The Illusion of Control: Trusting Software in the Age of Compromise

Moin Rahman <moin@cybermancer.is>

Who Am I?

- Conference hopper & repeat offender
- Focused on FreeBSD & Networking
- Active contributor to FreeBSD (Packaging, Release Engineering, CI)
- Independent consultant helping enterprises with building secured infrastructure
- Part of the FreeBSD ecosystem but here representing myself & my work
- Believer in open-source resilience & long-term stability
- Occasionally break things in the name of security & innovation

The Early Days of DIY Builds

- Not just about FreeBSD—applies to UNIX/Linux distributions
- In the past, we built everything ourselves
- Examples: Gentoo, Linux From Scratch (LFS), BSDs
- Over time, control shifted away from users

The Birth of Linux & Distributions

- 1991 – Linus Torvalds releases the Linux kernel
- Needed userland utilities, package management, a full OS
- Early Linux was built from source by users
- Distributions emerged to simplify the process:
  - Slackware (1993), Debian (1993), Red Hat (1994), Gentoo (1999)
- Made Linux more accessible but reduced user control

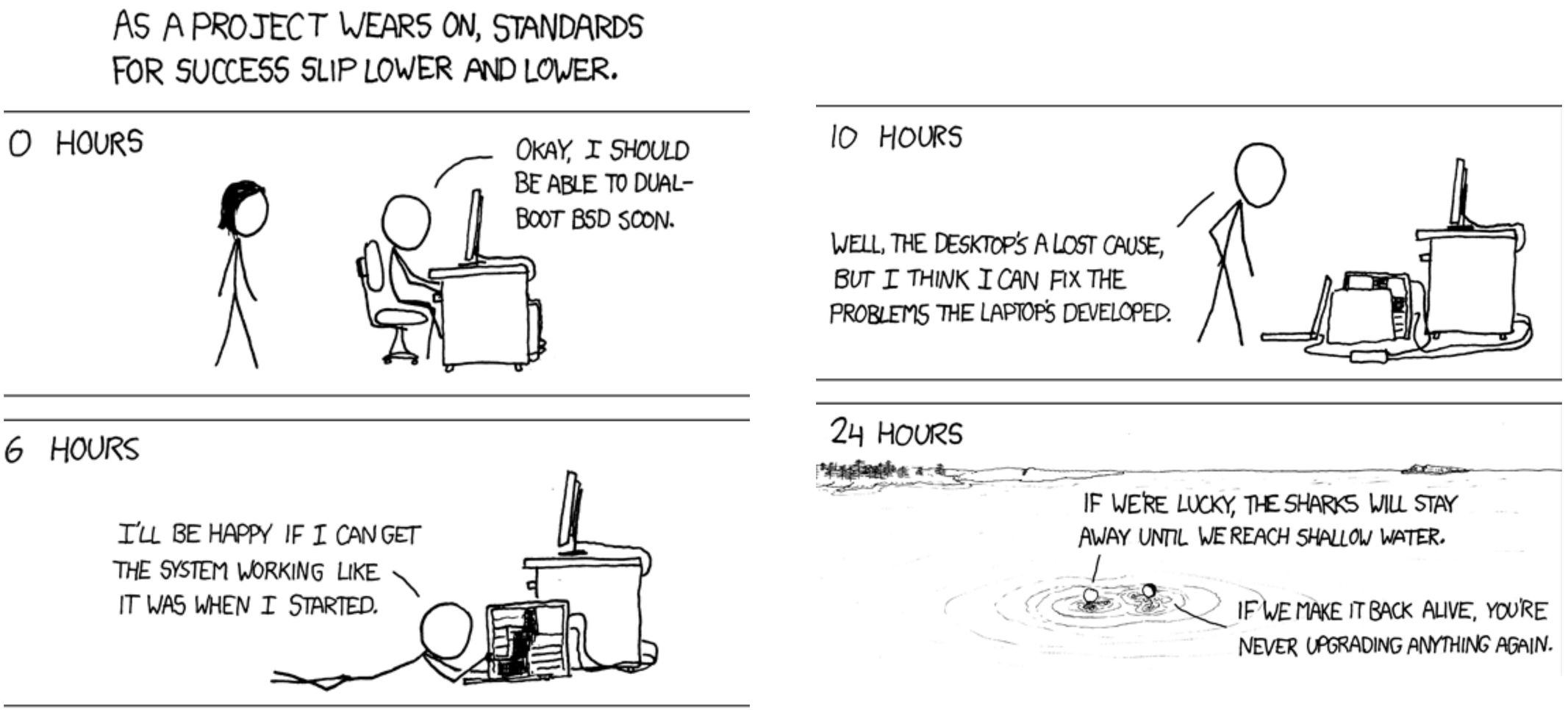
Linux From Scratch: The DIY Approach

- 1999 – LFS created as a manual Linux build guide
- Why it mattered:
  - Compile everything from source
  - Learn how OS components fit together
  - You are the package manager
- Challenges:
  - Time-consuming, complex, not scalable

LFS: More Than Just DIY

- Educational value:
  - Teaches kernel-userland interactions
  - Understanding toolchains, linking, init systems
  - Explains why package managers exist
- Valuable for security & system knowledge

I Built LFS and Got an Existential Crisis

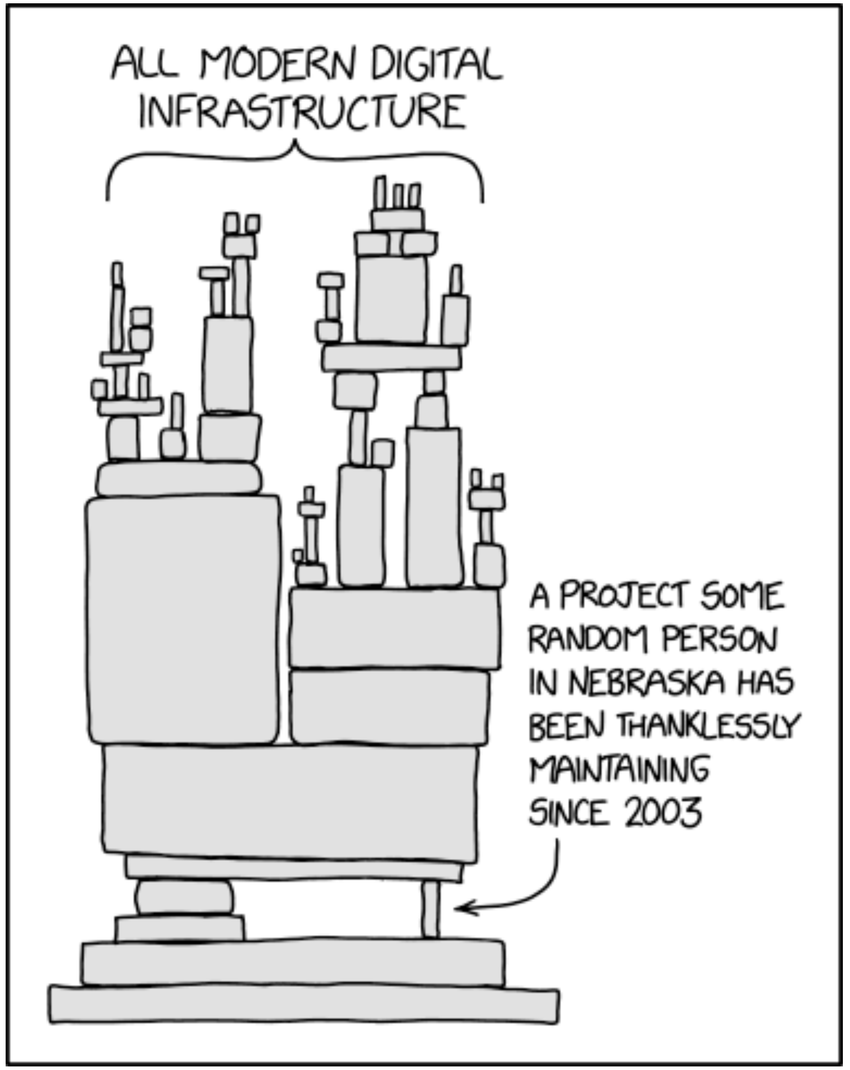


Why Enterprises Stopped DIY Builds

- Time & Complexity – Building from source isn’t scalable
- Reliance on Vendors – Prebuilt, “secure” distributions
- Package Managers Took Over – Convenience > Control
- Security Blind Spots – No transparency in builds

