Intercepting and modifying Linux system calls with ptrace

notes.eatonphil.com/2023-10-01-intercepting-and-modifying-linux-system-calls-with-ptrace.html

How software fails is interesting. But real-world errors can be infrequent to manifest. Fault injection is a formal-sounding term that just means: trying to explicitly trigger errors in the hopes of discovering bad logic, typically during automated tests.

Jepsen and ChaosMonkey are two famous examples that help to trigger process and network failure. But what about disk and filesystem errors?

A few avenues seem worth investigating:

- A custom FUSE filesystem
- An LD_PRELOAD interception layer
- A ptrace system call interception layer
- A `SECCOMP_RET_TRAP` interception layer
- Or, symbolic analysis a la Alice from University of Wisconsin-Madison

I would like to try out FUSE sometime. But LD_PRELOAD layer only works if IO goes through libc, which won't be the case for all programs. ptrace is something I've wanted to dig into for years since learning about gvisor.

`SECCOMP_RET_TRAP` doesn't have the same high-level guides that ptrace does so maybe I'll dig into it later. And symbolic analysis might be able to detect bad workloads but it also isn't fault injection. Maybe it's the better idea but fault injection just sounds more fun.

So this particular post will cover intercepting system calls (syscalls) using ptrace with code written in Zig. Not because readers will likely write their own code in Zig but because hopefully the Zig code will be easier for you to read and adapt to your language compared to if we had to deal with the verbosity and inconvenience of C.

In the end, we'll be able to intercept and force short (incomplete) writes in a Go, Python, and C program. Emulating a disk that is having an issue completing the write. This is a case that isn't common, but should probably be handled with retries in production code.

This post corresponds roughly to this commit on GitHub.

A bad program

First off, let's write some code for a program that would exhibit a short write. Basically, we write to a file and don't check how many bytes we wrote. This is extremely common code; or at least I've written it often.

```
$ cat test/main.go
package main

import (
        "os"
)

func main() {
        f, err := os.OpenFile("test.txt", os.O_RDWR|os.O_CREATE|os.O_TRUNC, 0755)
        if err != nil {
                panic(err)
        }

        text := "some great stuff"
        _, _ = f.Write([]byte(text))

        _ = f.Close()
}
```

With this code, if the `Write()` call doesn't actually succeed in writing everything, we won't know that. And the file will contain less than all of `some great stuff`.

This logical mistake will happen rarely, if ever, on a normal disk. But it is possible.

Now that we've got an example program in mind, let's see if we can trigger the logic error.

ptrace

ptrace is a somewhat cross-platform layer that allows you to intercept syscalls in a process. You can read and modify memory and registers in the process, when the syscalls starts and before it finishes.

gdb and strace both use ptrace for their magic.

Google's gvisor that powers various serverless runtimes in Google Cloud was also historically based on ptrace (`PTRACE_SYSEMU` specifically, which we won't explore much in this post).

Interestingly though, gvisor switched only this year (2023) to a different default backend for trapping system calls. Based on `SECCOMP_RET_TRAP`.

