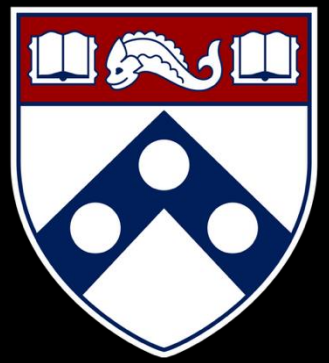
Secure Systems Engineering and Management

A Data-driven Approach

Michael Hicks



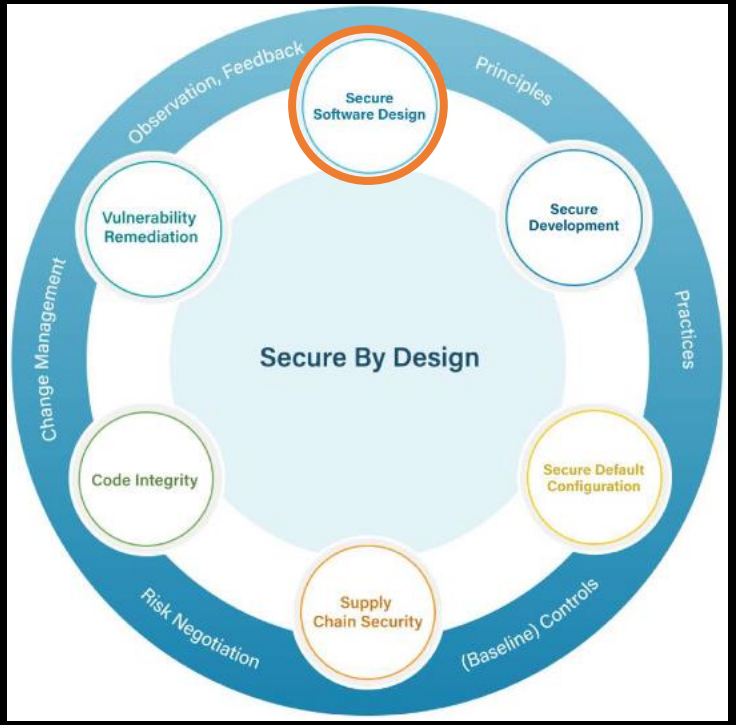
Managing Secure Software in an Organization

Supply Chain Security, Code Integrity, Vulnerability Remediation, and SBOMs

UPenn CIS 7000-003
Spring 2026

Security by Design

1. Secure Software Design
2. Secure Development
3. Secure Default Configuration

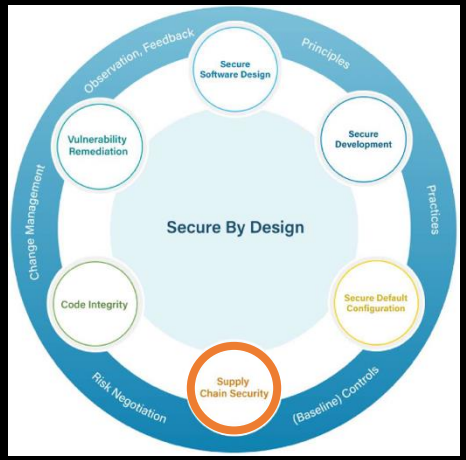


4. Supply Chain Security
5. Code Integrity
6. Vulnerability Remediation

Today's lecture

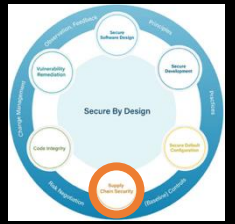
Why Management Matters

- Perfect security is impossible — the goal is to **manage risk**
- “Secure” is not a finished state — it is a **continuum** of components, composition, and operational practice
- Security must be an **ongoing organizational commitment**, not a one-time development activity



Supply Chain Security

What’s in your software?