The Illusion of Control: Prebuilt Binaries Are a Risk

- Where do binaries come from?
  - Upstream maintainers?
  - Package repositories?
  - Automated CI/CD systems?
- Key issues:
  - No guarantee binaries match source code
  - Dependencies change silently
  - Recent attacks (SolarWinds, XZ Backdoor) prove the risk

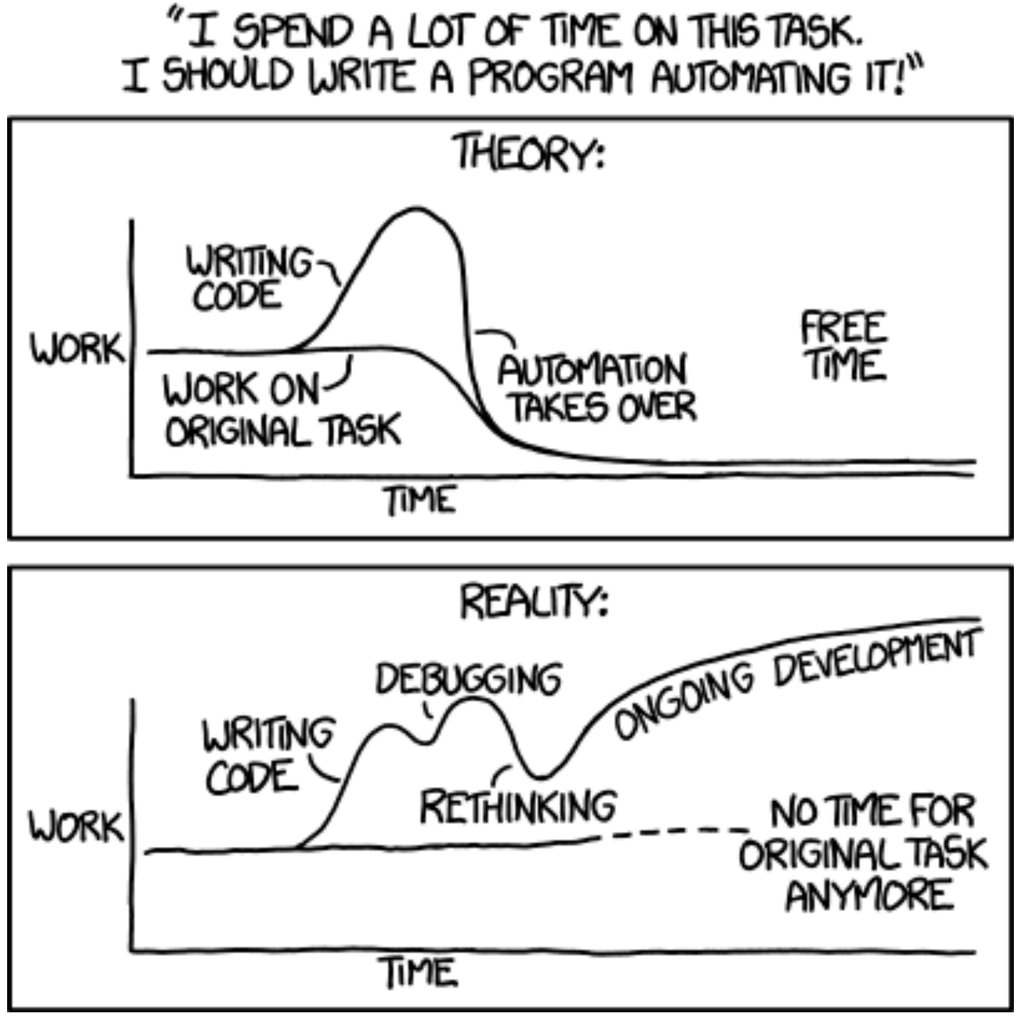


Does This Sound Like Zero Trust?

