Writing GRUB modules in Rust

—

Daniel Axtens

What this *is* and *is **not***

A discussion of writing Rust modules in GRUB.

A talk by a GRUB hacker and C programmer.

Not an effort to replace GRUB.

Not an effort to rewrite all of GRUB in Rust.

Not an effort to make Rust a requirement for building GRUB.

Not an effort to drop or reduce support for your platform of choice.

Not an effort to break backwards compatibility.

Not a talk by a Rust language expert (yet).

Why bother with Rust
in GRUB?

Why Rust?

Squash unsafety bugs

Rust makes it much harder* to have buffer overflows, uses-after-free, and other nasty bugs.

With pure C, we are restricted to *trying hard* or *fuzzing everything*.

Trying hard hasn’t worked for us before, and hasn’t worked for anyone else.

Fuzzing is reactive at best.

* you can do anything in an `unsafe {}` block but at least it’s highlighted.

C compatible

Rust supports the ELF ABI which GRUB uses (even on e.g. UEFI platforms with other calling conventions to firmware)

Rust can call C code.

Rust can be called from C code, including via function pointers.

Rust doesn’t have a heavyweight runtime.

Platform support is decent

Rust (based on LLVM) supports a range of platforms.

But not all of them: no support for e.g. Itanium

Rust support for architectures that might do UEFI secure boot or Power secure boot is otherwise good.

We can provide both C and Rust versions of a module while GRUB supports architectures that Rust doesn’t.

Rust has been used in this domain before

There is a decent body of work using Rust in embedded systems.

The Rust for Linux project has dealt with the complexities of interoperating with C and non-Rust build systems in a very similar domain and we can learn from them.

GRUB modules provide an excellent ground for experimentation and proof-of-concept.

Rough proportion of bugs due to memory unsafety in large C/C++ codebases

~70%

https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/

Recent GRUB CVEs relating to memory unsafety

~40%

–**Memory unsafety:** CVE-2020-27749, CVE-2021-20233, CVE-2021-20225, CVE-2020-25647, CVE-2020-25632, CVE-2020-15706, CVE-2020-10713* (n=6/7)

–**Logic error:** CVE-2020-15705, CVE-2021-3418, CVE-2020-27779, CVE-2020-14372, CVE-2020-10713* (n=4/5)

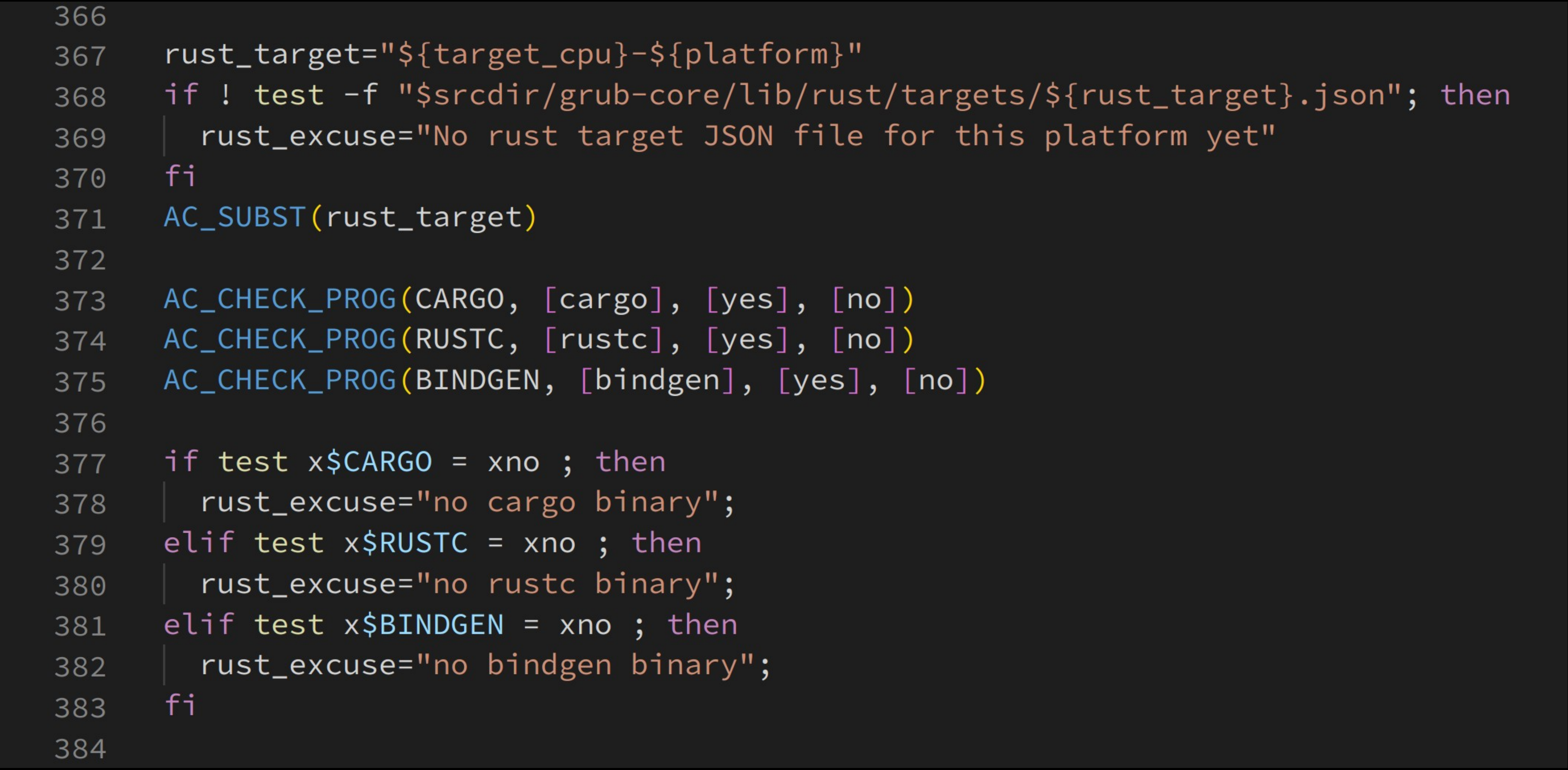
–**Integer overflow:** CVE-2020-15707, CVE-2020-14309, CVE-2020-14310, CVE-2020-14311, CVE-2020-14308 (n=5) – all leading to potential memory corruption

