

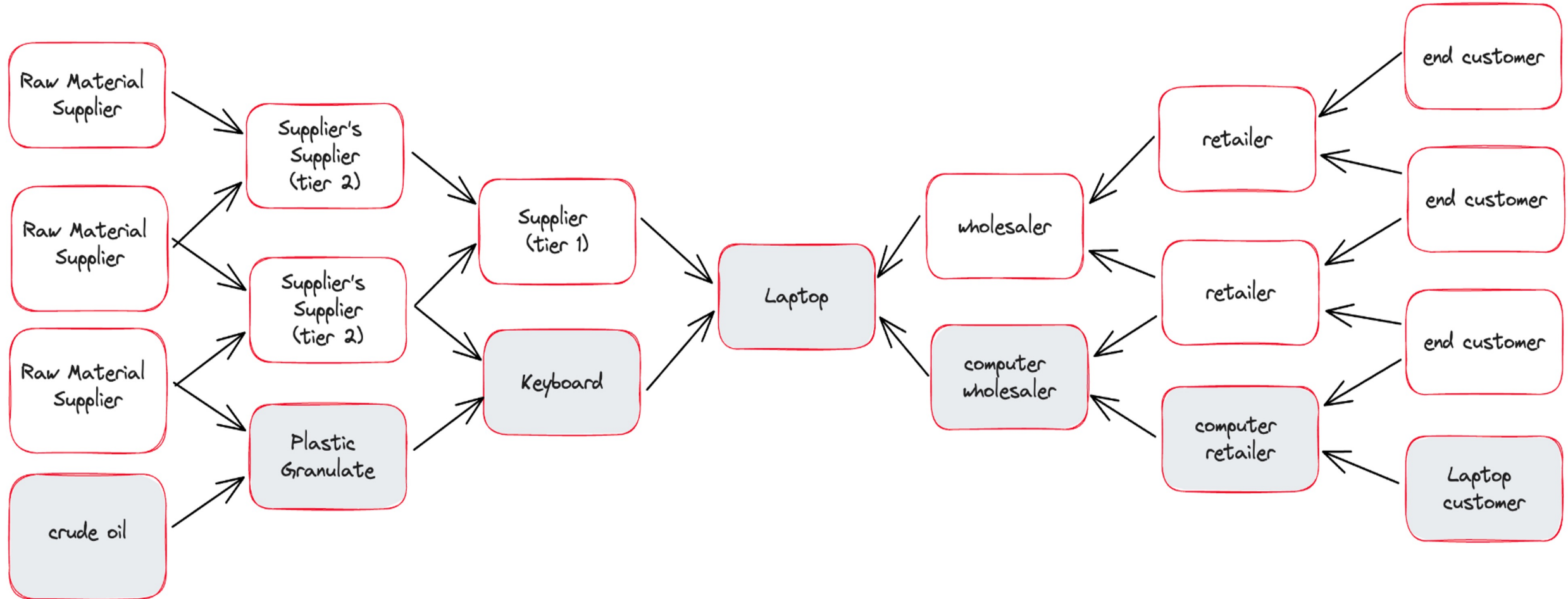
# Beyond the Code / SBOM

Supply Chain Security

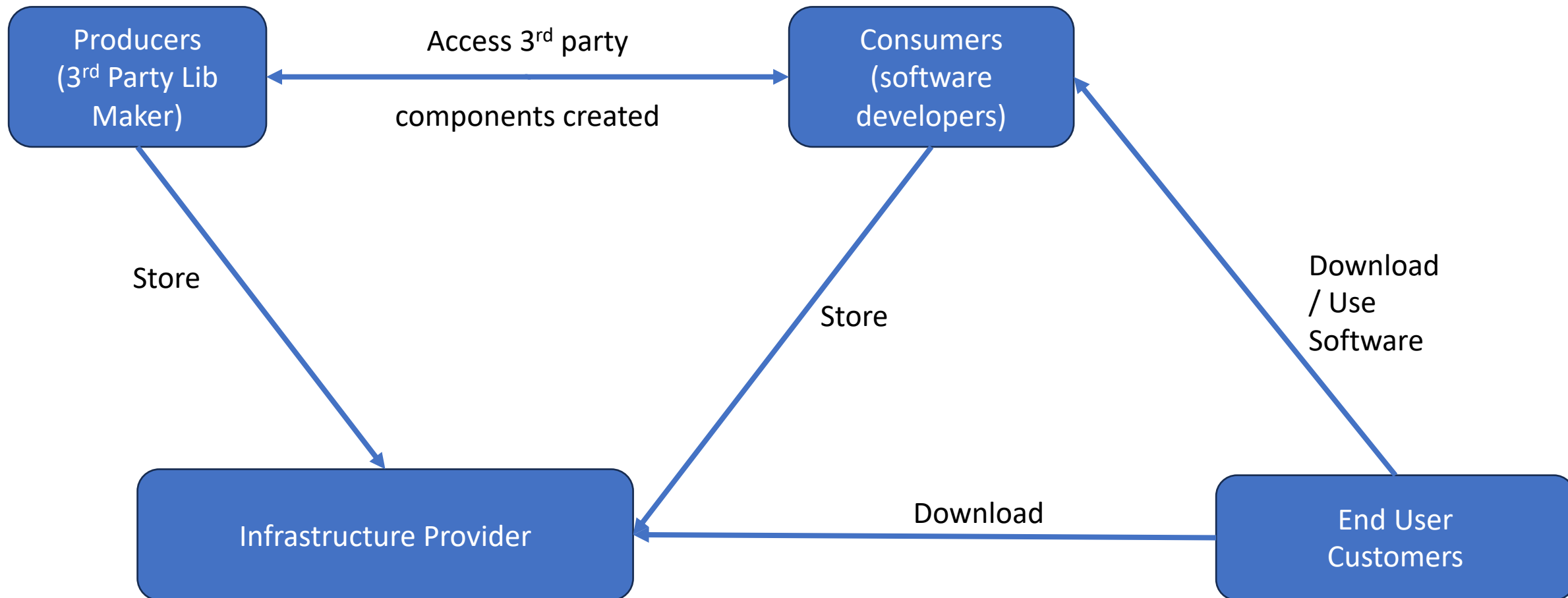
# Anant Shrivastava

- Chief researcher @ Cyfinoid Research
- 15+ yrs of corporate exposure
- **Speaker / Trainer:** BlackHat, c0c0n, nullcon, RootConf, RuxCon
- **Project Lead:**
  - Code Vigilant (Code Review Project)
  - Hacking Archives of India,
  - TamerPlatform (Android Security)
- (@anantshri on social platforms) <https://anantshri.info>