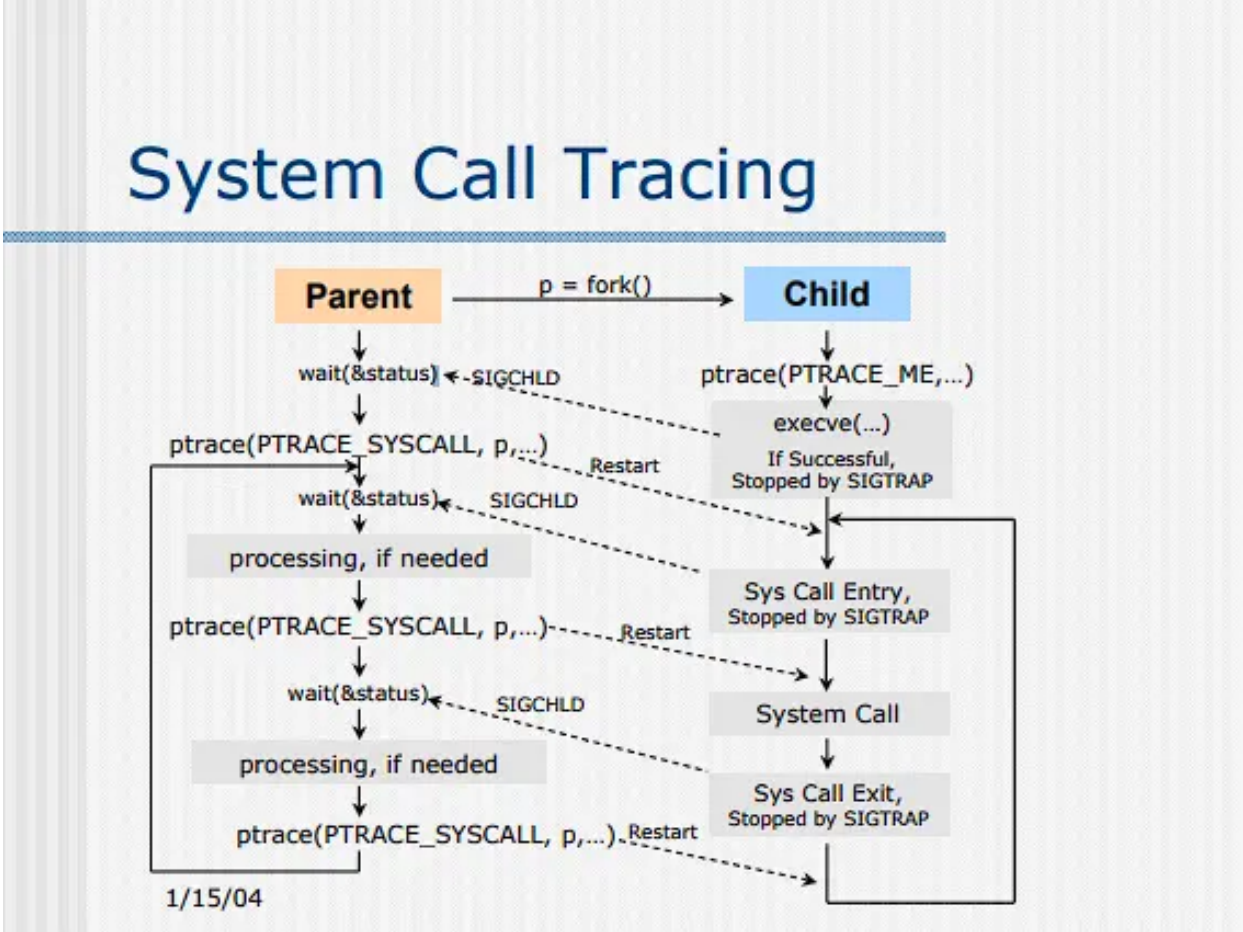
You can get similar vibes from this Brendan Gregg post on the dangers of using strace (that is based on ptrace) in production.

Although ptrace is cross-platform, actually writing cross-platform-aware code with ptrace can be complex. So this post assumes amd64/linux.

Protocol

The ptrace protocol is described in the ptrace manpage, but Chris Wellons and a University of Alberta group also wrote nice introductions. I referenced these three pages heavily.

Here's what the UAlberta page has to say:



We fork and have the child call `PTRACE_TRACEME`. Then we handle each syscall entrance by calling `PTRACE_SYSCALL` and waiting with `wait` until the child has entered the syscall. It is in this moment we can mess with things.