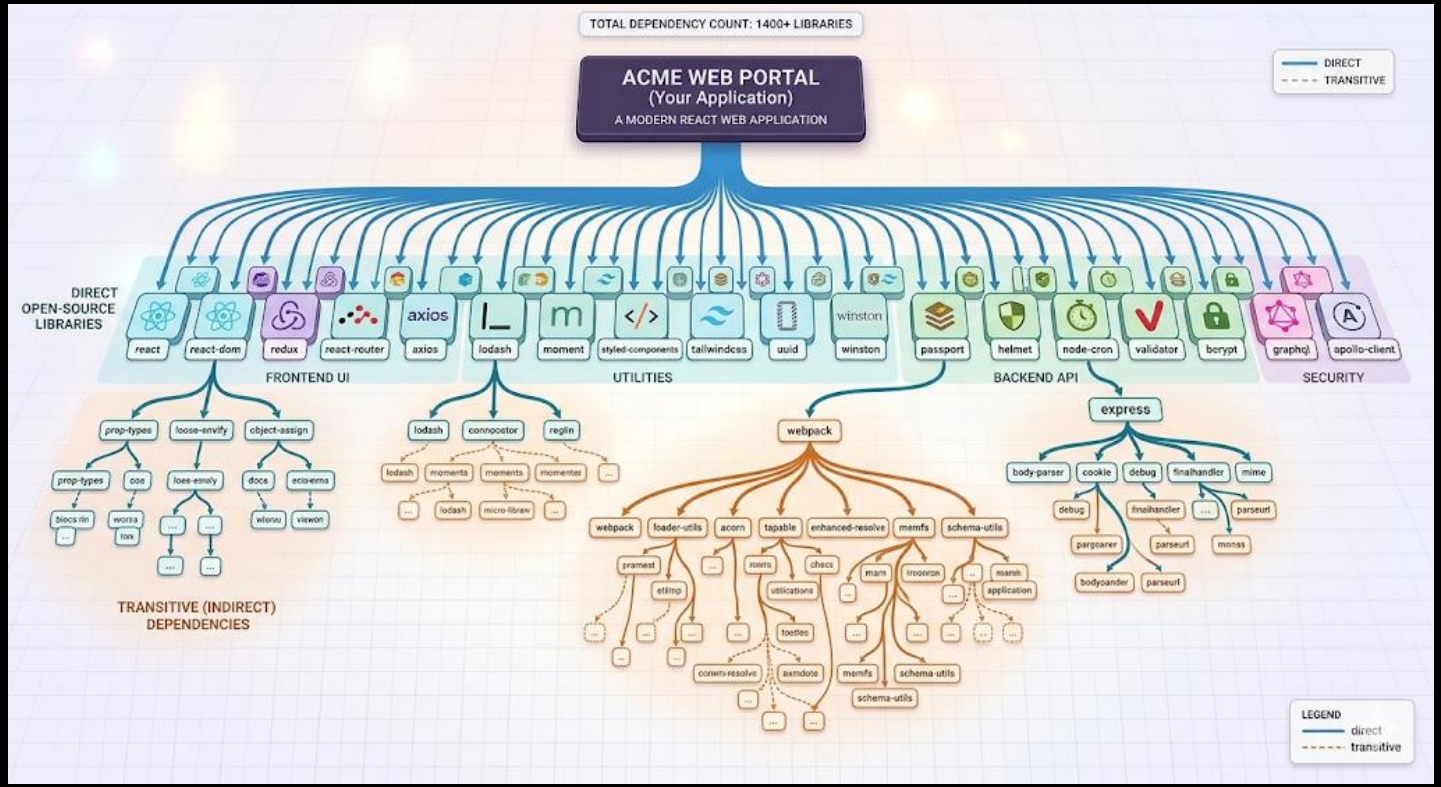


Modern software is **mostly code you didn’t write**.

Each dependency is a **trust decision**:

- Is it actively maintained?
- Has it been audited for security?
- What happens when a vulnerability is found in it?

A Modern Software Application



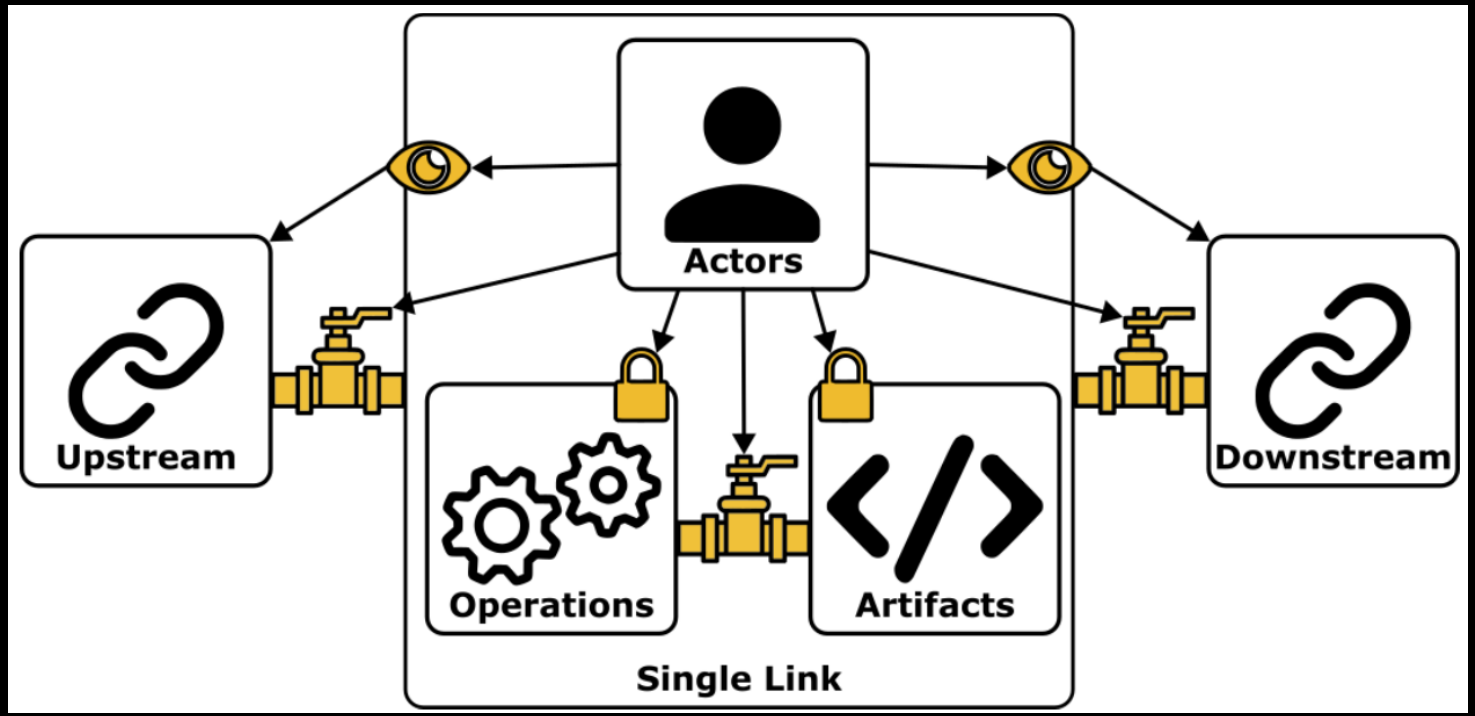
Supply Chain Challenge: Log4Shell

December 2021: CVE-2021-44228 in Apache Log4j

- A logging library used by thousands of Java applications
- Remote code execution via a simple log message
- Many organizations didn’t know they were using it

Key lesson: You must know what’s in your software so you can respond when a dependency is compromised.