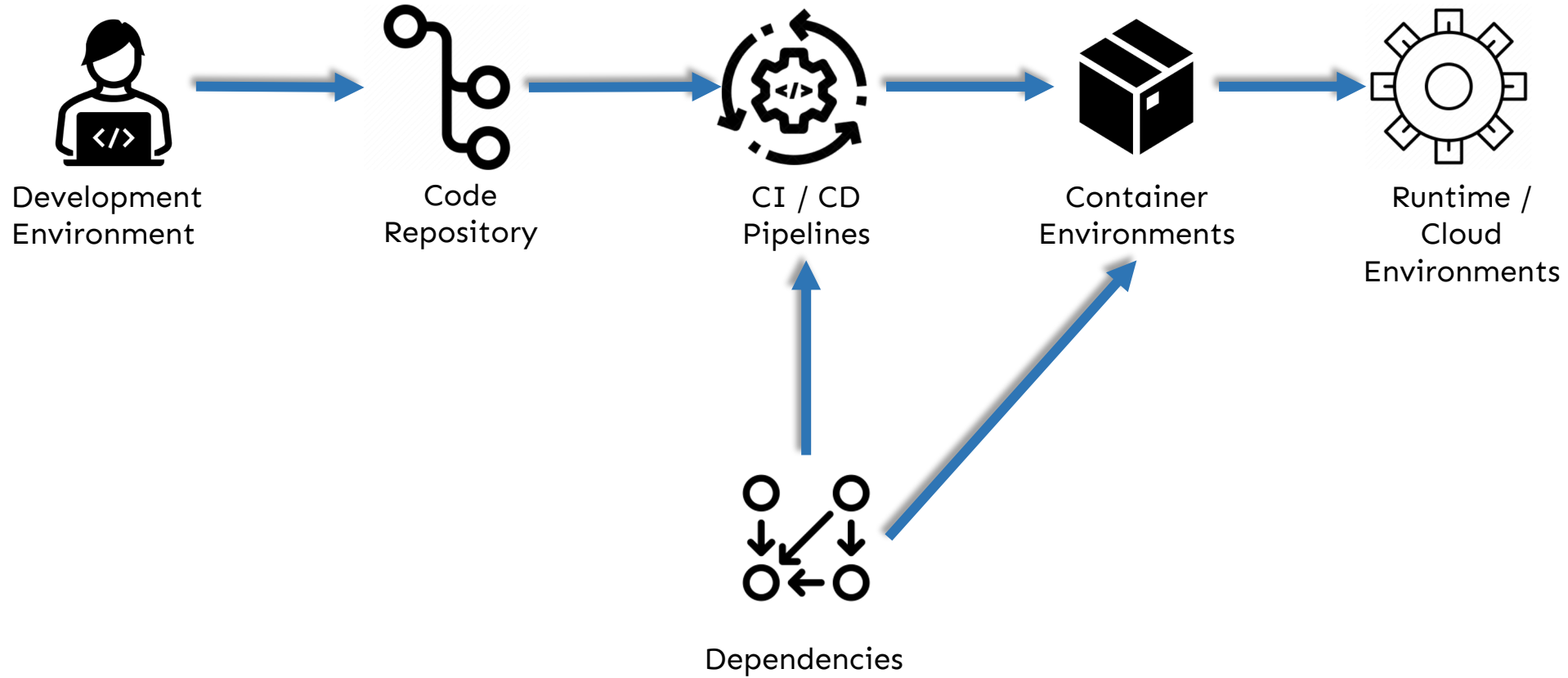
# Supply chain



# It's a chain like any other chain



# Software Supply chain



TURING AWARD LECTURE

# Reflections on Trusting Trust

*To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.*

# What can go wrong

---

Quick History Lesson

**KEN THOMPSON**

## INTRODUCTION

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX<sup>1</sup> swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bob-

programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

## STAGE I

1. I will first present to you the cutest program I ever wrote.

```

    {
        if(match(s, "pattern")) {
            compile("bug");
            return;
        }
        ...
    }

```

**FIGURE 3.2.**

```

compile(s)
char *s;
{
    if(match(s, "pattern1")) {
        compile ("bug1");
        return;
    }
    if(match(s, "pattern 2")) {
        compile ("bug 2");
        return;
    }
    ...
}

```

**FIGURE 3.3.**

**Acknowledgment.** I first read of the possibility of such a Trojan horse in an Air Force critique [4] of the security of an early implementation of Multics. I cannot find a more specific reference to this document. I would appreciate it if anyone who can supply this reference would let me know.