– *CVE-2020-10713 – yylex not fatal enough: logic bug leads to exploitable memory corruption

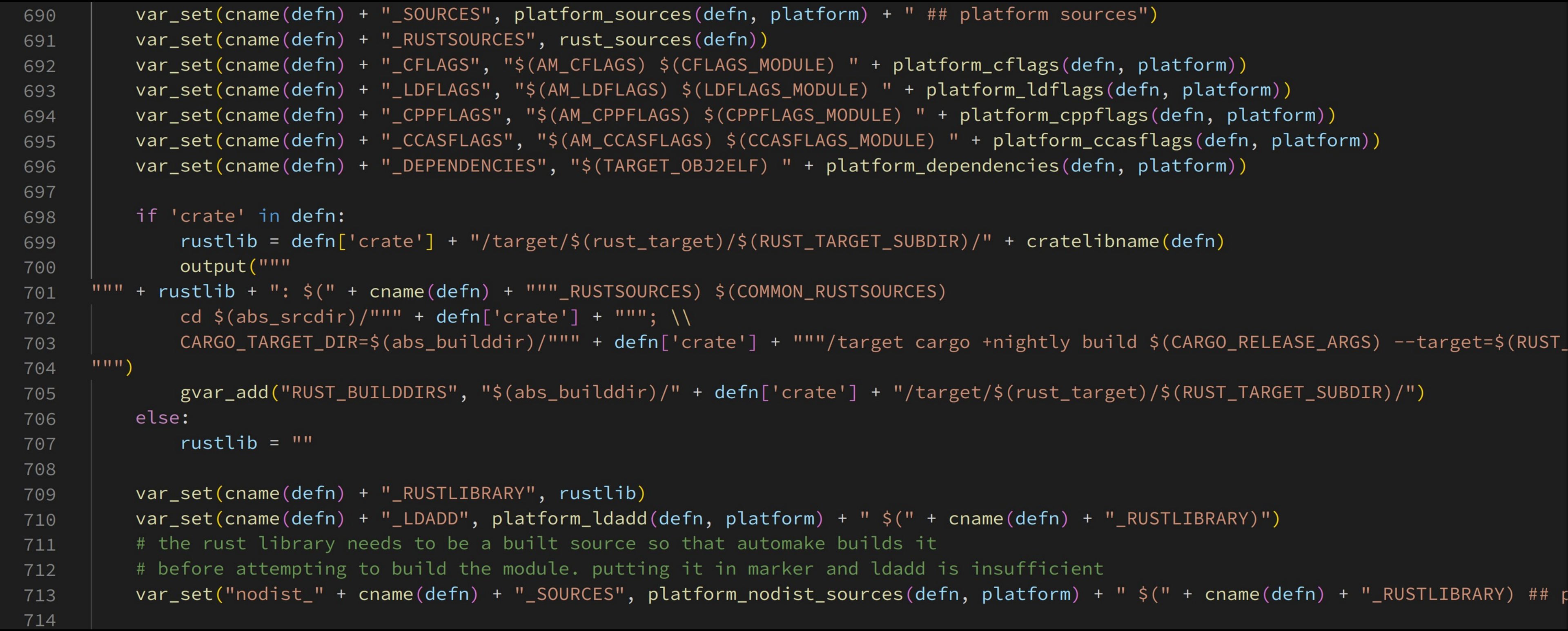
–Total n=16

Show me the code

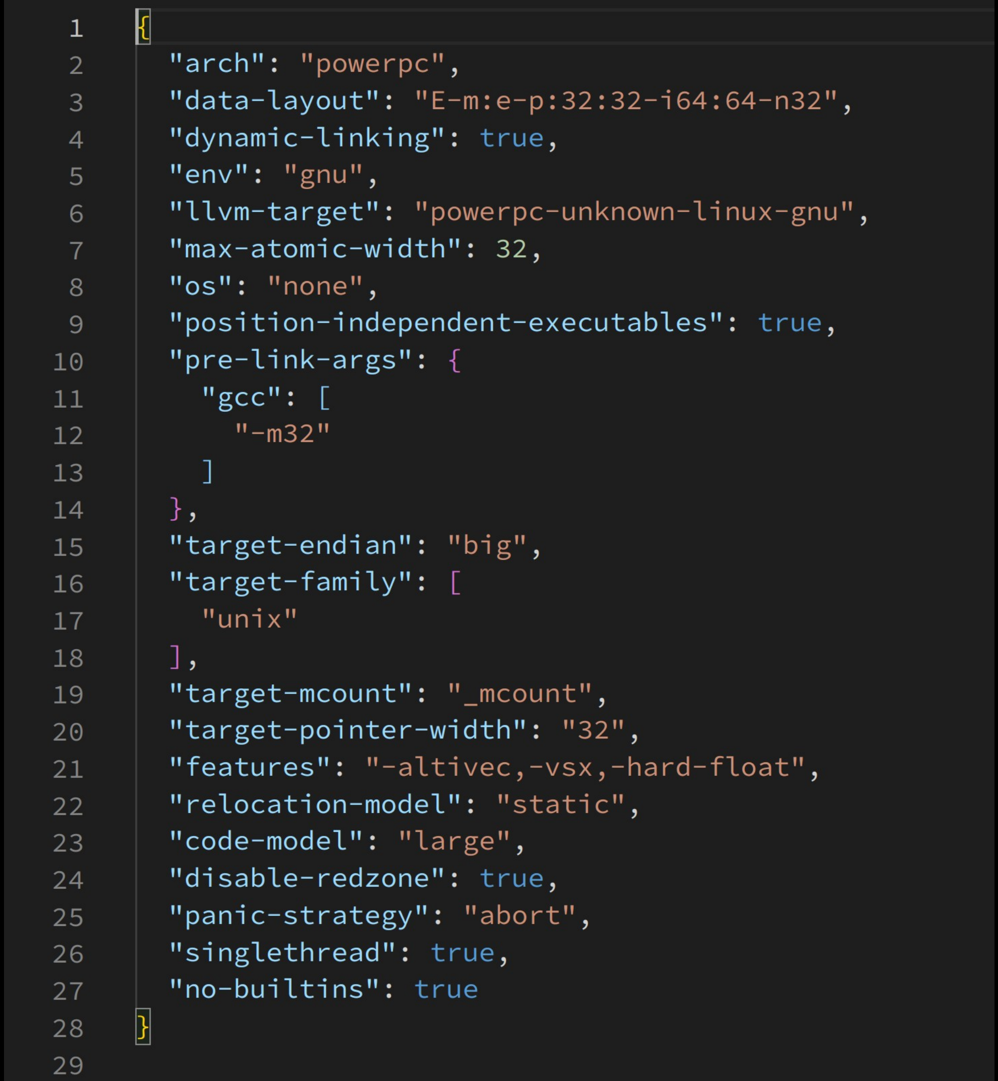
Show me the code, the ‘something has gone wrong with screen sharing’ edition