Supply Chain Structure



Trust is Transitive

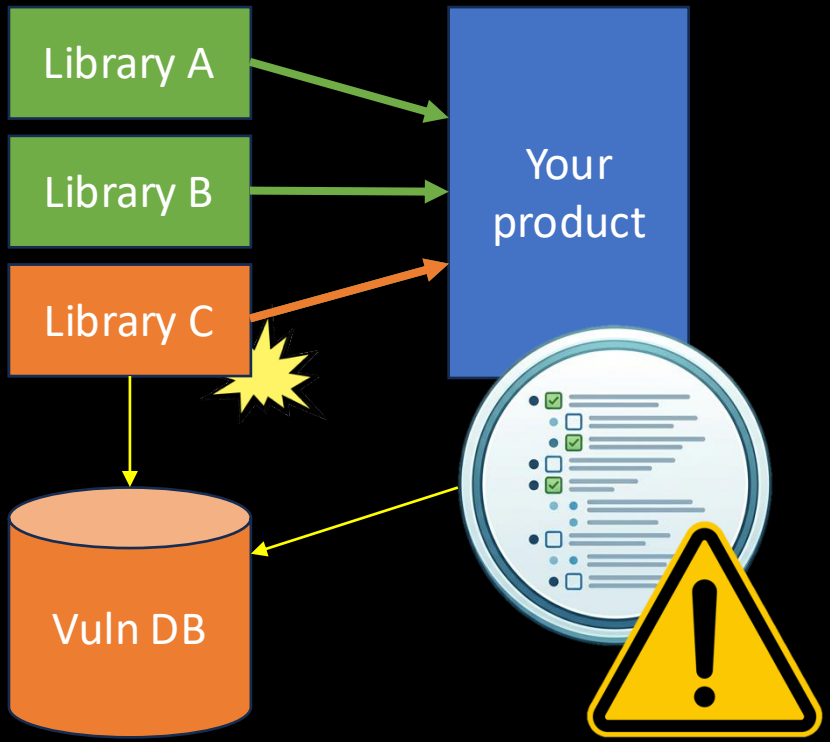
- You inherit the **security posture** of every dependency
- Your dependencies have dependencies — all the way down
- A vulnerability in *any* transitive dependency is a vulnerability in *your* product

Tracking Trust: Software Bill of Materials (SBOM)

A **machine-readable inventory** of the components of a software product

- Component name, version, supplier
- Relationship information (dependency graph)
- Generated at **build time**, updated with every release

Reducing Vulnerability Exposure with SBOMs

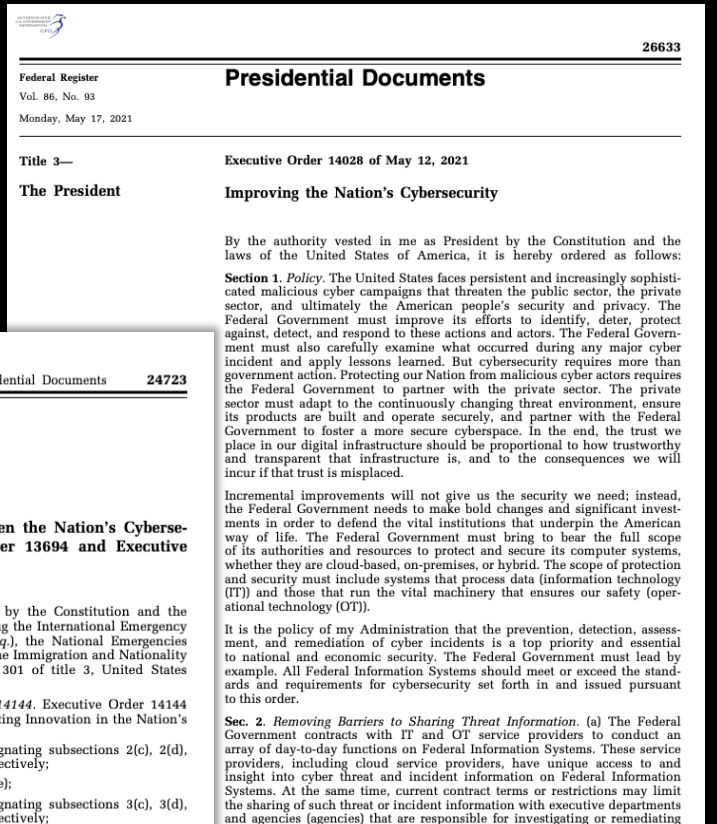


1. Vulnerability discovered
2. Vulnerability reported, logged to management DB
3. Local scanner discovers dependency is vulnerable
4. Alerts developer to take action

SBOMs are Now Legally Required

U.S. Executive Order 14028 (2021, sustained by EO 14306 in 2025):

SBOMs required for federal software procurement



26633

Federal Register
Vol. 86, No. 93
Monday, May 17, 2021

Presidential Documents

Title 3—

The President

Executive Order 14028 of May 12, 2021

Improving the Nation's Cybersecurity

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, ensure its products are built and operate securely, and partner with the Federal Government to foster a more secure cyberspace. In the end, the trust we place in our digital infrastructure should be proportional to how trustworthy and transparent that infrastructure is, and to the consequences we will incur if that trust is misplaced.

Incremental improvements will not give us the security we need; instead, the Federal Government needs to make bold changes and significant investments in order to defend the vital institutions that underpin the American way of life. The Federal Government must bring to bear the full scope of its authorities and resources to protect and secure its computer systems, whether they are cloud-based, on-premises, or hybrid. The scope of protection and security must include systems that process data (information technology (IT)) and those that run the vital machinery that ensures our safety (operational technology (OT)).

It is the policy of my Administration that the prevention, detection, assessment, and remediation of cyber incidents is a top priority and essential to national and economic security. The Federal Government must lead by example. All Federal Information Systems should meet or exceed the standards and requirements for cybersecurity set forth in and issued pursuant to this order.