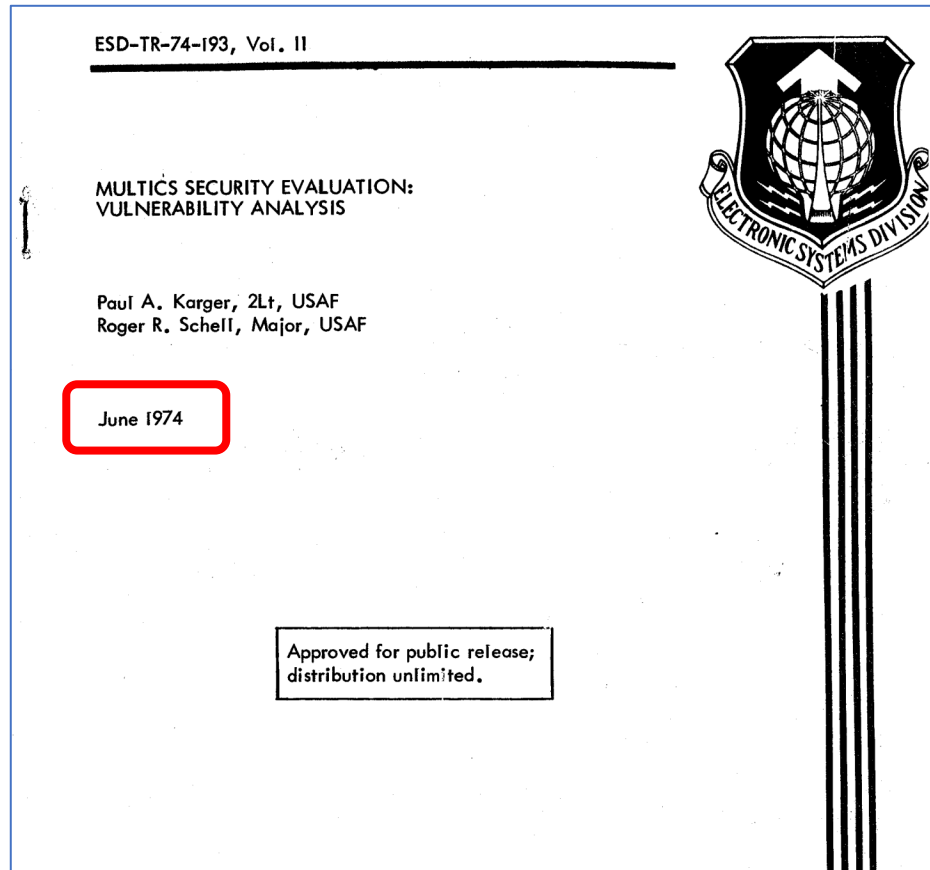
#### REFERENCES

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time-sharing system for the PDP-10. *Commun. ACM* 15, 3 (Mar. 1972), 135-143.
2. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Commun. ACM* 17, (July 1974), 365-375.
4. Unknown Air Force Document.

Author's Present Address: Ken Thompson, AT&T Bell Laboratories, Room 2C-519, 600 Mountain Ave., Murray Hill, NJ 07974.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

# First Reference (found so far)



It was noted above that while object code trap doors are invisible, they are vulnerable to recompilations. The compiler (or assembler) trap door is inserted to permit object code trap doors to survive even a complete recompilation of the entire system. In Multics, most of the ring 0 supervisor is written in PL/I. A penetrator could insert a trap door in the PL/I compiler to note when it is compiling a ring 0 module. Then the compiler would insert an object code trap door in the ring 0 module without listing the code in the listing. Since the PL/I compiler is itself written in PL/I, the trap door can maintain itself, even when the compiler is recompiled. (38) Compiler trap doors are significantly more complex than the other trap doors described here, because they require a detailed knowledge of the compiler design. However, they are quite practical to implement at a cost of perhaps five times the level shown in Section 3.5. It should be noted that even costs several hundred times larger than those shown here would be considered nominal to a foreign agent.



# Is it re-newed: lets Check OWASP

- OWASP Top 10 : 2021 : A06 : Vulnerable and outdated Components
- OWASP Top 10 : 2017 : A09 : Using Components of Known Vulnerability
- OWASP Top 10 : 2013 : A09 : Using Known Vulnerable Components
- OWASP Top 10 : 2010 : A06 : Security Misconfiguration
- OWASP Top 10 : 2007 : MISSING
- OWASP Top 10 : 2004 : A10 : Insecure Config Management

# So, what do we know

1. It's a known problem for many decades
2. So, what changed recently that it become a big thing

# Events

---

## Incidences

- SolarWind
- CodeCov
- Colonial Pipeline

## Resultant

- EO by US President
- NIST SSDF Framework
- SLSA by google

MAY 12, 2021

# Executive Order on Improving the Nation's Cybersecurity



› BRIEFING ROOM

› PRESIDENTIAL ACTIONS

By the authority vested in me as President by the Constitution and the laws of the United States of America, it is hereby ordered as follows:

Section 1. Policy. The United States faces persistent and increasingly sophisticated malicious cyber campaigns that threaten the public sector, the private sector, and ultimately the American people's security and privacy. The Federal Government must improve its efforts to identify, deter, protect against, detect, and respond to these actions and actors. The Federal Government must also carefully examine what occurred during any major cyber incident and apply lessons learned. But cybersecurity requires more than government action. Protecting our Nation from malicious cyber actors requires the Federal Government to partner with the private sector. The private sector must adapt to the continuously changing threat environment,

# Industry and buzz word brigade

- Software Bill of Material : Ledger of ingredients
- Provenance : Proof of authenticity of documents (namely sbom)

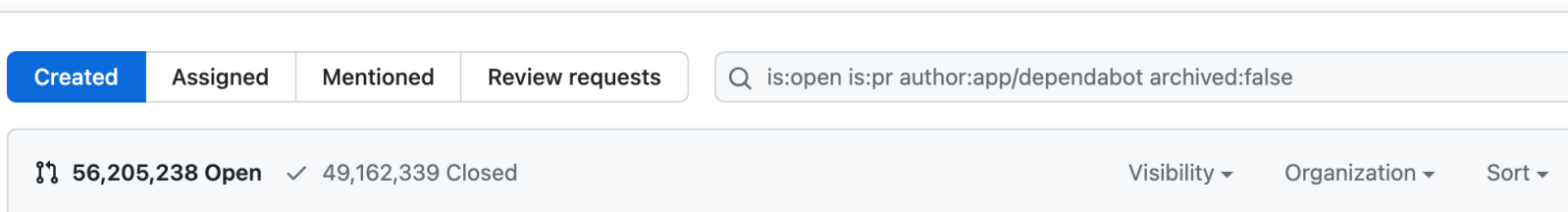
# Funny thought

Most of the times


software industry is fixing problems,  
that are created by software industry


# What problems have we created

- Software build automation == quicker release cycle
- Automated release cycle == less wait for features
- Faster feature release == less inclination to upgrade dependencies
- Too much focus on OSS Codebase without helping the maintainers
- Impossible segregation of features and bug fixes
- Automated notification of vulnerability (hedonic hamster wheel)



# Example


 Closed

**Rapid release cycles** 

anantshri opened this issue on Oct 18 · 1 comment

---

We generally release whenever we have changes ready, as long as there aren't other changes that we expect to land immediately afterwards. There's not a particularly strong reason to have a feature ready and *not* make it available to users.

As an ahead-of-time preprocessor that doesn't have any built-in IO facilities  isn't likely to have security vulnerabilities. In the extremely unlikely case that it does, we'll release fixes as part of the standard release cycle. We don't have the resources to maintain multiple branches of the codebase over time.

I personally recommend that projects stay up-to-date using dependabot or similar tools. We try to avoid breaking changes whenever possible, so most releases will be fully compatible with previous ones. When breaking changes are unavoidable for CSS compatibility reasons, we produce deprecation warnings well before the breakage, so staying up to date will ensure you see deprecations in time to apply fixes before you're broken.

- Just to clarify they are not wrong in their assumption.
- I am giving example of where we are, not what's right or wrong.

# Business

So, the industry now is in the business of:

- Finding and reporting as many 3<sup>rd</sup> party dependency issues as possible
- Creating inventory
- Validating inventory signatures
- & Feeling good about ourselves that we have saved the city





SBoM (Software Bill of Material)

# NIST Cyber Security Framework 1.1

---

- **Identify** : Know your assets
- Protect: Establish Baseline protection
- Detect : Identify attacks
- Respond : evict, monitor or respond
- Recover : Get back on feet



# Software Bill of Material

- Itemized list of all the *ingredients* in the software
- Ingredients means mostly third-party components
  - Software name
  - Version
  - Checksum
  - License information
  - Dependencies list if possible
- SBoM's are mostly for one level depth only with other levels plugged in each others.