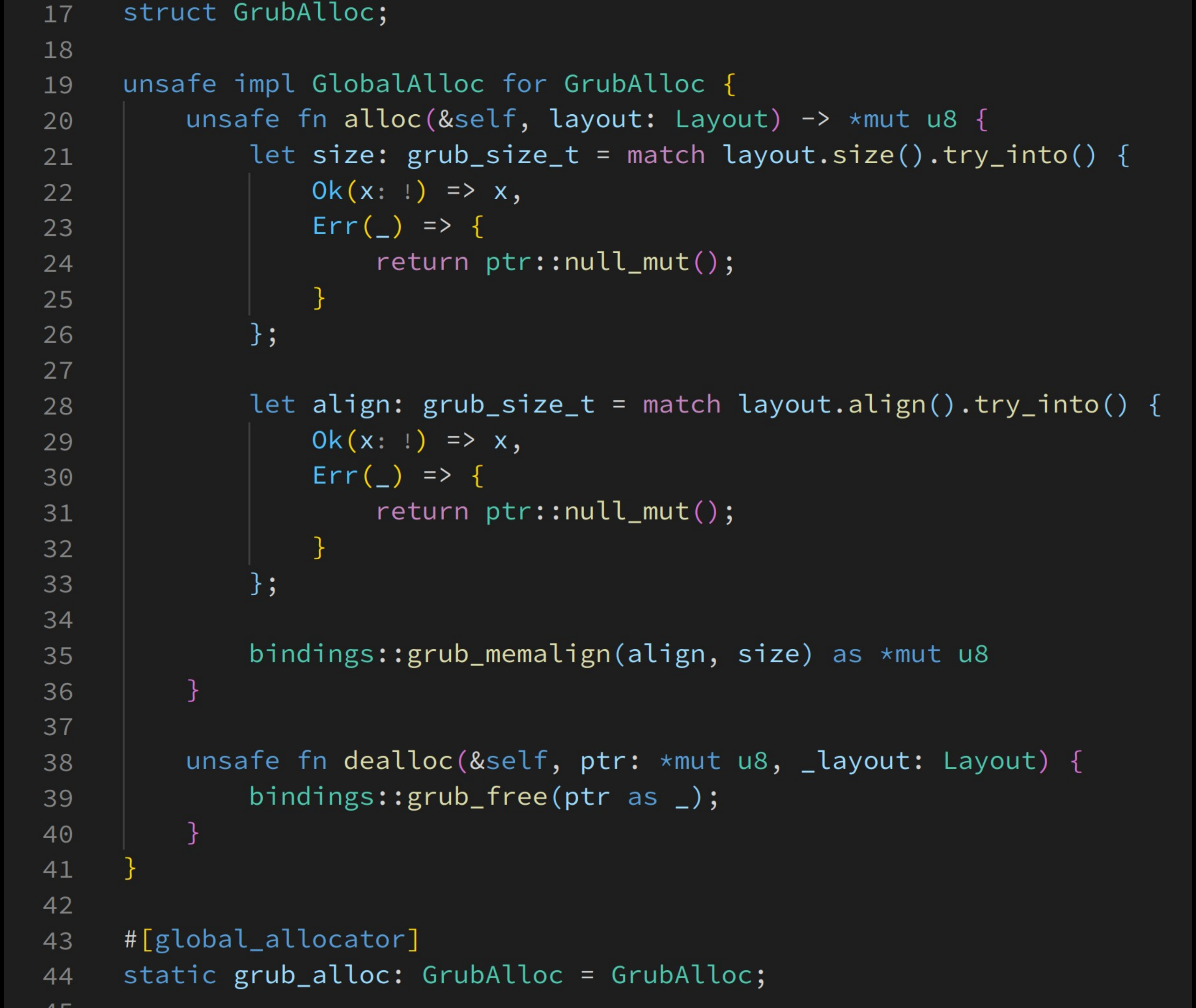
```

rust_target="${target_cpu}-${platform}"
if ! test -f "$srcdir/grub-core/lib/rust/targets/${rust_target}.json"; then
  rust_excuse="No rust target JSON file for this platform yet"
fi
AC_SUBST(rust_target)

AC_CHECK_PROG(CARGO, [cargo], [yes], [no])
AC_CHECK_PROG(RUSTC, [rustc], [yes], [no])
AC_CHECK_PROG(BINDGEN, [bindgen], [yes], [no])

if test x$CARGO = xno ; then
  rust_excuse="no cargo binary";
elif test x$RUSTC = xno ; then
  rust_excuse="no rustc binary";
elif test x$BINDGEN = xno ; then
  rust_excuse="no bindgen binary";
fi

