```
cannot execute: required file not found
```

Nix Anywhere Else
Relocatable Binaries via ELF Surgery

Wes Payne · LINUX Unplugged · Planet Nix 2026 @ SCALE 23x

You know this pattern

```
$ ./myapp
./myapp: error while loading shared libraries:
  libstdc++.so.6: cannot open shared object file: No such file or directory
```

```
$ python3 -c "import numpy"
ImportError: libz.so: cannot open shared object file: No such file or directory
```

```
$ nix run nixpkgs#some-prebuilt-tool
error: file '/lib64/ld-linux-x86-64.so.2' was not found
```

The usual answers

```
programs.nix-ld.enable = true;
```

```
services.envfs.enable = true;
```

```
buildFHSEnv { ... }
```

These fix 80% of cases. `nix-ld` especially — reach for it first.

But they don’t help when you’re building something that needs to *leave* NixOS.

Understanding *why* they work unlocks something more powerful.

Your binary is a data structure

ELF — **E**xecutable and **L**inkable **F**ormat.
Not a magic blob. A record with fields.

```
;; what lives inside your ELF binary

{:interpreter "/lib64/ld-linux-x86-64.so.2"   ; who loads me?

 :rpath       ["/nix/store/abc123-glibc/lib"]  ; where are my libs?

 :needed      ["libz.so"                        ; what do I need?
               "libssl.so.1"
               "libstdc++.so.6"]}
```

Your binary is a data structure

ELF — **E**xecutable and **L**inkable **F**ormat.
Not a magic blob. A record with fields.

```
;; what lives inside your ELF binary

{:interpreter "/lib64/ld-linux-x86-64.so.2"   ; who loads me?

 :rpath       ["/nix/store/abc123-glibc/lib"]  ; where are my libs?

 :needed      ["libz.so"                        ; what do I need?
               "libssl.so.1"
               "libstdc++.so.6"]}
```

Your binary is a data structure

ELF — **E**xecutable and **L**inkable **F**ormat.
Not a magic blob. A record with fields.

```
;; what lives inside your ELF binary

{:interpreter "/lib64/ld-linux-x86-64.so.2"   ; who loads me?

 :rpath       ["/nix/store/abc123-glibc/lib"]  ; where are my libs?

 :needed      ["libz.so"                       ; what do I need?
               "libssl.so.1"
               "libstdc++.so.6"]}
```

Your binary is a data structure

ELF — **E**xecutable and **L**inkable **F**ormat.
Not a magic blob. A record with fields.

```
;; what lives inside your ELF binary

{:interpreter "/lib64/ld-linux-x86-64.so.2"   ; who loads me?

 :rpath       ["/nix/store/abc123-glibc/lib"]  ; where are my libs?

 :needed      ["libz.so"                        ; what do I need?
               "libssl.so.1"
               "libstdc++.so.6"]}
```

Reading the fields

```
$ readelf -l ./gws | grep interpreter
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]

$ readelf -d ./gws | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [ld-linux-x86-64.so.2]
```

```
$ ldd ./gws
  linux-vdso.so.1 (0x00007ffff7fc2000)
  libgcc_s.so.1 => not found
  libm.so.6 => not found
  libc.so.6 => not found
  /lib64/ld-linux-x86-64.so.2 => not found
```

`gws` — Google's new Workspace CLI — dropped this week. Nix flake in the repo.
But if you just grab the binary, this is what NixOS sees.

Standard Distro

```
$ ldd ./gws
  linux-vdso.so.1 (0x00007ffff7fc2000)
  libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1
  libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
  /lib64/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.
```

- ✓ `/lib64/ld-linux-x86-64.so.2` exists
- ✓ `/usr/lib` is populated
- ✓ everything resolves

NixOS

```
$ ldd ./gws
  linux-vdso.so.1 (0x00007ffff7fc2000)
  libgcc_s.so.1 => not found
  libm.so.6 => not found
  libc.so.6 => not found
  /lib64/ld-linux-x86-64.so.2 => not found
```

- ✗ `/lib64/ld-linux-x86-64.so.2` doesn't exist
- ✗ no `/usr/lib`
- ✗ everything lives in `/nix/store/<hash>/lib`

patchelf

If `readelf` and `ldd` are readers, patchelf is the writer.