- Zero Trust Principles:
  - Never trust, always verify
  - Assume compromise
  - Reduce attack surface
- Reality:
  - Prebuilt binaries = Trusting someone else’s security
  - If you don’t control your build, you don’t control security

The Cultural Shift – From DIY to Blind Trust

- Why did this happen?
  - Package managers simplified updates
  - Security updates & automation prioritized over ownership
  - Enterprises chose efficiency over control
- Result:
  - Security is assumed, not verified
  - Enterprises trust vendor binaries without proof



We Need a Better Way – Enter Zero Trust Builds

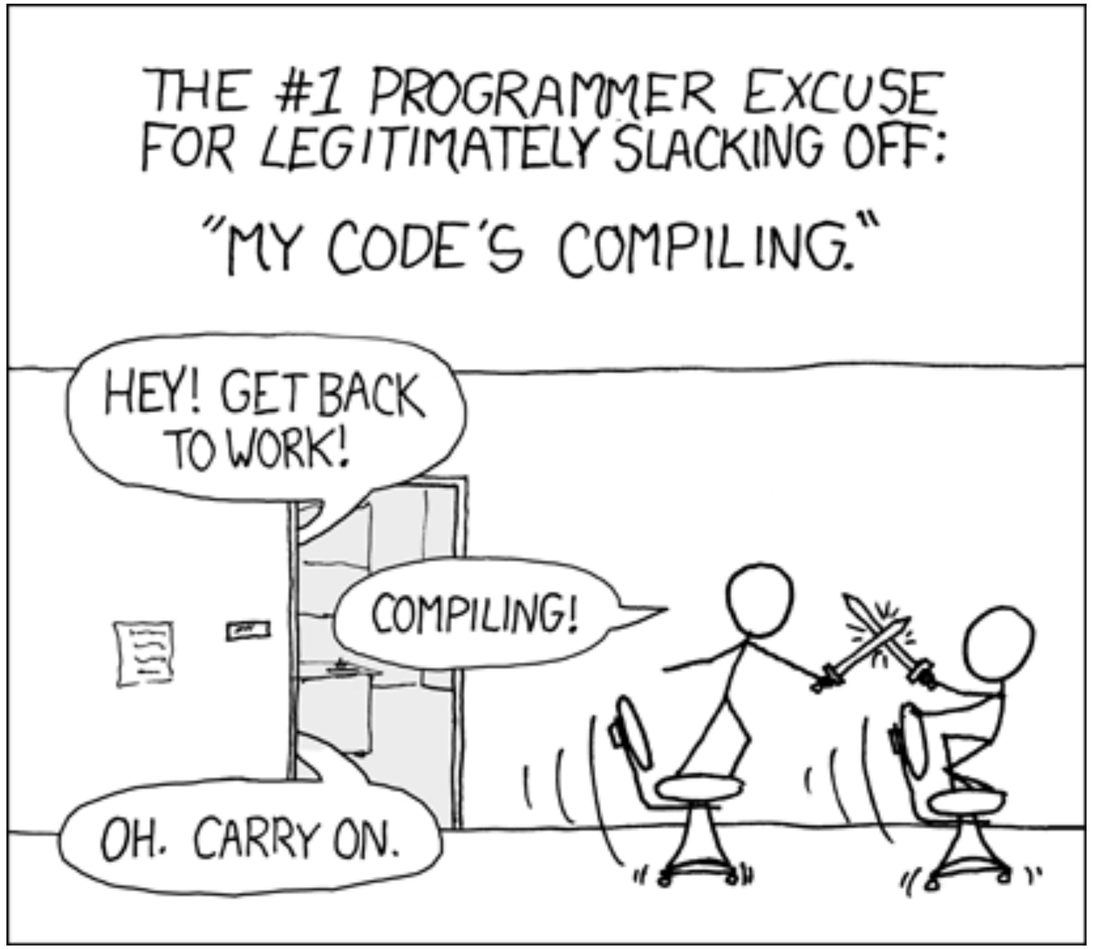
- Principles of Zero Trust Builds:
  - Verify everything deployed
  - Remove trust assumptions in artifacts
  - Control the full software supply chain

Traditional Model	Zero Trust Build Model
Trust vendor binaries	Own your build pipeline
Assume security is tested	Verify every step
Blindly accept updates	Audit & reproduce builds