Implementation

Let's turn that graphic into Zig code.

```
const std = @import("std");
const c = @cImport({
    @cInclude("sys/ptrace.h");
    @cInclude("sys/user.h");
    @cInclude("sys/wait.h");
    @cInclude("errno.h");
});

const cNullPtr: ?*anyopaque = null;

// TODO //

pub fn main() !void {
    var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena.deinit();

    var args = try std.process.argsAlloc(arena.allocator());
    std.debug.assert(args.len >= 2);

    const pid = try std.os.fork();

    if (pid < 0) {
        std.debug.print("Fork failed!\n", .{});
        return;
    } else if (pid == 0) {
        // Child process
        _ = c.ptrace(c.PTRACE_TRACEME, pid, cNullPtr, cNullPtr);
        return std.process.execv(
            arena.allocator(),
            args[1..],
        );
    } else {
        // Parent process
        const childPid = pid;
        _ = c.waitpid(childPid, 0, 0);
        var cm = ChildManager{ .arena = &arena, .childPid = childPid };
        try cm.childInterceptSyscalls();
    }
}
```

So like the graphic suggested, we fork and start a child process. That means this Zig program should be called like:

```
$ zig build-exe --library c main.zig
$ ./main /actual/program/to/intercept --and --its args
```

Presumably, as with strace or gdb, we could instead attach to an already running process with `PTRACE_ATTACH` or `PTRACE_SEIZE` (based on the ptrace manpage) rather than going the `PTRACE_TRACEME` route. But I haven't tried that out yet.

With the child ready to be intercepted, we can implement the `ChildManager` that actually does the interception.

ChildManager

The core of the `ChildManager` is an infinite loop (at least as long as the child process lives) that waits for the next syscall and then calls a hook for the sytem call if it exists.

```
const ChildManager = struct {
    arena: *std.heap.ArenaAllocator,
    childPid: std.os.pid_t,

    // TODO //

    fn childInterceptSyscalls(
        cm: *ChildManager,
    ) !void {
        while (true) {
            // Handle syscall entrance
            const status = cm.childWaitForSyscall();
            if (std.os.W.IFEXITED(status)) {
                break;
            }

            var args: ABIArguments = cm.getABIArguments();
            const syscall = args.syscall();

            for (hooks) |hook| {
                if (syscall == hook.syscall) {
                    try hook.hook(cm.*, &args);
                }
            }
        }
    }
};
```

Later we'll write a hook for the `sys_write` syscall that will force an incomplete write.

Back to the protocol, `childWaitForSyscall` will call `PTRACE_SYSCALL` to allow the child process to start up again and continue until the next syscall. We'll follow that by `wait`-ing for the child process to be stopped again so we can handle the syscall entrance.

```
    fn childWaitForSyscall(cm: ChildManager) u32 {
        var status: i32 = 0;
        _ = c.ptrace(c.PTRACE_SYSCALL, cm.childPid, cNullPtr, cNullPtr);
        _ = c.waitpid(cm.childPid, &status, 0);
        return @bitCast(status);
    }
```

Now that we've intercepted a syscall (after `waitpid` finishes blocking), we need to figure out what syscall it was. We do this by calling `PTRACE_GETREGS` and reading the `rax` register which according to amd64/linux calling convention is the syscall called.

Registers

`PTRACE_GETREGS` fills out the following struct:

```
struct user_regs_struct
{
  unsigned long r15;
  unsigned long r14;
  unsigned long r13;
  unsigned long r12;
  unsigned long rbp;
  unsigned long rbx;
  unsigned long r11;
  unsigned long r10;
  unsigned long r9;
  unsigned long r8;
  unsigned long rax;
  unsigned long rcx;
  unsigned long rdx;
  unsigned long rsi;
  unsigned long rdi;
  unsigned long orig_rax;
  unsigned long rip;
  unsigned long cs;
  unsigned long eflags;
  unsigned long rsp;
  unsigned long ss;
  unsigned long fs_base;
  unsigned long gs_base;
  unsigned long ds;
  unsigned long es;
  unsigned long fs;
  unsigned long gs;
};
```