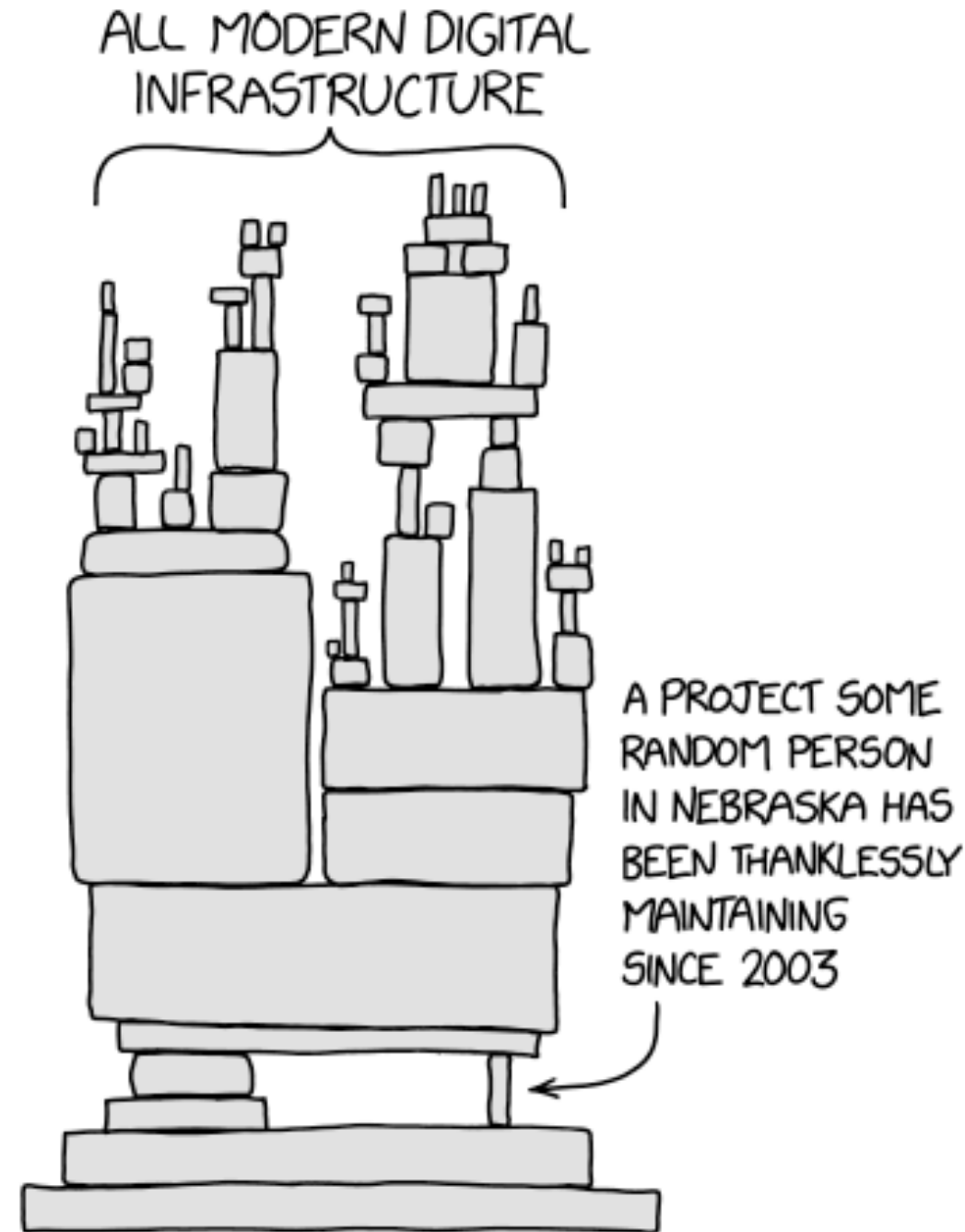
# What does it miss?

- Compiler used
  - ML models in use
  - SaaS services used
  - Operating environment (OS etc)
  - And much more
- 
- Shout to OWASP CycloneDX project they are doing some good work

# SBoM can help

- Identifying incorrect use of software
- Identify what to fix in scenarios like log4shell
- Identify impact in sec bug release in a core component
- Basically, Inventory problems

Ref: XKCD.com/2347



# SBoM can't help

- Identify codecov incidence (creds from docker image)
- Identify solarwind instance (build system hacked)
- Anything not dealing with third party code inventory
- It can't spot if the bug is actually in use in your code

# SBoM the Holy Grail

- Its not the be all, end all solution.
- IT'S the good first step.
- One thing that this industry failed all the time.

