Using Ptrace For Fun And Profit

Jan 30, 2016

I've been working on a project at work where I am implementing a heap profiler (written in C) for Python. It works in a way that is very similar to the tracemalloc project. The main difference is that my implementation does not require patching the Python interpreter. The tracemalloc project works by changing the definition of `struct PyObject` to embed a pointer to the allocation information. This has the unfortunate effect of changing the Python ABI (e.g. Python wheels that contain compiled C code need to be recompiled). My implementation avoids changing the `struct PyObject` definition and therefore maintains ABI compatibility, but comes at the cost of being slightly less memory/CPU efficient. I wrote briefly about the technique I'm using in an earlier blog post. Hopefully I'll be able to write more about the project later (or even get it open sourced).

The first version of my profiler worked by installing a signal handler for SIGUSR1. When the signal handler ran it would dump all of the heap profiling information to a file in `/tmp`. This works, but it has a couple of limitations:

- The signal handler can be overridden later, e.g. uWSGI installs its own SIGUSR1 handler. You can work around this by trying to find a less commonly used signal to use (e.g. SIGXCPU), but it still felt kind of hacky.
- There's no way to pass any information with a signal. For instance, you can imagine that you might want a way to ask the process to dump the heap information to a particular file path that you supply, or ask it to only dump information about certain types of objects, etc. None of this information can be supplied when sending a signal to a process.

The original implementation worked by overwriting Python's memory allocation routines using `LD_PRELOAD`. I got requests to be able to attach to an already running process and start tracking memory information from that point on. This is something that you can't do with `LD_PRELOAD` since it has to be present at the time the process is started.

Someone advised that I look into the `ptrace()` system call. The ptrace system call is how GDB works on Linux (and other Unix systems). The ptrace system call is also how strace is implemented. The way it works is that the process that calls `ptrace()` (which is referred to as the "tracer") attaches to another process (which is referred to as the "tracee"). The tracer then has essentially unlimited power to control, modify, and alter the tracee. Here is an incomplete list of some of the things the tracer can do:

- read any memory in the tracee's address space
- modify any memory in the tracee's address space
- modify `.text` data (i.e. the actual assembler instructions) in the tracee, even though that area is normally mapped read-only
- read or modify registers of the tracee

Basically this system call would let me do what I want. Using ptrace you can dynamically attach to a process and patch the PLT and GOT to point to your own custom methods which effectively emulates what LD_PRELOAD does. Using ptrace you can also control the tracee to send in information. For instance, you can invoke a memory dump routine and supply that routine with any arguments you want, e.g. a filename, flags, etc.

Black Magic

In theory, this is all pretty straightforward. In pratice, it is anything but.

First of all, to do anything remotely interesting with ptrace you will need to know some assembler (x86 in my case). This is because if you want to run code in the tracee you will need to inject that code into the process. This means at the very minimum you need to know how to either generate a CALL instruction to call a userspace method or SYSENTER to make a system call. In both cases you need to know the calling convention, i.e. what registers need to be set to make a function call (and possibly what you'll need to push onto the stack).

Let's say you want your tracer to call a userspace method in the tracee. This is surprisingly difficult.

To call a method, you need to know it's address. When you're writing your own code that's easy to do:

```
int x;
printf("address of x is %p\n", &x);
printf("address of printf is %p\n", printf);
```

However, you can't do this with the tracee. Normally what happens when you run a program is that any methods that are part of your code are put into a fixed location when the linker links your object code. Methods that are part of shared libraries are linked into your code at runtime by `ld.so` when your process starts up. What literally happens is `ld.so` scans through the object code in your process and looks for all of the references to library calls (e.g. `printf()`) and then modifies the x86 code in the memory space of your process so that all of your `CALL` and `JMP` instructions go to the right place.

In fact, it gets even more complicated. For security reasons Linux implements this thing called ASLR. This means that when a shared library is loaded it's put into a random memory location. **This is done intentionally to make it difficult to find and call arbitrary methods.** It has a good purpose: it's meant to improve the the security of the system, so that attacks like return to libc are more difficult. But it makes it really hard to call methods using ptrace.