Let's write a little amd64/linux-specific wrapper for accessing meaningful fields.

```
const ABIArguments = struct {
    regs: c.user_regs_struct,

    fn nth(aa: ABIArguments, i: u8) c_ulong {
        std.debug.assert(i < 4);

        return switch (i) {
            0 => aa.regs.rdi,
            1 => aa.regs.rsi,
            2 => aa.regs.rdx,
            else => unreachable,
        };
    }

    fn setNth(aa: *ABIArguments, i: u8, value: c_ulong) void {
        std.debug.assert(i < 4);

        switch (i) {
            0 => { aa.regs.rdi = value; },
            1 => { aa.regs.rsi = value; },
            2 => { aa.regs.rdx = value; },
            else => unreachable,
        }
    }

    fn result(aa: ABIArguments) c_ulong { return aa.regs.rax; }

    fn setResult(aa: *ABIArguments, value: c_ulong) void {
        aa.regs.rax = value;
    }

    fn syscall(aa: ABIArguments) c_ulong { return aa.regs.orig_rax; }
};
```

One thing to note is that the field we read to get `rax` is not `aa.regs.rax` but `aa.regs.orig_rax`. This is because `rax` is also the return value and `PTRACE_SYSCALL` gets called twice for some syscalls on entrance and exit. The `orig_rax` field preserves the original `rax` value on syscall entrance. You can read more about this here.

Getting and setting registers

Now let's write the `ChildManager` code that actually calls `PTRACE_GETREGS` to fill out one of these structs.

```
    fn getABIArguments(cm: ChildManager) ABIArguments {
        var args = ABIArguments{ .regs = undefined };
        _ = c.ptrace(c.PTRACE_GETREGS, cm.childPid, cNullPtr, &args.regs);
        return args;
    }
```

Setting registers is similar, we just pass the struct back and call `PTRACE_SETREGS` instead:

```
fn setABIArguments(cm: ChildManager, args: *ABIArguments) void {
    _ = c.ptrace(c.PTRACE_SETREGS, cm.childPid, cNullPtr, &args.regs);
}
```

A hook

Now we finally have enough code to write a hook that can get and set registers; i.e. manipulate a system call!

We'll start by registering a `sys_write` hook in the `hooks` field we check in `childInterceptSyscalls` above.

```
const hooks = &[_]struct {
    syscall: c_ulong,
    hook: *const fn (ChildManager, *ABIArguments) anyerror!void,
}{.{
    .syscall = @intFromEnum(std.os.linux.syscalls.X64.write),
    .hook = writeHandler,
}};
```

If we look at the manpage for `write` we see it takes three arguments

1. The file descriptor (fd) to write to
2. The address to start writing data from
3. And the number of bytes to write

Going back to the calling convention that means the fd will be in `rdi`, the data address in `rsi`, and the data length in `rdx`.

So if we shorten the data length, we should be creating a short (incomplete) write.

```
fn writeHandler(cm: ChildManager, entryArgs: *ABIArguments) anyerror!void {
    const fd = entryArgs.nth(0);
    const dataAddress = entryArgs.nth(1);
    var dataLength = entryArgs.nth(2);

    // Truncate some bytes
    if (dataLength > 2) {
        dataLength -= 2;
        entryArgs.setNth(2, dataLength);
        cm.setABIArguments(entryArgs);
    }
}
```

In a more sophisticated version of this program, we could randomly decide when to truncate data and randomly decide how much data to truncate. However, for our purposes this is sufficient.

But there are some real problems with this code. When I ran this program against a basic Go program, I saw duplicate requests.

> Ah ok, PTRACE_SYSCALL gets hit when you both enter and exit a syscall.
>
> So each time you call PTRACE_SYSCALL and you do stuff, you just call it again afterwards to handle/wait for the exit. pic.twitter.com/PjmNwcMepG
>
> — Phil Eaton (@eatonphil) September 29, 2023

So the deal with `PTRACE_SYSCALL` is that for (most?) syscalls, you get to modify data before the data actually is handled by the kernel. And you get to modify data after the kernel has finished the syscall too.

This makes sense because `PTRACE_SYSCALL` (unlike `PTRACE_SYSEMU`) allows the syscall to actually happen. And if we wanted to, for example, modify the syscall exit code, we'd have to do that after the syscall was done not before it started. We are modifying registers directly after all.