```



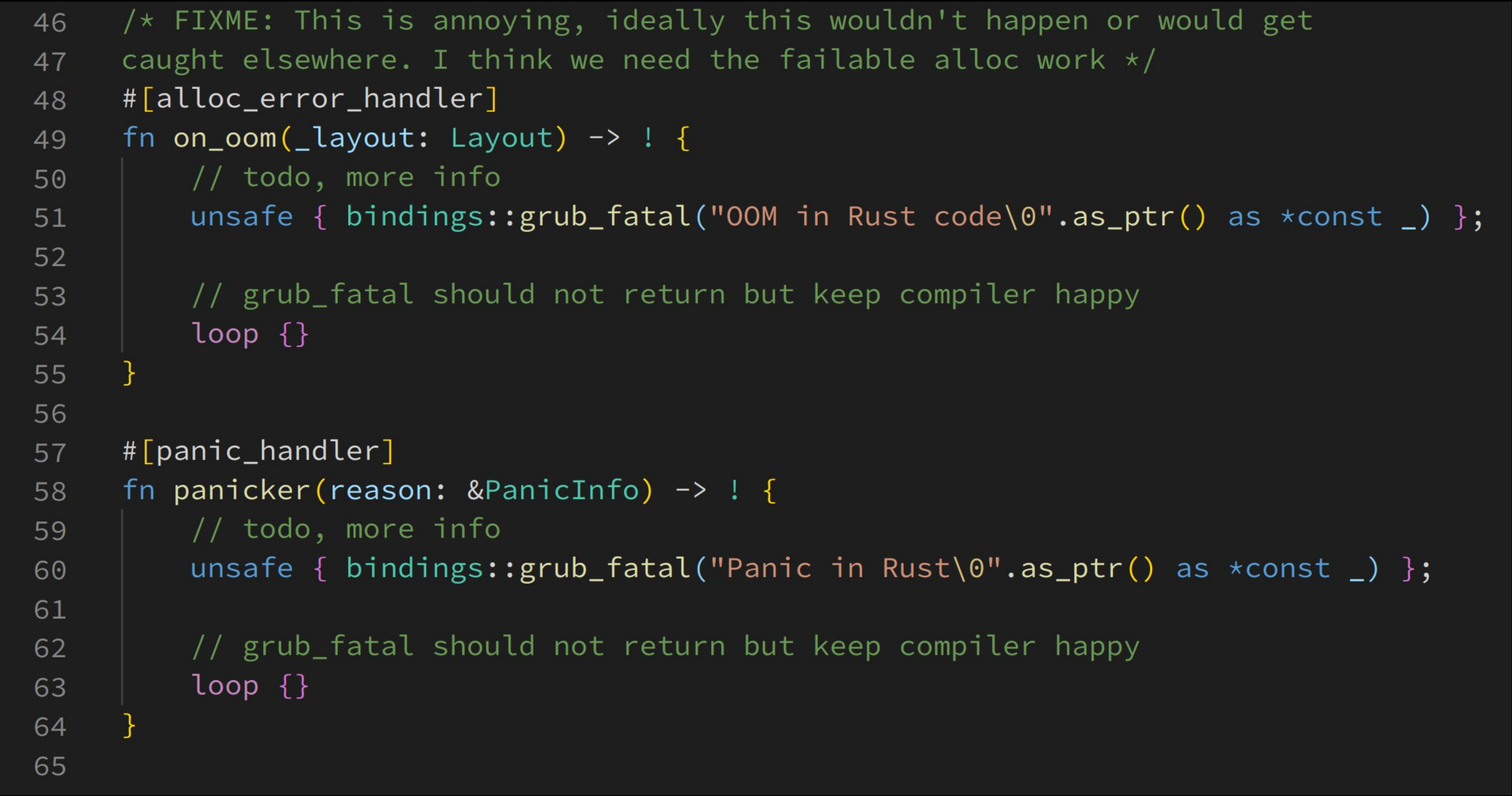
```
    var_set(cname(defn) + "_SOURCES", platform_sources(defn, platform) + " ## platform sources")
    var_set(cname(defn) + "_RUSTSOURCES", rust_sources(defn))
    var_set(cname(defn) + "_CFLAGS", "$(AM_CFLAGS) $(CFLAGS_MODULE) " + platform_cflags(defn, platform))
    var_set(cname(defn) + "_LDFLAGS", "$(AM_LDFLAGS) $(LDFLAGS_MODULE) " + platform_ldflags(defn, platform))
    var_set(cname(defn) + "_CPPFLAGS", "$(AM_CPPFLAGS) $(CPPFLAGS_MODULE) " + platform_cppflags(defn, platform))
    var_set(cname(defn) + "_CCASFLAGS", "$(AM_CCASFLAGS) $(CCASFLAGS_MODULE) " + platform_ccasflags(defn, platform))
    var_set(cname(defn) + "_DEPENDENCIES", "$(TARGET_OBJ2ELF) " + platform_dependencies(defn, platform))

    if 'crate' in defn:
        rustlib = defn['crate'] + "/target/$(rust_target)/$(RUST_TARGET_SUBDIR)/" + cratelibname(defn)
        output("""
""" + rustlib + ": $(" + cname(defn) + """_RUSTSOURCES) $(COMMON_RUSTSOURCES)
        cd $(abs_srcdir)/""" + defn['crate'] + """; \\
        CARGO_TARGET_DIR=$(abs_builddir)/""" + defn['crate'] + """/target cargo +nightly build $(CARGO_RELEASE_ARGS) --target=$(RUST_
""")
        gvar_add("RUST_BUILDDIRS", "$(abs_builddir)/" + defn['crate'] + "/target/$(rust_target)/$(RUST_TARGET_SUBDIR)/")
    else:
        rustlib = ""

    var_set(cname(defn) + "_RUSTLIBRARY", rustlib)
    var_set(cname(defn) + "_LDADD", platform_ldadd(defn, platform) + " $(" + cname(defn) + "_RUSTLIBRARY)")
    # the rust library needs to be a built source so that automake builds it
    # before attempting to build the module. putting it in marker and ldadd is insufficient
    var_set("nodist_" + cname(defn) + "_SOURCES", platform_nodist_sources(defn, platform) + " $(" + cname(defn) + "_RUSTLIBRARY) ## p