Sec. 2. *Removing Barriers to Sharing Threat Information*. (a) The Federal Government contracts with IT and OT service providers to conduct an array of day-to-day functions on Federal Information Systems. These service providers, including cloud service providers, have unique access to and insight into cyber threat and incident information on Federal Information Systems. At the same time, current contract terms or restrictions may limit the sharing of such threat or incident information with executive departments and agencies (agencies) that are responsible for investigating or remediating

Federal Register / Vol. 90, No. 111 / Wednesday, June 11, 2025 / Presidential Documents 24723

Presidential Documents

Executive Order 14306 of June 6, 2025

Sustaining Select Efforts To Strengthen the Nation's Cybersecurity and Amending Executive Order 13694 and Executive Order 14144

By the authority vested in me as President by the Constitution and the laws of the United States of America, including the International Emergency Economic Powers Act (50 U.S.C. 1701 *et seq.*), the National Emergencies Act (50 U.S.C. 1601 *et seq.*), section 212(f) of the Immigration and Nationality Act of 1952 (8 U.S.C. 1182(f)), and section 301 of title 3, United States Code, it is hereby ordered:

Section 1. *Amendments to Executive Order 14144*. Executive Order 14144 of January 16, 2025 (Strengthening and Promoting Innovation in the Nation's Cybersecurity), is hereby amended by:

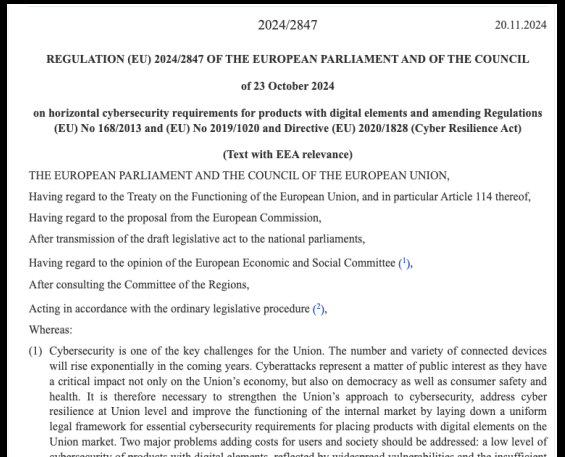
(a) striking subsections 2(a)–(b) and redesignating subsections 2(c), 2(d), and 2(e) as subsections 2(a), 2(b), and 2(c), respectively;

(b) striking the first sentence of subsection 2(e);

(c) striking subsections 3(a)–(b) and redesignating subsections 3(c), 3(d), and 3(e) as subsections 3(a), 3(b), and 3(c), respectively;

SBOMs are Now Legally Required

- **EU Cyber Resilience Act** (adopted Mar 2024): SBOM requirements for *all software sold in the EU*; phasing in 2025–2027
- **FDA** (2023): SBOMs required for medical devices
- Automotive (ISO/SAE 21434), telecom sectors following suit



2024/2847 20.11.2024

REGULATION (EU) 2024/2847 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL

of 23 October 2024

on horizontal cybersecurity requirements for products with digital elements and amending Regulations (EU) No 168/2013 and (EU) No 2019/1020 and Directive (EU) 2020/1828 (Cyber Resilience Act)

(Text with EEA relevance)

THE EUROPEAN PARLIAMENT AND THE COUNCIL OF THE EUROPEAN UNION,

Having regard to the Treaty on the Functioning of the European Union, and in particular Article 114 thereof,

Having regard to the proposal from the European Commission,

After transmission of the draft legislative act to the national parliaments,

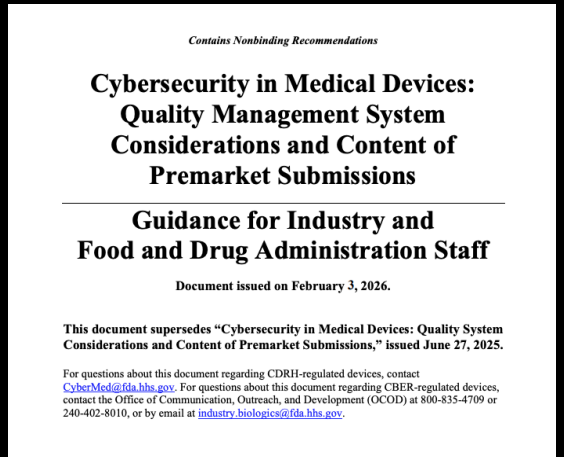
Having regard to the opinion of the European Economic and Social Committee (1),

After consulting the Committee of the Regions,

Acting in accordance with the ordinary legislative procedure (2),

Whereas:

(1) Cybersecurity is one of the key challenges for the Union. The number and variety of connected devices will rise exponentially in the coming years. Cyberattacks represent a matter of public interest as they have a critical impact not only on the Union's economy, but also on democracy as well as consumer safety and health. It is therefore necessary to strengthen the Union's approach to cybersecurity, address cyber resilience at Union level and improve the functioning of the internal market by laying down a uniform legal framework for essential cybersecurity requirements for placing products with digital elements on the Union market. Two major problems adding costs for users and society should be addressed: a low level of



Contains Nonbinding Recommendations

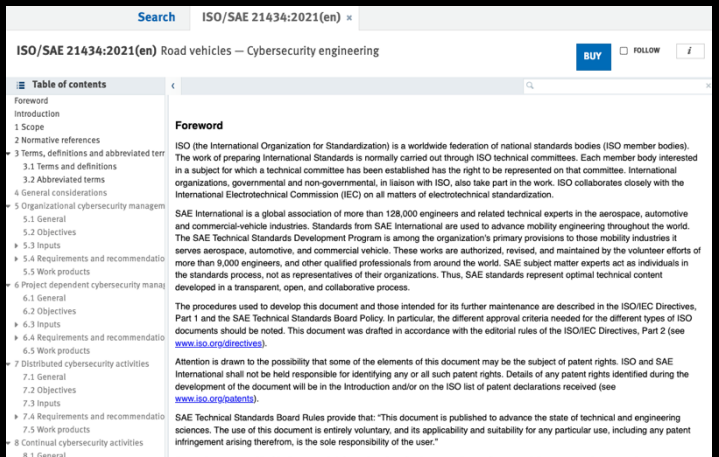
Cybersecurity in Medical Devices: Quality Management System Considerations and Content of Premarket Submissions

Guidance for Industry and Food and Drug Administration Staff

Document issued on February 3, 2026.

