TRAIL OF BITS

it’s 6PM: do you know what your builds are doing?

William Woodruff

hello

- **william woodruff**
  - engineering director @ trail of bits
  - open source team, working primarily on OSS projects: LLVM, Homebrew, PyPI, pip-audit, etc.
  - @yossarian@infosec.exchange
- **Trail of Bits**
  - ~150 person cybersecurity auditing and engineering consultancy
  - specialities: cryptography, compilers, program analysis, “supply chain”, general high assurance software development

TRAIL OF BITS

let’s talk about build systems

- **(almost) nobody likes build systems**
  - flakey, crufty, fragile, typically break when you need them, indeterminate build states, etc.
- **(almost) nobody uses build systems correctly**
  - typical development flow for build system engineering is to hit it until it runs locally
  - ...then hit it some more on each other machine it needs to run on
- **billions of dollars in ~~fake money~~ VC capital spent on killing build systems**
  - so far this has resulted in more build systems

why are build systems so hard and opaque?

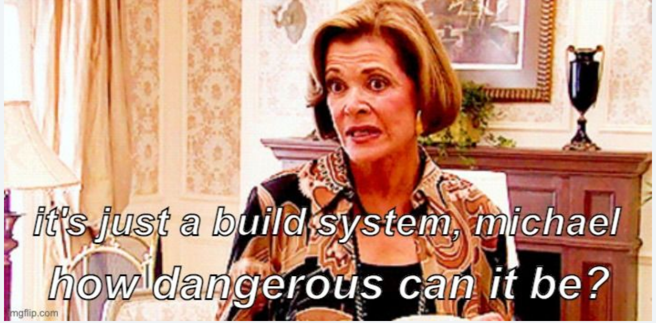
- ***conceptually***: build systems are pure functions that map an input space (sources, dependencies, etc.) into an output space (binaries, tarballs, etc.)
- ***in reality***: build systems mash a bunch of mutable state around in a shared global namespace (the filesystem)
  - programmers care about performance, so we throw multiprocessing in there for good measure
- ***tools themselves don't help***: native toolchains have global header, linker, etc. paths, all kinds of special "escape hatches" to help stressed engineers get around their problems and back to work

sometimes we want to instrument builds

- …for ***caching***: we want to instrument builds to replace or cache steps that don’t need to be repeated
    - See: ccache, sccache
- …for ***profiling***: we want to identify the slowest parts of the build for refactoring, replacing, or caching (see above)
- …for ***rewriting:*** we want to programmatically modify the build’s behavior (e.g. debug info, opt level) without playing find-the-flag in 16 different Makefiles
- …for ***static analysis***: we want to programmatically modify the program itself to make it more amenable to analysis (e.g. rewriting the source on each step)
- …for ***security:*** we want to see if the build itself is vulnerable or makes its outputs vulnerable

build system instrumentation

…for ***security?***

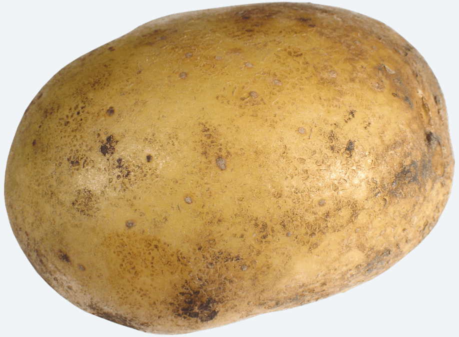


- **builds can produce insecure programs, especially in non-obvious ways:**
  - what's wrong with this flag? `-DFORTIFY_SOURCE=2`
    → `-D_FORTIFY_SOURCE=2`
  - what's wrong with this line? `-Wall @extra.txt`
    → `-Wall -w` (expanded from `extra.txt`, helpfully added by your build engineer)
- **builds can produce *contextually incorrect* programs:**
  - release builds containing debug symbols (ask MS how they feel about this one)
- **builds can *themselves* be insecure and open to manipulation**
  - "helpfully" pulling dependencies from the 'net, enabling local features + unsafe configurations in production, …
- **we'd like to be able to detect these kinds of flaws and weaknesses automatically, and ~~exploit~~ prevent them**

build instrumentation: how hard could it be?