Written by Eelco Dolstra. Part of the Nix project since 2004.

```
# fix the interpreter
patchelf --set-interpreter \
  /nix/store/abc123-glibc/lib/ld-linux-x86-64.so.2 ./binary

# fix the library search path
patchelf --set-rpath \
  /nix/store/abc123-glibc/lib:/nix/store/def456-zlib/lib ./binary

# inject a new dependency
patchelf --add-needed libz.so.1 ./binary
```

In nixpkgs: `autoPatchelfHook` automates this for prebuilt binaries.
Scans outputs, finds `DT_NEEDED` entries, patches interpreter + RPATH automatically.

Before → After

```
# ldd ./gws (stock binary)
  linux-vdso.so.1 (0x00007ffff7fc2000)
  libgcc_s.so.1 => not found
  libm.so.6 => not found
  libc.so.6 => not found
  /lib64/ld-linux-x86-64.so.2 => not found
```

We changed some strings in a file.
The machine code didn’t change. Just the fields.

Before → After

```
# ldd ./gws (patched via flake)
  linux-vdso.so.1 (0x00007ffff7fc2000)
  libgcc_s.so.1 => /nix/store/...-gcc-15.2.0-libgcc/lib/libgcc_s.so.1
  libm.so.6 => /nix/store/...-glibc-2.42-51/lib/libm.so.6
  libc.so.6 => /nix/store/...-glibc-2.42-51/lib/libc.so.6
  /lib64/ld-linux-x86-64.so.2 => /nix/store/...-glibc-2.42-51/lib64/ld-linux-x86-64.so.2
```

We changed some strings in a file.
The machine code didn’t change. Just the fields.

Python on NixOS

The pain is real. The cause is specific.

```
$ uv pip install numpy
# installs into .venv/lib/python3.13/site-packages/numpy/
# numpy's C extensions live outside the Nix store
# they have DT_NEEDED entries for libstdc++.so.6, libz.so.1

$ python3 -c "import numpy"
ImportError: libstdc++.so.6: cannot open shared object file: No such file or directory
```

`pip` and `uv` don't solve the `.so` layer. Nix does — but it needs a little help.

The usual band-aid: `export LD_LIBRARY_PATH=$(nix eval --raw nixpkgs#gcc.lib)/lib`

It works. Until you close the terminal.

Patch the operating table, not the patient

```
pkgs.stdenv.mkDerivation {
  name = "pythonWithLibs";
  # ... copy standard python to $out ...

  nativeBuildInputs = [ pkgs.autoPatchelfHook ];
  buildInputs = [ pkgs.zlib pkgs.stdenv.cc.cc.lib ];

  installPhase = ''
    patchelf --add-needed "libz.so.1"      "$out/bin/python3.13"
    patchelf --add-needed "libstdc++.so.6" "$out/bin/python3.13"
  '';
}
```

When Python loads, the linker pulls those libraries into the process.
Every C extension that loads later finds them already there.

Patch the operating table, not the patient

```
pkgs.stdenv.mkDerivation {
  name = "pythonWithLibs";
  # ... copy standard python to $out ...

  nativeBuildInputs = [ pkgs.autoPatchelfHook ];
  buildInputs = [ pkgs.zlib pkgs.stdenv.cc.cc.lib ];

  installPhase = ''
    patchelf --add-needed "libz.so.1"      "$out/bin/python3.13"
    patchelf --add-needed "libstdc++.so.6" "$out/bin/python3.13"
  '';
}
```

When Python loads, the linker pulls those libraries into the process.
Every C extension that loads later finds them already there.