How OS Build Systems Work Today

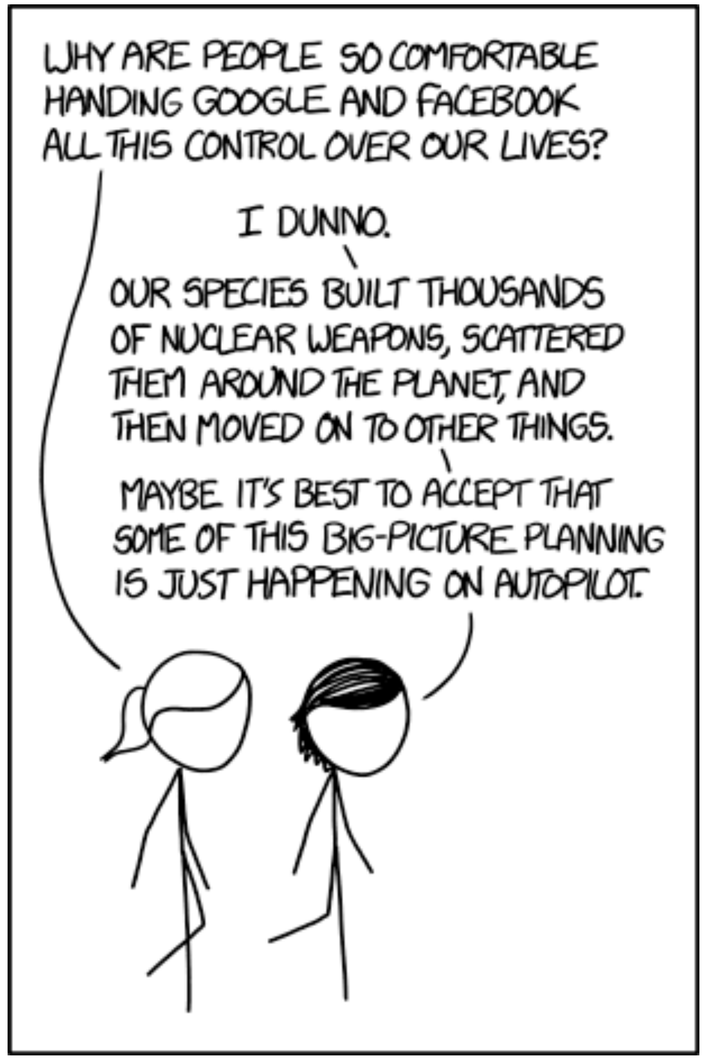
1. Source Code is Pulled
2. Automated CI/CD Builds
3. Packages & OS Components Are Generated
4. Installation Media & Artifacts Created
5. Distributed via Package Managers

- Issues: Opaque processes, silent dependency changes, attack vectors



Who Signs the Binaries?

- Key questions:
- Trust the source code or the compiled binaries?
- Recent Attacks:
  - SolarWinds (2020), XZ Backdoor (2024), CCleaner (2017)
- Reality:
  - Signed binaries ≠ Secure binaries
  - Without build control, you don’t own your security



Implementing Zero Trust in Software Builds

1. **Source Selection – Trusted repositories only**
2. Reproducible Builds – Deterministic output
3. Isolated Build Environments – Hardened pipelines
4. Cryptographic Verification – Sign & verify integrity

One Character Off

- https://github.com/freebsd
- https://github.com/free6sd
- One character difference - Full Pipeline Compromise
- Typo-Squatted URLs often go unnoticed
- Exploiting trust in URLs and human review fatigue
- Will the build system detect this?