# Has the world done nothing so far

- There are two approaches taken before 2021 around this.
- Centralized approach (vetted software)
  - Central body takes care of security for packages
    - Debian or linux distro's
- De-Centralized Approach (isolated software)
  - Dev's maintain their own setup but isolate from system
    - Python venv
    - Homebrew (to some extent)
    - Npm



# Vetting Software

- Software gets vetted by the team of dedicated volunteer's
- Volunteer's take ownership of keeping up with software
- All updates are marked as feature or security.
- Separate branches maintained to handle feature or bug fix
- Stability over new features

# Vetted Software approach

## Pro's

- Single authority tracking bugs
- Stability over new features
- Volunteers have deeper understanding of software
- Centralized upgrade

## Con's

- Limited tracking capability
- Features don't reach end user
- Delay in case volunteer is unavailable

# Isolated Software Approach

- End users or consumers directly use the software's.
- Faster and quicker upgrades
- but more importantly controlled upgrades
- Software Isolated in environment to not conflict with base items

# Isolated approach

## Pro's

- Dev's have full control of package versions
- Packages localized so different project can work on different version of packages

## Con's

- In a log4jam or log4shell scenario
- No centralized update so security fix push is hard

# Modern Problem : Modern Solutions

- After decades of software development, we now have
  - Infrastructure as code
  - Better cryptographic standards
  - Hardware, software and storage capabilities to maintain provenances

# Provenance

- The proof of authenticity of the document
- Let me explain in negative
  - If a software download link is for example.com domain
  - The checksum of the downloadable file is stored at example.com domain
  - Is it safe?

# Provenance

- Ideally the provenance record and or document should be kept in two different places.
- At the very least the signing should be done in such a way that third party needs to be called to change anything