All this means for our Zig code is that when we handle `sys_write`, we need to call `PTRACE_SYSCALL` again to process the syscall exit. Otherwise we'd reach this `writeHandler` for both entries and exits, which would require some additional way of disambiguating entrances from exits.

```
fn writeHandler(cm: ChildManager, entryArgs: *ABIArguments) anyerror!void {
    const fd = entryArgs.nth(0);
    const dataAddress = entryArgs.nth(1);
    var dataLength = entryArgs.nth(2);

    // Truncate some bytes
    if (dataLength > 2) {
        dataLength -= 2;
        entryArgs.setNth(2, dataLength);
        cm.setABIArguments(entryArgs);
    }

    const data = try cm.childReadData(dataAddress, dataLength);
    defer data.deinit();
    std.debug.print("Got a write on {}: {s}\n", .{ fd, data.items });

    // Handle syscall exit
    _ = cm.childWaitForSyscall();
}
```

We could put the `cm.childWaitForSyscall()` waiting for the syscall exit in the main loop and I did try that at first. However, not all syscalls seemed to have the same entry and exit hook and this resulted in the hooks sometimes starting with a syscall exit rather than a

syscall entry. So rather than making the code more complicated, I decided to only wait for the exit on syscalls I knew had an exit (by observation at least), like `sys_write`.

Multiple writes? No bad logic?

So I had this code as is, correctly handling syscall entrances and exits, but I was seeing multiple write calls. And the text file I was writing to had the complete text I wanted to write. There was no short write even though I truncated the data length.

> Ok so what happens in this Go program if I truncate the amount of data?
>
> I assumed Go would do nothing since all I did was call `f.Write()` once and `f.Write()` returns a number of bytes written.
>
> But actually, it still writes everything! pic.twitter.com/OSalKEbERM
>
> — Phil Eaton (@eatonphil) September 29, 2023

This took some digging into Go source code to understand. If you trace what `os.File.Write()` does on Linux you eventually get to src/internal/poll/fd_unix.go:

```
// Write implements io.Writer.
func (fd *FD) Write(p []byte) (int, error) {
        if err := fd.writeLock(); err != nil {
                return 0, err
        }
        defer fd.writeUnlock()
        if err := fd.pd.prepareWrite(fd.isFile); err != nil {
                return 0, err
        }
        var nn int
        for {
                max := len(p)
                if fd.IsStream && max-nn > maxRW {
                        max = nn + maxRW
                }
                n, err := ignoringEINTRIO(syscall.Write, fd.Sysfd, p[nn:max])
                if n > 0 {
                        nn += n
                }
                if nn == len(p) {
                        return nn, err
                }
                if err == syscall.EAGAIN && fd.pd.pollable() {
                        if err = fd.pd.waitWrite(fd.isFile); err == nil {
                                continue
                        }
                }
                if err != nil {
                        return nn, err
                }
                if n == 0 {
                        return nn, io.ErrUnexpectedEOF
                }
        }
}
```

This might be common knowledge but I didn't realize Go did this. And when I tried out the same basic program in Python and even C, the behavior was the same. The builtin `write()` behavior on a file (in many languages apparantly) is to retry until all data is written, with some exceptions.

This makes sense since files on disk, unlike file descriptors backed by network sockets, are generally always available. Compared to a network connection, disks are physically close and almost always stay connected. (With some obvious exceptions like network-attached storage and thumb drives.)

So to trigger the short write, the easiest way seems to have the `sys_write` call return an error that is NOT `EAGAIN` since the code will retry if that is the error.

After looking through the list of errors that sys_write can return, `EIO` seems like a nice one.

So let's do our final version of `writeHandler` and on the syscall exit, we'll modify the return value (`rax` in amd64/linux) to be `EIO`.

```
    fn writeHandler(cm: ChildManager, entryArgs: *ABIArguments) anyerror!void {
        const fd = entryArgs.nth(0);
        const dataAddress = entryArgs.nth(1);
        var dataLength = entryArgs.nth(2);

        // Truncate some bytes
        if (dataLength > 2) {
            dataLength -= 2;
            entryArgs.setNth(2, dataLength);
            cm.setABIArguments(entryArgs);
        }

        // Handle syscall exit
        _ = cm.childWaitForSyscall();

        var exitArgs = cm.getABIArguments();
        dataLength = exitArgs.nth(2);
        if (dataLength > 2) {
            // Force the writes to stop after the first one by returning EIO.
            var result: c_ulong = 0;
            result = result -% c.EIO;
            exitArgs.setResult(result);
            cm.setABIArguments(&exitArgs);
        }
    }
```

Let's give it a whirl!