Patch the operating table, not the patient

```
pkgs.stdenv.mkDerivation {
  name = "pythonWithLibs";
  # ... copy standard python to $out ...

  nativeBuildInputs = [ pkgs.autoPatchelfHook ];
  buildInputs = [ pkgs.zlib pkgs.stdenv.cc.cc.lib ];

  installPhase = ''
    patchelf --add-needed "libz.so.1"      "$out/bin/python3.13"
    patchelf --add-needed "libstdc++.so.6" "$out/bin/python3.13"
  '';
}
```

When Python loads, the linker pulls those libraries into the process.
Every C extension that loads later finds them already there.

Patch the operating table, not the patient

```
pkgs.stdenv.mkDerivation {
  name = "pythonWithLibs";
  # ... copy standard python to $out ...

  nativeBuildInputs = [ pkgs.autoPatchelfHook ];
  buildInputs = [ pkgs.zlib pkgs.stdenv.cc.cc.lib ];

  installPhase = ''
    patchelf --add-needed "libz.so.1"      "$out/bin/python3.13"
    patchelf --add-needed "libstdc++.so.6" "$out/bin/python3.13"
  '';
}
```

When Python loads, the linker pulls those libraries into the process.
Every C extension that loads later finds them already there.

It works

```
==== TESTING PYTHON .../python3.13 ====
ERROR
ImportError: libstdc++.so.6: cannot open shared object file: No such file or directory
Original error was: libstdc++.so.6: cannot open shared object file: No such file or directory
======

==== TESTING PYTHON .../pythonWithLibs/bin/python3.13 ====
SUCCESS
======
```

Same numpy. Same venv. Same everything.
Two `` `patchelf --add-needed` `` calls on the interpreter is the only difference.

It works

```
==== TESTING PYTHON .../python3.13 ====
ERROR
ImportError: libstdc++.so.6: cannot open shared object file: No such file or directory
Original error was: libstdc++.so.6: cannot open shared object file: No such file or directory
======

==== TESTING PYTHON .../pythonWithLibs/bin/python3.13 ====
SUCCESS
======
```

Same numpy. Same venv. Same everything.
Two `` `patchelf --add-needed` `` calls on the interpreter is the only difference.

It works

```
==== TESTING PYTHON .../python3.13 ====
ERROR
ImportError: libstdc++.so.6: cannot open shared object file: No such file or directory
Original error was: libstdc++.so.6: cannot open shared object file: No such file or directory
======

==== TESTING PYTHON .../pythonWithLibs/bin/python3.13 ====
SUCCESS
======
```

Same numpy. Same venv. Same everything.
Two `` `patchelf --add-needed` `` calls on the interpreter is the only difference.

Surgery In

Foreign binary → NixOS

- `--set-interpreter` → Nix store `ld-linux`
- `--set-rpath` → Nix store lib paths
- `--add-needed` → pre-load missing libs

Prebuilt binaries, pip C extensions,
anything built outside NixOS.

Surgery Out

Nix binary → anywhere

- `--set-rpath '$ORIGIN/../lib'` → relative paths
- Bundle required `.so` files alongside binary
- Ship it. No Nix store required.

Standard distros, CI artifacts,
portable tools.

bcachefs-tools on Debian

bcachefs-tools: C + Rust, `Cargo.lock`, pinned deps.
Traditional distro packaging has institutional friction with that.
For a period: no maintained Debian package.

NixOS has no such constraint. nixpkgs already has `bcachefs-tools`.

```
# bcachefs-tools — simplified flake.nix
{
  bcachefs-tools = pkgs.bcachefs-tools.override { fuseSupport = true; };

  libs = pkgs.symlinkJoin {
    name = "bcachefs-libs";
    paths = [
      pkgs.fuse3.out pkgs.glibc pkgs.libsodium
      pkgs.libaio pkgs.libcap.lib pkgs.zstd
      # ... and 8 other dependencies
    ];
  };
}
```