There is also really no programmatic way to find and enumerate methods in C. In theory what you can do is parse the ELF data from the binary you're attached to and its shared libraries, but in practice this is so complex that it's not even worth trying.

Tools like `nm`, `objdump`, and gdb actually do this. But for regular mortals it is unreasonably complicated to try to do this at runtime in a real program.

However, not all is lost. If you want to call methods in the tracee that are compiled in as part of its source code (i.e. that are static to the executable) you can disassemble the binary and hardcode those locations into your ptracer program. This is kind of awful because it means that you will have to update the hardcoded constants any time the binary changes (e.g. it's updated or recompiled with new flags), but it does work.

If you want to call methods in shared libraries, I found a hack that lets you figure out how to find the locations of methods. Here's how it works. Let's say you want to call a method defined in libc. For our example we'll consider calling the `fprintf()` method. You look at the file `/proc/<pid>/maps` for the other process, and that file will tell you where the ASLR decided to actually load the library. This will only work if you have the same user id as that process, or if you are root, since you're not allowed to look at the memory mapping for arbitrary processes as an unprivileged user. Then you look at `/proc/self/maps` and again, find where your process decided to load libc. Then what you do is in your own process you take the address of `fprintf()` and subtract from it the address of where libc was loaded in your process. This will give you the number of bytes past the start of libc where `fprintf()` is defined. Let's say, for example, that when you do this you get 20544.

Now that you know htat `fprintf()` is defined 20544 bytes after the start of libc, you can guess that `fprintf()` in the tracee is located 20544 bytes after that process' libc. So using the libc start you found from looking at the proc maps file, you can compute the address of `fprintf()` in the tracee!

You can also use this technique for other arbitrary shared libraries by just linking your tracer against them. For instance, let's say I want my tracer to remotely call a method that's defined by Python, such as `PyObject_Malloc()`. If I link my ptracer using `-lpython2.7` or `-lpython3.4m` then I can use the same trick to find where a running instance of `/usr/bin/python` has `PyObject_Malloc()` loaded.

There is one major caveat here. On Linux you can update shared libraries at any time. Old processes that have already loaded the shared library will continue to have the previous version of the library loaded into memory. So if you try this technique against a process that's been running for a long time (say, weeks or months) it's possible that a library method you want to call might have changed locations if the shared library has been updated since the tracee was started.

Show Me The Code!

I put some code up on GitHub that shows exactly how to employ this technique. The project is at eklitzke/ptrace-call-userspace and if you want to just dive directly into the code look at call_fprintf.c. The codes runs the equivalent of:

```
fprintf(stderr, "instruction pointer = %p\n", rip);
```

in the traced process, where `rip` is the value of the instruction pointer when the process was initially attached by the tracer.

If you look at the code you can see that it's actually pretty short, and overall isn't that complicated. However, it took me a lot of time to actually get this working. You can't use GDB with a program you are ptrace attached to, since only one process is allowed to ptrace at a time. So debugging was more difficult than with a normal C program: you're limited to debugigng core dumps if you crash the tracee. If you make a mistake, you're likely to not just get the pedestrian SIGSEGV, but the much more exciting SIGILL which means that you generated illegal x86 code! I also had effectively zero experience with x86 before starting this project. There's not even a lot of x86 in this project, but I spent a lot of time disassembling other C programs to understand how they work. I also found the whole CALL thing in x86 confusing since the target address is encoded in a weird way. You don't call an absolute address like 0x7fd3edb64000, instead you have to compute the difference between the current value of the instruction pointer, your call target, and then take into account the actual size of the CALL instruction (5 bytes for a rel32 call) when you encode the instruction.

I had a lot of fun writing this and felt like I learned a lot about reading disassembled code. I also have a newfound appreciation for the `objdump` command and all of the amazing things you can do with it, and I learned a lot of fun new GDB commands for looking at register state and frobbing memory and whatnot. I hope other people find this example code useful in their own projects.

Home Email PGP RSS GitHub