This document supersedes "Cybersecurity in Medical Devices: Quality System Considerations and Content of Premarket Submissions," issued June 27, 2025.

For questions about this document regarding CDRH-regulated devices, contact CyberMed@fda.hhs.gov. For questions about this document regarding CBER-regulated devices, contact the Office of Communication, Outreach, and Development (OCOD) at 800-835-4709 or 240-402-8010, or by email at industry.biologics@fda.hhs.gov.



Search ISO/SAE 21434:2021(en) ×

ISO/SAE 21434:2021(en) Road vehicles — Cybersecurity engineering

BUY FOLLOW

Table of contents

Foreword
Introduction
1 Scope
2 Normative references
3 Terms, definitions and abbreviated terr
3.1 Terms and definitions
3.2 Abbreviated terms
4 General considerations
5 Organizational cybersecurity managem
5.1 General
5.2 Objectives
5.3 Inputs
5.4 Requirements and recommendatio
5.5 Work products
6 Project dependent cybersecurity manag
6.1 General
6.2 Objectives
6.3 Inputs
6.4 Requirements and recommendatio
6.5 Work products
7 Distributed cybersecurity activities
7.1 General
7.2 Objectives
7.3 Inputs
7.4 Requirements and recommendatio
7.5 Work products
8 Continual cybersecurity activities
8.1 General

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

SAE International is a global association of more than 128,000 engineers and related technical experts in the aerospace, automotive and commercial-vehicle industries. Standards from SAE International are used to advance mobility engineering throughout the world. The SAE Technical Standards Development Program is among the organization's primary provisions to those mobility industries it serves aerospace, automotive, and commercial vehicle. These works are authorized, revised, and maintained by the volunteer efforts of more than 9,000 engineers, and other qualified professionals from around the world. SAE subject matter experts act as individuals in the standards process, not as representatives of their organizations. Thus, SAE standards represent optimal technical content developed in a transparent, open, and collaborative process.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1 and the SAE Technical Standards Board Policy. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and SAE International shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

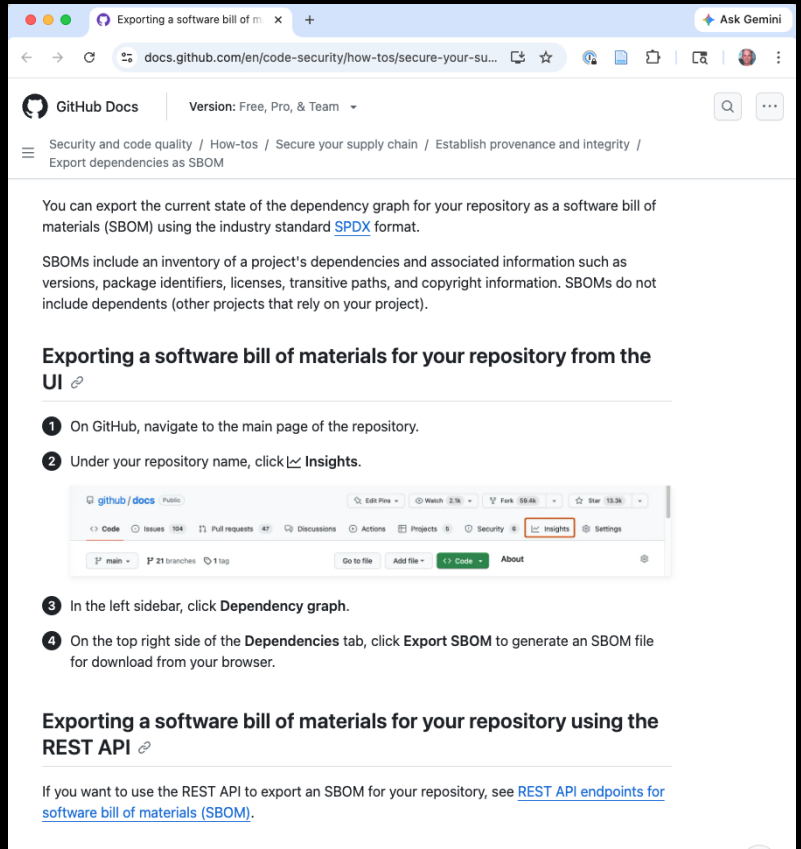
SAE Technical Standards Board Rules provide that: "This document is published to advance the state of technical and engineering sciences. The use of this document is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SBOM Tooling: Generation

GitHub/GitLab have native SBOM export features

Other tools too:

Tool	Scope
`syft` (Anchore)	Multi-language, container images
`cdxgen` / `trivy`	Multi-language; trivy also scans for vulns



SBOM Tooling: Consumption

- Using SBOMs was the major gap in 2022. Better tooling now.