```



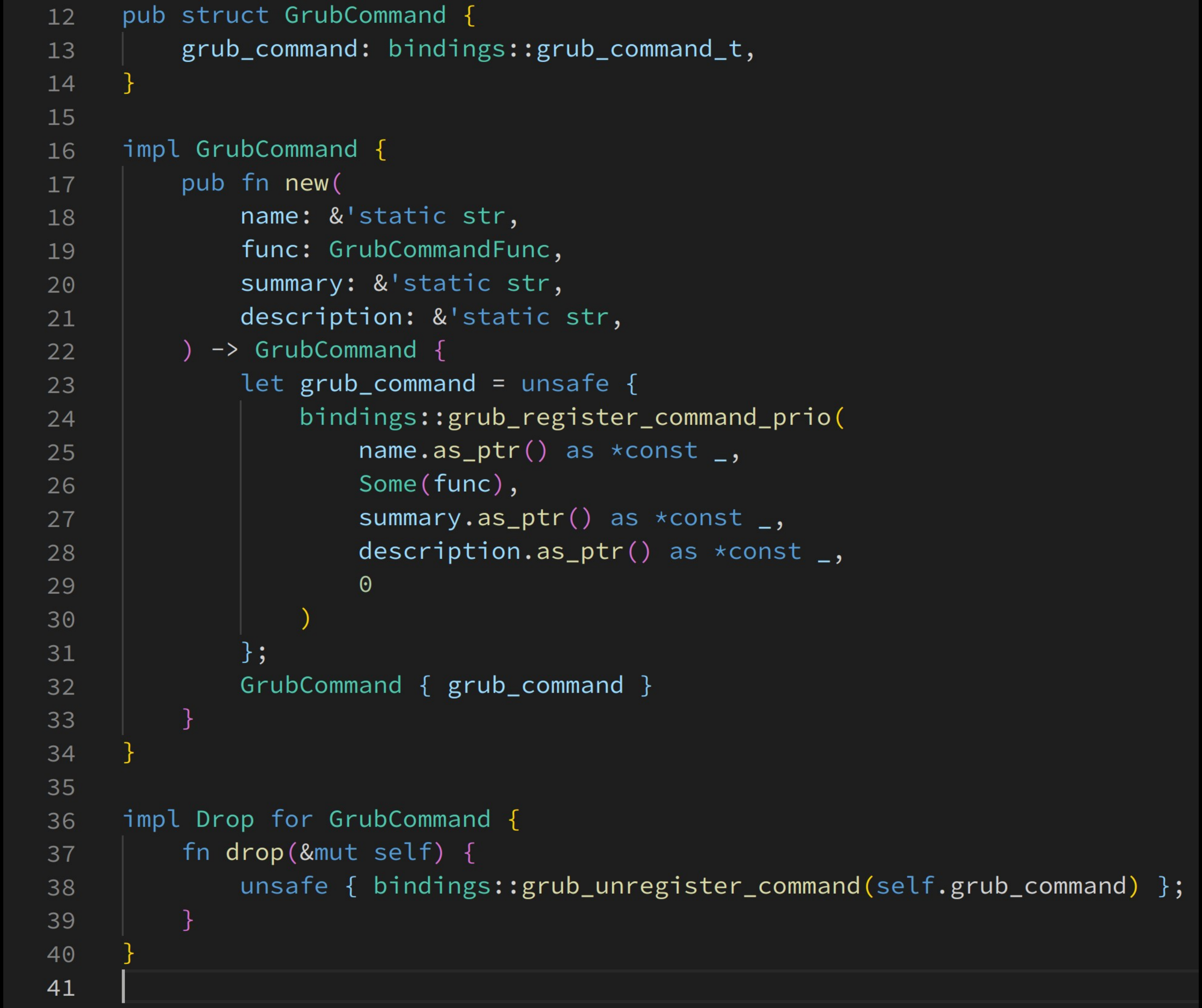
```json
{
  "arch": "powerpc",
  "data-layout": "E-m:e-p:32:32-i64:64-n32",
  "dynamic-linking": true,
  "env": "gnu",
  "llvm-target": "powerpc-unknown-linux-gnu",
  "max-atomic-width": 32,
  "os": "none",
  "position-independent-executables": true,
  "pre-link-args": {
    "gcc": [
      "-m32"
    ]
  },
  "target-endian": "big",
  "target-family": [
    "unix"
  ],
  "target-mcount": "_mcount",
  "target-pointer-width": "32",
  "features": "-altivec,-vsx,-hard-float",
  "relocation-model": "static",
  "code-model": "large",
  "disable-redzone": true,
  "panic-strategy": "abort",
  "singlethread": true,
  "no-builtins": true
}