The surgery

```
# 1. build with nix; copy out; resolve symlinks
nix build .#bcachefs-tools
cp -r ./result/. ./bcachefs-tools/
# dereference nix store symlinks to real files
fix_absolute_symlinks ./bcachefs-tools

# 2. patch the binaries
patchelf --set-interpreter \
  "/usr/local/nix/lib/ld-linux-x86-64.so.2" \
  ./bcachefs-tools/bin/*
patchelf --set-rpath '$ORIGIN/../nix/lib' \
  ./bcachefs-tools/bin/*

# 3. patch the libs (point at each other; skip the baseline)
find ./libs/lib -name '*.so*' \
  -not -name '*ld-linux*' -not -name 'libc.so' \
  -not -name 'libgcc_s.so' -not -name 'libm.so' \
  -exec patchelf --set-rpath '$ORIGIN' {} \;
```

The surgery

```
# 1. build with nix; copy out; resolve symlinks
nix build .#bcachefs-tools
cp -r ./result/. ./bcachefs-tools/
# dereference nix store symlinks to real files
fix_absolute_symlinks ./bcachefs-tools

# 2. patch the binaries
patchelf --set-interpreter \
  "/usr/local/nix/lib/ld-linux-x86-64.so.2" \
  ./bcachefs-tools/bin/*
patchelf --set-rpath '$ORIGIN/../nix/lib' \
  ./bcachefs-tools/bin/*

# 3. patch the libs (point at each other; skip the baseline)
find ./libs/lib -name '*.so*' \
  -not -name '*ld-linux*' -not -name 'libc.so' \
  -not -name 'libgcc_s.so' -not -name 'libm.so' \
  -exec patchelf --set-rpath '$ORIGIN' {} \;
```

The surgery

```
# 1. build with nix; copy out; resolve symlinks
nix build .#bcachefs-tools
cp -r ./result/. ./bcachefs-tools/
# dereference nix store symlinks to real files
fix_absolute_symlinks ./bcachefs-tools

# 2. patch the binaries
patchelf --set-interpreter \
  "/usr/local/nix/lib/ld-linux-x86-64.so.2" \
  ./bcachefs-tools/bin/*
patchelf --set-rpath '$ORIGIN/../nix/lib' \
  ./bcachefs-tools/bin/*

# 3. patch the libs (point at each other; skip the baseline)
find ./libs/lib -name '*.so*' \
  -not -name '*ld-linux*' -not -name 'libc.so' \
  -not -name 'libgcc_s.so' -not -name 'libm.so' \
  -exec patchelf --set-rpath '$ORIGIN' {} \;
```

The result

```
$ ldd /usr/local/bin/bcachefs
  linux-vdso.so.1 (0x00007fc936270000)
  liburcu.so.8 => /usr/local/nix/lib/liburcu.so.8
  libsodium.so.26 => /usr/local/nix/lib/libsodium.so.26
  libz.so.1 => /usr/local/nix/lib/libz.so.1
  libfuse3.so.3 => /usr/local/nix/lib/libfuse3.so.3
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6            # system baseline
  /usr/local/nix/lib/ld-linux-x86-64.so.2                 # bundled loader
```

```
$ dpkg -i bcachefs-tools_1.13.0_amd64.deb
$ bcachefs version
1.13.0
```

Nix built it. patchelf made it portable. `nfpm` packaged it as a `.deb`.

The store path isn't a prison. It's a starting point.

Same tool. Two directions.

Surgery In — foreign binary → NixOS
Fix interpreter. Fix RPATH. Add needed libs.

Surgery Out — Nix binary → anywhere
Swap absolute store paths for `$ORIGIN`. Bundle deps. Ship it.

`patchelf` is just writing strings.
Once you see that, you start seeing the fields everywhere.

Where to go from here

github.com/noblepayne/nix-anywhere-else-talk

github.com/noblepayne/patched-python-for-nix

github.com/noblepayne/bcachefs-tools-portable

github.com/noblepayne/opencode-flake

readelf · ldd · patchelf

Questions?