All together

Build the Zig fault injector and the Go test code:

```
$ zig build-exe --library c main.zig
$ ( cd test && go build main.go )
```

And run:

```
$ ./main test/main
```

And check `test.txt`:

```
$ cat test.txt
some great stu
```

Hey, that's a short write! :)

Sidenote: Reading data from the child

We accomplished everything we set out to, but there's one other useful thing we can do: reading the actual data passed to the write syscall.

Just like how we can get the child process registers with `PTRACE_GETREGS`, we can read child memory with `PTRACE_PEEKDATA`. `PTRACE_PEEKDATA` takes the child process id and the memory address in the child to read from. It returns a word of data (which on amd64/linux is 8 bytes).

We can use the syscall arguments (data address and length) to keep calling `PTRACE_PEEKDATA` on the child until we've read all bytes of the data the child process wanted to write:

```
fn childReadData(
    cm: ChildManager,
    address: c_ulong,
    length: c_ulong,
) !std.ArrayList(u8) {
    var data = std.ArrayList(u8).init(cm.arena.allocator());
    while (data.items.len < length) {
        var word = c.ptrace(
            c.PTRACE_PEEKDATA,
            cm.childPid,
            address + data.items.len,
            cNullPtr,
        );

        for (std.mem.asBytes(&word)) |byte| {
            if (data.items.len == length) {
                break;
            }
            try data.append(byte);
        }
    }
    return data;
}
```

And we could modify `writeHandler` to print out the entirety of the write message each time (for debugging):

```
fn writeHandler(cm: ChildManager, entryArgs: *ABIArguments) anyerror!void {
    const fd = entryArgs.nth(0);
    const dataAddress = entryArgs.nth(1);
    var dataLength = entryArgs.nth(2);

    // Truncate some bytes
    if (dataLength > 2) {
        dataLength -= 2;
        entryArgs.setNth(2, dataLength);
        cm.setABIArguments(entryArgs);
    }

    const data = try cm.childReadData(dataAddress, dataLength);
    defer data.deinit();
    std.debug.print("Got a write on {}: {s}\n", .{ fd, data.items });

    // Handle syscall exit
    _ = cm.childWaitForSyscall();

    var exitArgs = cm.getABIArguments();
    dataLength = exitArgs.nth(2);
    if (dataLength > 2) {
        // Force the writes to stop after the first one by returning EIO.
        var result: c_ulong = 0;
        result = result -% c.EIO;
        exitArgs.setResult(result);
        cm.setABIArguments(&exitArgs);
    }
}
```

That's pretty neat!

Next steps

Short writes are just one of many bad IO interactions. Another fun one would be to completely buffer all writes on a file descriptor (not allowing anything to be written to disk at all) until fsync is called on the file descriptor. Or forcing fsyncs to fail.

An interesting optimization would be to apply seccomp filters so that rather than paying a penalty for watching every syscall, I only get notified about the ones I have hooks for like `sys_write`. Here's another post that explores ptrace with seccomp filters.

Credits: Thank you Charlie Cummings and Paul Khuong for reviewing a draft of this post!

Selected responses after publication

> oscooter on Reddit gave some tips on using ptrace, including using `process_vm_readv` instead of `PTRACE_PEEKDATA` to read memory from the tracee process.

> Fault injection is a scary-sounding term. Intercepting and modifying Linux system calls sounds scary too.
>
> But it's a neat way to trigger logical errors in programs, to build confidence we wrote code correctly.
>
> Let's trigger short writes to disk in Zig!https://t.co/0C3tWt3vtT pic.twitter.com/OS7auDe8jR
>
> — Phil Eaton (@eatonphil) October 1, 2023

Feedback

As always, please email or tweet me with questions, corrections, or ideas!