```



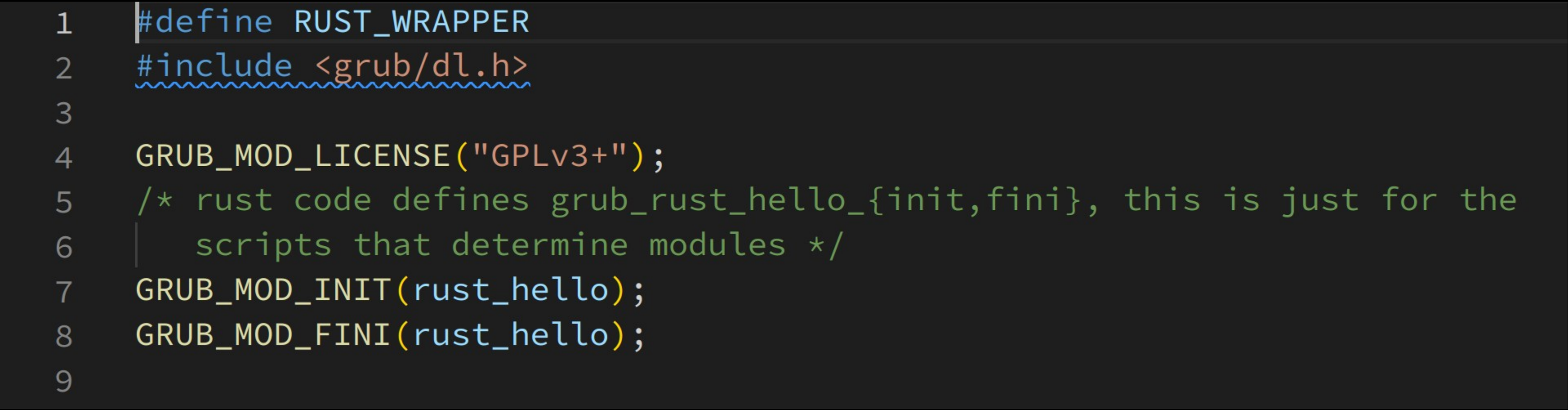
```
struct GrubAlloc;

unsafe impl GlobalAlloc for GrubAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        let size: grub_size_t = match layout.size().try_into() {
            Ok(x: !) => x,
            Err(_) => {
                return ptr::null_mut();
            }
        };

        let align: grub_size_t = match layout.align().try_into() {
            Ok(x: !) => x,
            Err(_) => {
                return ptr::null_mut();
            }
        };

        bindings::grub_memalign(align, size) as *mut u8
    }

    unsafe fn dealloc(&self, ptr: *mut u8, _layout: Layout) {
        bindings::grub_free(ptr as _);
    }
}

#[global_allocator]