Legendary 90s mischief

```
:(){ :|:& };:
```

Twenty First Century ForkBomb

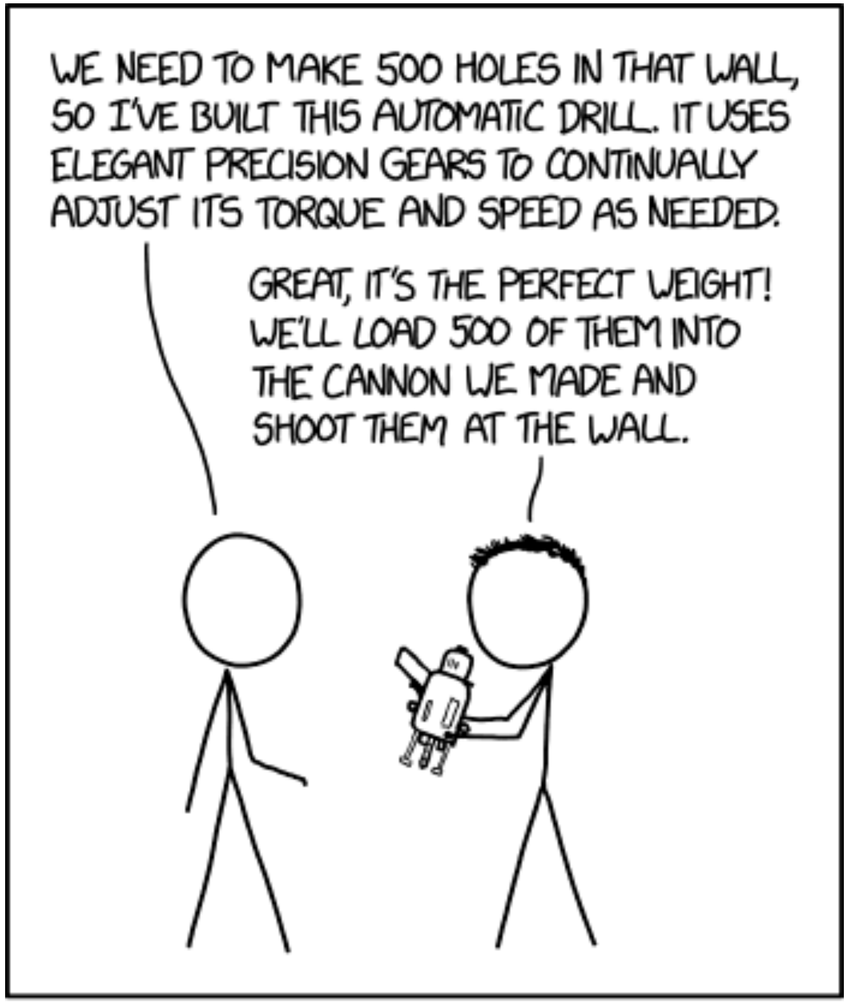
The Hidden Challenge

- Reproducibility is hard - even in Open Source
- Key challenges that break reproducibility:
  - Timestamp and Non-Deterministic Data
  - Out-of-order Compilation
  - Hardcoded Absolute Paths
  - Embedded Random Data (UUIDs, seeds)
  - Locale-Sensitive Builds
  - Uncontrolled Internet Dependencies

- The result
  - If upstream is not reproducible, neither can enterprise be
  - Creates blind spots in verifying software integrity

What Upstream Developer can do?

- Reproducibility starts upstream:
  - Normalize timestamps using SOURCE_DATE_EPOCH
  - Ensure deterministic build processes
  - Avoid hardcoded paths and hostnames
  - Minimize random data and set fixed seeds
  - Standardize locale and environment settings
  - Pin dependencies and avoid dynamic Internet pulls
  - Use tools like diffoscope for reproducibility checks
- Why it matters:
  - Strengthen trust in Open Source supply chain
  - Enables enterprises to verify builds confidently
  - Reduce the risk of hidden compromises in software stack



The Reproducible Builds Project

- Goal: Ensure that source code consistently produces identical binaries.
- Why It Matters:
  - Detects tampering or malicious modifications in the build process.
  - Builds trust by enabling independent verification of software artifacts.
- Key Challenges Addressed:
  - Timestamps, locale, and randomness affecting builds.
  - Dependency management to prevent upstream drift.
  - Promoting deterministic build environments across ecosystems.
- Regularly Testing Reproducibility In:
  - Debian, Arch Linux, FreeBSD, OpenBSD, Guix, NixOS, Alpine, Fedora, openSUSE

Next Steps for Enterprises

- Identify critical software dependencies
- Pick a key package & verify its build process
- Use reproducibility tools (Debian, FreeBSD, OpenBSD)
- Reduce reliance on external package repositories
- Treat build integrity as part of security
- **Zero Trust Builds don’t happen overnight—start small, improve continuously**

Final Thought – The Price of Convenience

- Security vs. Convenience is always a trade-off.
  - We used to build everything ourselves.
  - Then we trusted package managers.
  - Now we trust CI/CD pipelines and upstream maintainers.
- We used to build everything → Trusted package managers → Now trust CI/CD pipelines
- Where do we draw the line?
  - At some point, convenience stops being an optimization and starts being a risk.
  - Passion writes the code, but money decides its future—open or closed.
  - Even in open source, sustainability is built on financial reality.

Thank You & Q&A

- Let’s discuss:
  - What’s stopping you from verifying your own artifacts?
  - What steps can your organization take toward Zero Trust Builds?
  - What risks are acceptable in your software supply chain?