# So, what can we do?

Enough of discussion where do we go from here

- Frameworks
- Tooling



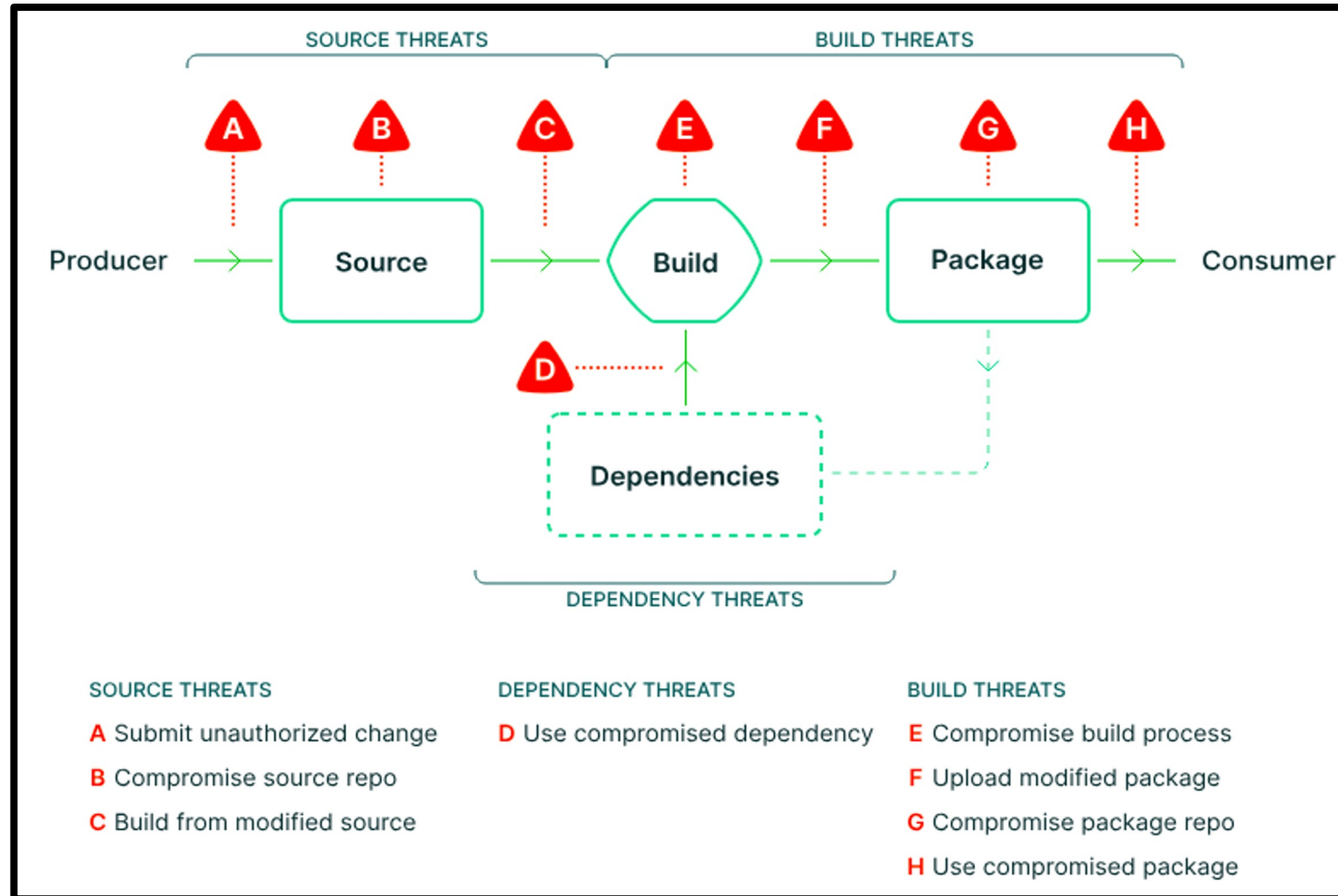
# Frameworks

- **SLSA Framework** is a good approach for producers and consumers
- **Cyber Security Framework v2** has govern section for end users and consumers
- **SSDF**: is guideline for all software developers (producers and consumers)

# SLSA Framework

- Effort by google to create a framework
- Different levels to represent current maturity levels
- Started with a lot of ambition in v0.1
- With v1.0 toned down to level Level 4 for future edition
  
- SLSA is nontransitive : Only assurance of current software not its transitive dependencies

# SLSA Framework



# SLSA Levels

Level	Description	Example
1	Documentation of the build process	Unsigned provenance
2	Tamper resistance of the build service	Hosted source/build, signed provenance
3	Extra resistance to specific threats	Security controls on host, non-falsifiable provenance
4	Highest levels of confidence and trust	Two-party review + hermetic builds

# NIST SSDF Framework

SSDF consists of following 4 section

- **Prepare the Organization (PO):** people, processes, and technology are prepared to perform secure software development at the organization level.
- **Protect the Software (PS):** Organizations should protect all components of their software from tampering and unauthorized access.
- **Produce Well-Secured Software (PW):** Organizations should produce well-secured software with minimal security vulnerabilities in its releases.
- **Respond to Vulnerabilities (RV):** Organizations should identify residual vulnerabilities in their software releases and respond appropriately to address those vulnerabilities and prevent similar ones from occurring in the future.

# Tooling

- <https://github.com/ossf/scorecard> : OSS health check
- <https://www.sigstore.dev/> : Distribution signing
- <https://github.com/sigstore/cosign> : container Signing
  
- <https://github.com/safedep/vet> : Org level package vetting + rules
- <https://stacklok.com/> : VSCode plugin health check + alternative

# Questions for Vendors

- What do I do with SBoM?
- If you report bugs, do you report based on reachability analysis (can you test it)
- Do you have alternative suggestions for the vulnerable library.

NAME WEBSITE

anant@anantshri.info

EMAIL

A diagram illustrating the components of the email address 'anant@anantshri.info'. The text is centered. Above the text, two blue brackets are positioned. The first bracket spans the word 'anant' and is labeled 'NAME' above it. The second bracket spans the '@anantshri.info' portion and is labeled 'WEBSITE' above it. Below the text, a single blue bracket spans the entire address and is labeled 'EMAIL' below it.