Tool	Purpose
GUAC (Google/OpenSSF)	Graph DB aggregating SBOMs + SLSA + vulns
Dependency-Track (OWASP)	SBOM lifecycle management, policy, VEX
Grype / OSV-Scanner	SBOM → vulnerability report

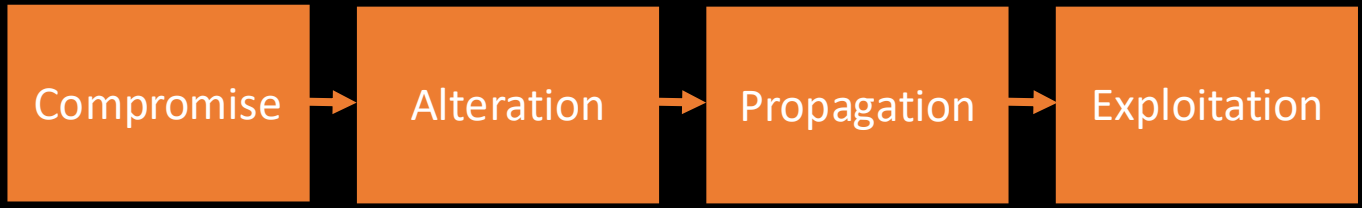
SBOM Formats

Standard	Maintained By	Version	Key Features
SPDX	Linux Foundation / ISO 5962	**3.0** (Apr 2024)	Modular profiles (Core, Software, Security, AI/ML); JSON-LD
CycloneDX	OWASP	**1.6** (Apr 2024)	CBOM (crypto BOM), inline VEX, attestations; rapid iteration
SWID Tags	ISO/IEC 19770-2	—	Older; not recommended for new implementations

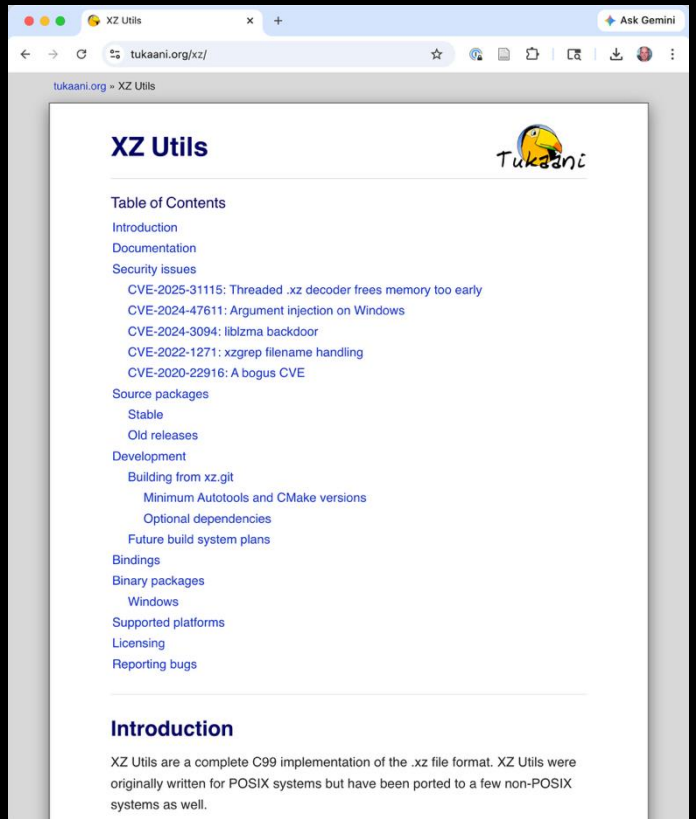
Supply Chain Attacks

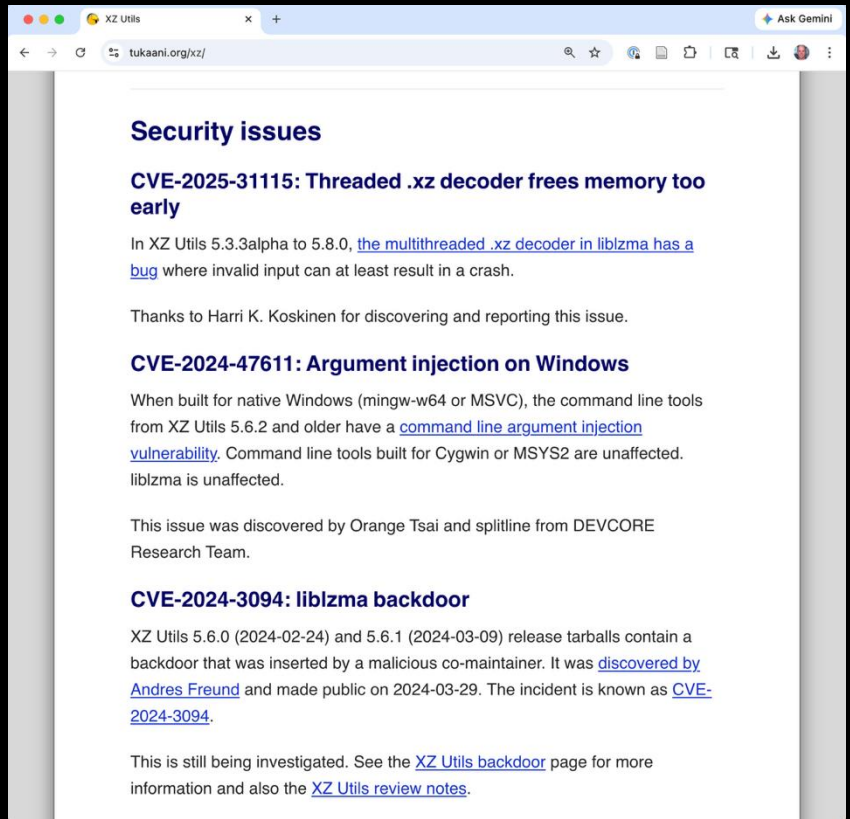
- Distinction: Supply chain vulnerability vs. supply chain *attack*
- Vulnerability: Log4Shell was discovered and reported by Chen Zhaojun of the Alibaba Cloud Security Team; dependents reacted
- Attack: A bad actor specifically targets a supply chain to compromise upstream users

Anatomy of a Supply Chain Attack



Case Study: The XZ Utils Backdoor (2024)





Case Study: The XZ Utils Backdoor (2024)

