure 1 and reimplement it for OS X. Feel the pain.

```
uint64_t get_instruction_pointer(pid_
  thread_act_port_array_t thread_list
  mach_msg_type_number_t thread_count
  x86_thread_state64_t thread_state;
  task_t port;
  mach_msg_type_number_t sc = x86_THR
  task_for_pid(mach_task_self(), pid,

  task_threads(d->port, &thread_list,


  thread_get_state(thread_list[0], x8
                   (thread_state_t)&t

  return thread_state.__rip;
}
```

Fig. 2: Let's just get this over wi—OH MY GOD, WHAT HAVE I DONE?!

Sorry you had to see that. I'm not going to walk you through that code—mostly because I think noone should have to know such a godawful API. If you're persistent or masochistic enough to want to learn more, I suggest you read this tutorial from Uninformed.

To make matters worse, there is a mind-boggling scarcity of resources regarding this API, and the tutorials that do exist are outdated. The Uninformed article I linked to above talks about PowerPC, for instance, an architecture that is pretty outdated and rare these days. And this is one of the best resources I could find. How did I find out how to change the macros and function calls to the x86 equivalent? I'm so glad you asked: I grepped through `/usr/include` and tried a few wild guesses. My favorite part is that the x in `x86` is the only thing that's not capitalized in the constant. Whoever decided on that naming caused me another minute of hell.

Of course this isn't portable across OS X machines either, because it's architecture-dependent—`x86` has two different structs and constants for 32 and 64 bit, respectively, and then there's also `i386` and `PPC`, but I sincerely hope I'll never have the urge to support those.

So what?

I guess the moral of the story is "be careful what you wish for", because my timid wishes for a bit of modularity just landed me in `ifdef` purgatory. There is something to be learned even in Gehenna, though, and that's something I've written about before: I like API updates. Some APIs need a thorough cleanup, and, much like people who change for the better, emerge as more productive members of society. I even applaud some updates that break my software, because they provide soothing for my code that has been plagued by a sore for no good reason. Sometimes, however, the updates don't lead anywhere; either they don't address a need/make things worse or they introduce incompatibilities that make supporting multiple platforms harder for the user. OS X's API is a prime example for both of these points.

Wish me luck on my next adventures in writing a debugger.

Want to go back to the list of posts?