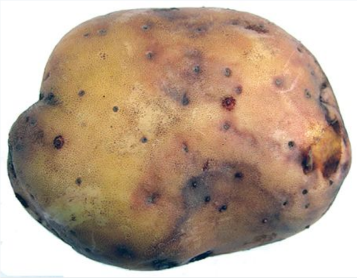
- **large diversity of build and “metabuild” systems**
  - Make, CMake, Bazel, Cargo, Docker, your coworker’s bash scripts
- **large diversity of compiler and tool frontends**
  - `clang`, GCC, MSVC, ICC, wrappers around `ld`, etc.
  - each has a large CLI with complex argument semantics for reasons™
- **large diversity of compiler-adjacent tooling**
  - lots of builds directly invoke `cpp`, `as`, `ar`, `ld`, `install`, `strip`, etc.
    - each of these (again) has a large and poorly-defined CLI
- **each of these needs to be modeled *precisely* and *accurately*, because build systems *will* use them to their weirdest extents**
  - did you know that you can `-D “FOO(X)=X + 1”`?
  - or: `clang++ -x c lol.c -x c++ lmao.cpp`

build instrumentation with blight



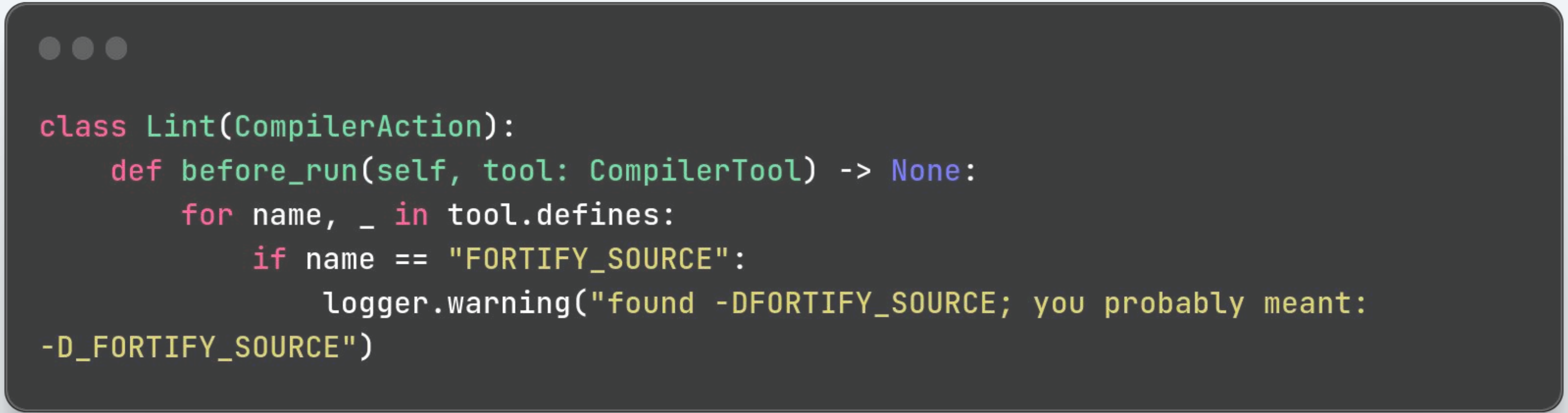
- `blight` **is a framework (and CLI tool) we wrote for instrumenting *arbitrary* build systems**
  - (meta)-build agnostic: doesn't care how it's run (as long as you run it)
- **minimally invasive: no funny business with `strace` or `LD_PRELOAD`**
  - contrast: [`bear`]() (`LD_PRELOAD`) and [`build-bom`]() (`strace`)
- **high-fidelity models of each "standard" build tool**
  - `CC`, `CXX`, `AS`, `AR`, etc.
- **a high level "actions" API for arbitrary instrumentation**
  - e.g. "each time the build invokes `CC`, replace it with `CXX`"
  - batteries-included actions for profiling, recording, basic rewrites

taming misbehaving builds



- **nice builds: ones that inherit `$CC`, etc., from the environment, or allow environment overrides**
  - modern-ish build and metabuild systems (like CMake, Meson)
- **not so nice builds: ones that hardcode `gcc`, `clang`, etc.**
  - lots of handwritten Makefiles
  - `blight` does "$PATH swizzling" to place fake `gcc`, etc. shims on `$PATH`
    - i made this term up because i didn't know what to call it
- **very naughty builds: hardcoding or code-genning `gcc-X`, `clang-X`, etc.**
  - lots of `build.sh` and autoconf stuff
  - `blight` can detect most of these, but some might slip through
- **unworkable: builds that hardcode the whole path (e.g. `/usr/bin/gcc`)**
  - there's very little we can do about this without being more invasive; thankfully they aren't very common