static grub_alloc: GrubAlloc = GrubAlloc;
```



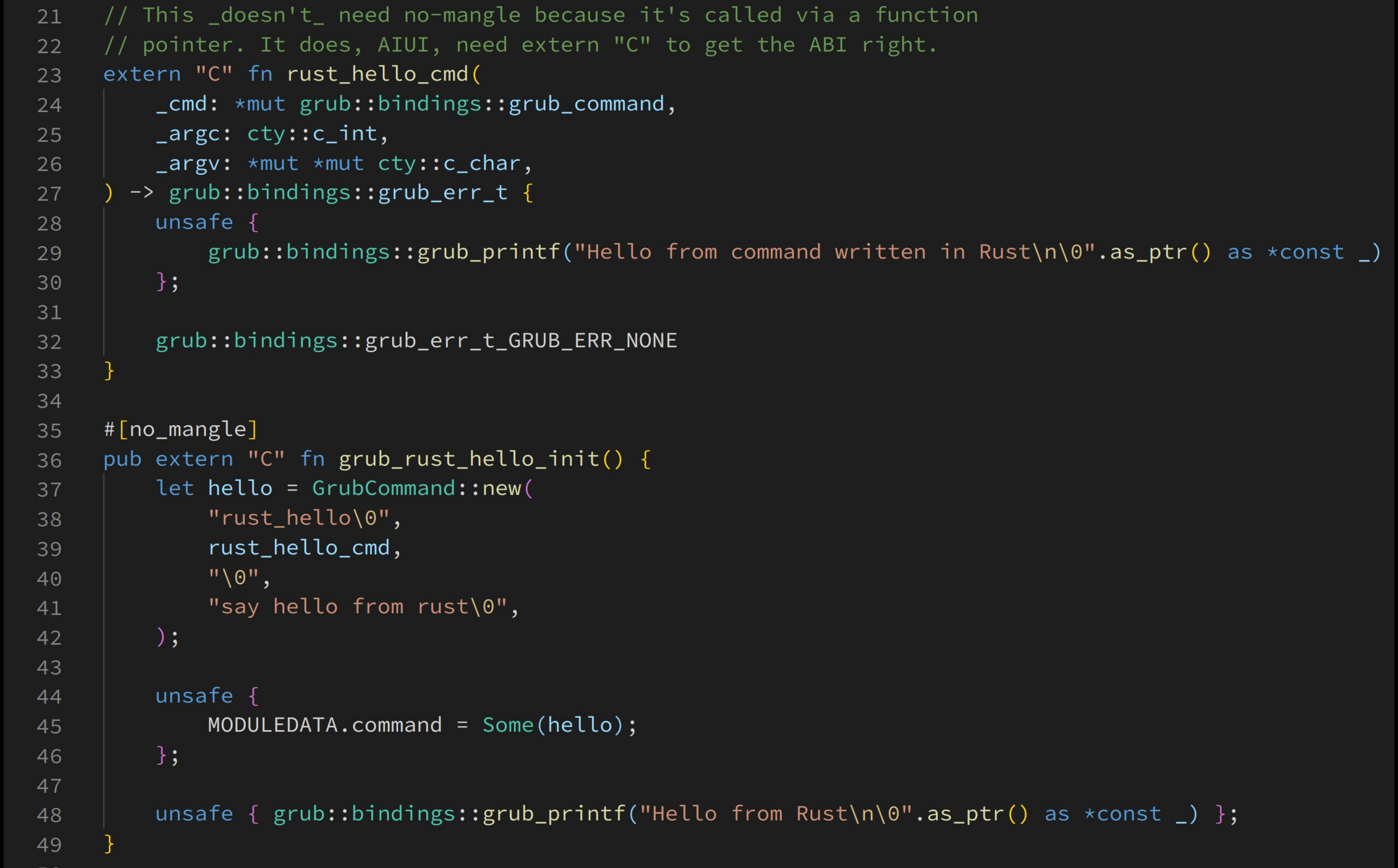
```rust
/* FIXME: This is annoying, ideally this wouldn't happen or would get
caught elsewhere. I think we need the failable alloc work */
#[alloc_error_handler]
fn on_oom(_layout: Layout) -> ! {
    // todo, more info
    unsafe { bindings::grub_fatal("OOM in Rust code\0".as_ptr() as *const _) };

    // grub_fatal should not return but keep compiler happy
    loop {}
}

#[panic_handler]
fn panicker(reason: &PanicInfo) -> ! {
    // todo, more info
    unsafe { bindings::grub_fatal("Panic in Rust\0".as_ptr() as *const _) };

    // grub_fatal should not return but keep compiler happy
    loop {}
}