- “Jia Tan” spent **~2 years** contributing to xz-utils, building trust with the sole maintainer
- **Sockpuppet accounts** pressured the maintainer (citing burnout, slow responses) to add a co-maintainer
- Once granted commit access, inserted a **sophisticated backdoor into liblzma** that would compromise SSH on Linux
- Discovered **by accident** — Andres Freund noticed anomalous SSH performance

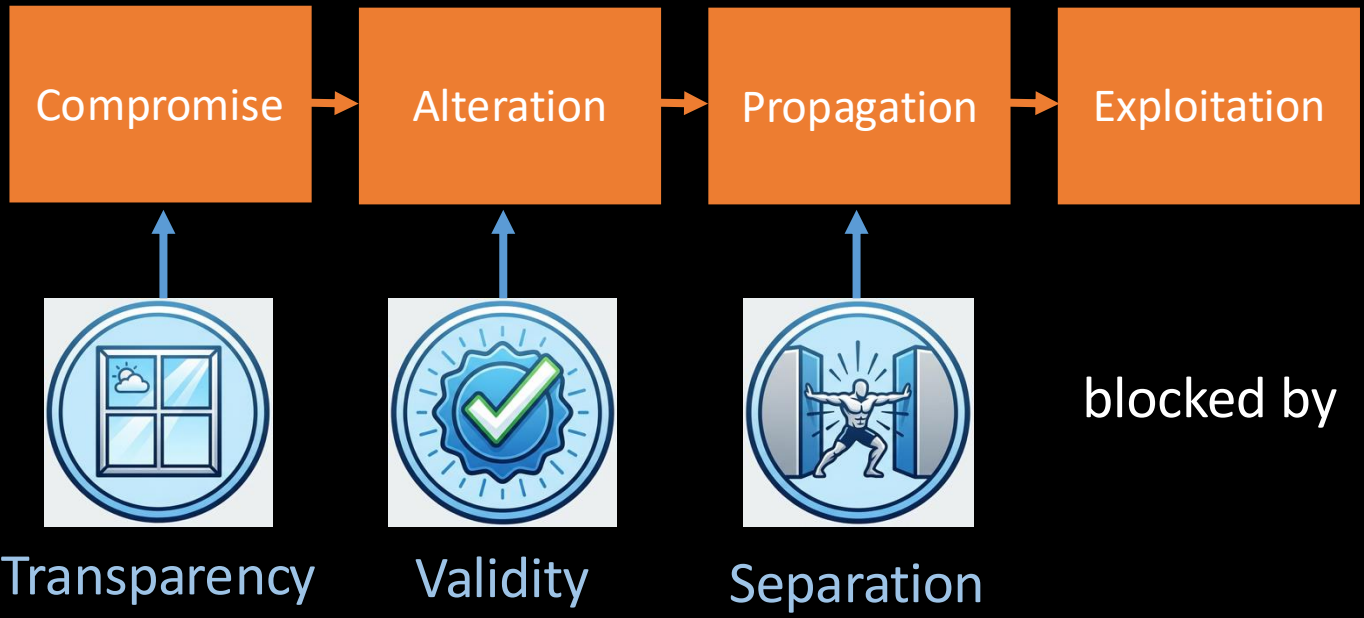
Supply Chain Attacks are Growing

Attack	Year	What Happened
`event-stream` npm	2018	Maintainer handoff to attacker; backdoor added
SolarWinds	2020	Build system compromised; trojanized updates
Codecov	2021	CI tool compromised; credentials exfiltrated
XZ Utils backdoor	2024	Years-long social engineering of maintainer

Case Study: 450,000 Malicious Packages in 2025

- Sonatype report: **450,000+ malicious components** published to open-source registries in one year
- **npm** is the primary target — central role in front-end dev and CI pipelines
- North Korea's **Lazarus Group** published hundreds of malicious npm packages

Defending a Supply Chain Attack



Defending a Supply Chain Attack

Property	What It Provides	Blocks Which Stage
Transparency	Visibility into actors, operations, and artifacts	Compromise
Validity	Integrity and authenticity of components and changes	Alteration
Separation	Compartmentalization to limit blast radius	Propagation

Managing Risk: Vendor Evaluation

- Maintain an **approved-components list** or internal registry
- Assess third-party suppliers against SSDF requirements
- Evaluate open-source projects for:
  - Maintenance health (active maintainers? recent commits?)
  - Known vulnerabilities
  - License compliance

Managing Risk: Ongoing Monitoring

Tool	Type
Dependabot	SCA integrated into GitHub
Snyk	SCA with developer-first workflow
Grype	Open-source container/SBOM vulnerability scanner
OWASP Dependency-Check	Open-source SCA for Java, .NET, etc.

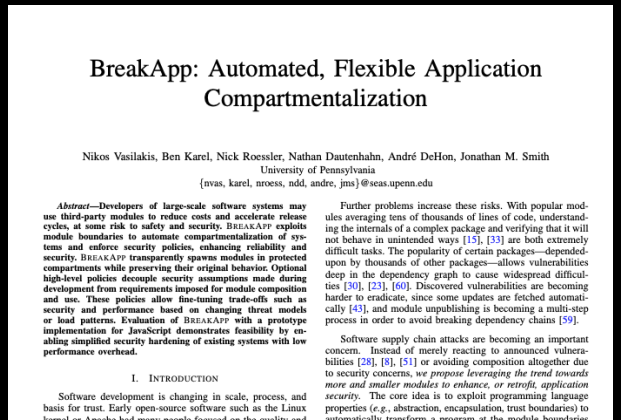
- Subscribe to vulnerability feeds: **NVD**, **GitHub Advisories**, **OSV**
- Use **Software Composition Analysis (SCA)** tools to continuously scan

Managing Risk: Procurement

- Require **SBOMs** from vendors
- Include security requirements in contracts
- Verify vendor attestations (SOC 2, ISO 27001, SSDF compliance)