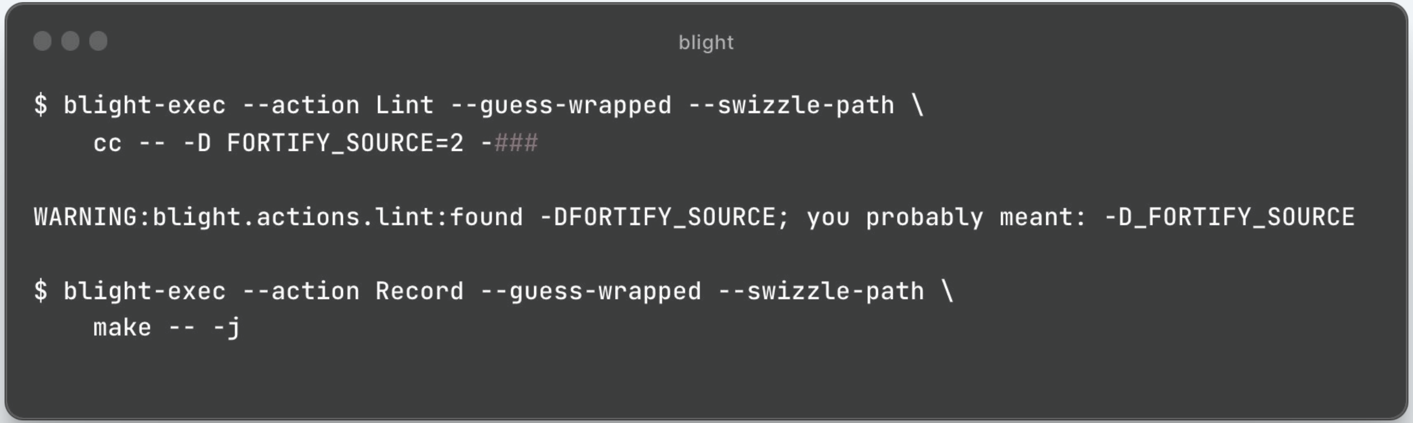
build instrumentation with blight



```
class Lint(CompilerAction):
    def before_run(self, tool: CompilerTool) -> None:
        for name, _ in tool.defines:
            if name == "FORTIFY_SOURCE":
                logger.warning("found -DFORTIFY_SOURCE; you probably meant:
-D_FORTIFY_SOURCE")
```

build instrumentation with blight

```
python -m pip install blight
```



```
$ blight-exec --action Lint --guess-wrapped --swizzle-path \
    cc -- -D FORTIFY_SOURCE=2 -###

WARNING:blight.actions.lint:found -DFORTIFY_SOURCE; you probably meant: -D_FORTIFY_SOURCE

$ blight-exec --action Record --guess-wrapped --swizzle-path \
    make -- -j
```

wrapup

concluding thoughts

- **anything can be a program analysis/instrumentation problem if you try hard enough**
- **we built `blight` to solve a single problem on a research project, but it's generic enough to be a building block for all kinds of build instrumentation tools. some ideas:**
  - a tracking/burndown progress meter for builds that don't natively support progress
  - a bitcode/IR collection layer a la WLLVM/GLLVM
- **small QoL API features make annoying analysis tasks less annoying**
  - `blight` does a ludicrous amount of modeling and wrapping so that users don't have to handle `@file` or `-DFOO -UFOO -DFOO`

the end of the talk

thanks!

these slides will be available at:

https://yossarian.net/publications#osiris-2023

links:

GitHub: trailofbits/blight

Blog post: High-fidelity build instrumentation with blight

Docs: trailofbits.github.io/blight

PyPI: pypi.org/p/blight