```



```rust
pub struct GrubCommand {
    grub_command: bindings::grub_command_t,
}

impl GrubCommand {
    pub fn new(
        name: &'static str,
        func: GrubCommandFunc,
        summary: &'static str,
        description: &'static str,
    ) -> GrubCommand {
        let grub_command = unsafe {
            bindings::grub_register_command_prio(
                name.as_ptr() as *const _,
                Some(func),
                summary.as_ptr() as *const _,
                description.as_ptr() as *const _,
                0
            )
        };
        GrubCommand { grub_command }
    }
}

impl Drop for GrubCommand {
    fn drop(&mut self) {
        unsafe { bindings::grub_unregister_command(self.grub_command) };
    }
}

```



```c
#define RUST_WRAPPER
#include <grub/dl.h>

GRUB_MOD_LICENSE("GPLv3+");
/* rust code defines grub_rust_hello_{init,fini}, this is just for the
   scripts that determine modules */
GRUB_MOD_INIT(rust_hello);
GRUB_MOD_FINI(rust_hello);

```



```rust
// This _doesn't_ need no-mangle because it's called via a function
// pointer. It does, AIUI, need extern "C" to get the ABI right.
extern "C" fn rust_hello_cmd(
    _cmd: *mut grub::bindings::grub_command,
    _argc: cty::c_int,
    _argv: *mut *mut cty::c_char,
) -> grub::bindings::grub_err_t {
    unsafe {
        grub::bindings::grub_printf("Hello from command written in Rust\n\0".as_ptr() as *const _)
    };

    grub::bindings::grub_err_t_GRUB_ERR_NONE
}

#[no_mangle]
pub extern "C" fn grub_rust_hello_init() {
    let hello = GrubCommand::new(
        "rust_hello\0",
        rust_hello_cmd,
        "\0",
        "say hello from rust\0",
    );

    unsafe {
        MODULEDATA.command = Some(hello);
    };

    unsafe { grub::bindings::grub_printf("Hello from Rust\n\0".as_ptr() as *const _) };
}
```

Where to from here?

Questions the GRUB project needs to decide.

Do we want Rust in GRUB?

How much do we want to interact with the Rust build system, cargo?

- Completely bypass it? (Rust for Linux)
- Use it to build rust modules? (my early RFC)
- Rewrite our build system around it? (lol)

Should we target a specific version of Rust?

Rust for Linux does this – Rust 1.54 and the unstable features therein.

What about platforms with no or broken Rust support?

Are we happy providing 2, largely functionally identical, sets of modules?

How will we deal with the alloc failure → panic problem?

Rust for Linux has had to address this but I haven’t looked at how they do it yet.

Other questions

How do we build .lst files in the presence of Rust modules?

- E.g. dyncmd reads command.lst to autoload a module when the relevant command is called.
- command.lst is generated by looking for things like COMMAND_LIST_MARKER in a bundle of preprocessed sources.
- Rust does not use the C preprocessor.
- Rust will probably have to specify {commands, file systems, terminals, …} explicitly – should C do that too?

How will we support Rust’s built in testing framework? (and formatting and linting from the Rust ecosystem)

Rust for Linux has a solution to this but I haven’t fully comprehended it yet.

How much do we want to directly borrow from Rust for Linux?

- GPLv3+ vs GPLv2
- Do we still need FSF copyright assignments?

Your questions

Thank you

Daniel Axtens
Linux Security Engineer
—
daniel.axtens1@ibm.com
dja@axtens.net

The views expressed in this presentation are those of the author, not necessarily those of IBM.

IBM