Managing Risk: Compartmentalization

- Sandbox 3rd party components, to limit blast radius of a vulnerability
  - Uses Web Assembly to memory-isolate the component
- Used in Firefox since 2020



BreakApp: Automated, Flexible Application Compartmentalization

Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, Jonathan M. Smith
University of Pennsylvania
{nvas, karel, nroess, ndd, andre, jms}@seas.upenn.edu

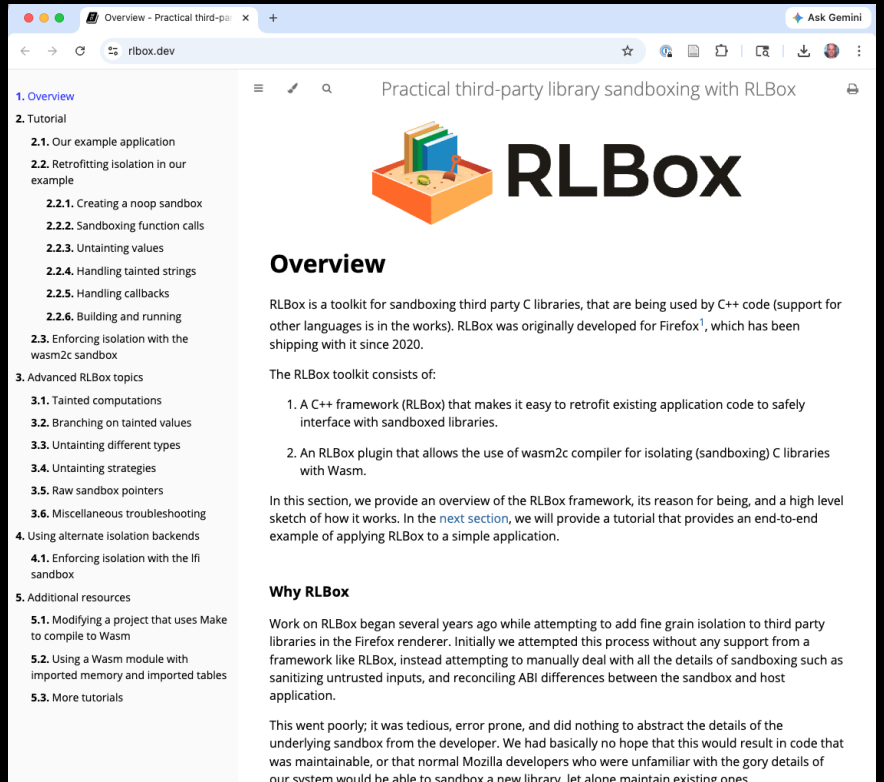
***Abstract*—Developers of large-scale software systems may use third-party modules to reduce costs and accelerate release cycles, at some risk to safety and security. BreakApp exploits module boundaries to automate compartmentalization of systems and enforce security policies, enhancing reliability and security. BreakApp transparently spawns modules in protected compartments while preserving their original behavior. Optional high-level policies decouple security assumptions made during development from requirements imposed for module composition and use. These policies allow fine-tuning trade-offs such as security and performance based on changing threat models or load patterns. Evaluation of BreakApp with a prototype implementation for JavaScript demonstrates feasibility by enabling simplified security hardening of existing systems with low performance overhead.**

I. Introduction

Software development is changing in scale, process, and basis for trust. Early open-source software such as the Linux

Further problems increase these risks. With popular modules averaging tens of thousands of lines of code, understanding the internals of a complex package and verifying that it will not behave in unintended ways [15], [33] are both extremely difficult tasks. The popularity of certain packages—depended-upon by thousands of other packages—allows vulnerabilities deep in the dependency graph to cause widespread difficulties [30], [23], [60]. Discovered vulnerabilities are becoming harder to eradicate, since some updates are fetched automatically [43], and module unpublishing is becoming a multi-step process in order to avoid breaking dependency chains [59].

Software supply chain attacks are becoming an important concern. Instead of merely reacting to announced vulnerabilities [28], [8], [51] or avoiding composition altogether due to security concerns, *we propose leveraging the trend towards more and smaller modules to enhance, or retrofit, application security.* The core idea is to exploit programming language properties (*e.g.*, abstraction, encapsulation, trust boundaries) to



Overview - Practical third-pa

Ask Gemini

rlbox.dev

Practical third-party library sandboxing with RLBox

RLBox

1. Overview
2. Tutorial
   - 2.1. Our example application
   - 2.2. Retrofitting isolation in our example
     - 2.2.1. Creating a noop sandbox
     - 2.2.2. Sandboxing function calls
     - 2.2.3. Untainting values
     - 2.2.4. Handling tainted strings
     - 2.2.5. Handling callbacks
     - 2.2.6. Building and running
   - 2.3. Enforcing isolation with the wasm2c sandbox
3. Advanced RLBox topics
   - 3.1. Tainted computations
   - 3.2. Branching on tainted values
   - 3.3. Untainting different types
   - 3.4. Untainting strategies
   - 3.5. Raw sandbox pointers
   - 3.6. Miscellaneous troubleshooting
4. Using alternate isolation backends
   - 4.1. Enforcing isolation with the lfi sandbox
5. Additional resources
   - 5.1. Modifying a project that uses Make to compile to Wasm
   - 5.2. Using a Wasm module with imported memory and imported tables
   - 5.3. More tutorials

Overview

RLBox is a toolkit for sandboxing third party C libraries, that are being used by C++ code (support for other languages is in the works). RLBox was originally developed for Firefox[1], which has been shipping with it since 2020.

The RLBox toolkit consists of:

1. A C++ framework (RLBox) that makes it easy to retrofit existing application code to safely interface with sandboxed libraries.
2. An RLBox plugin that allows the use of wasm2c compiler for isolating (sandboxing) C libraries with Wasm.

In this section, we provide an overview of the RLBox framework, its reason for being, and a high level sketch of how it works. In the next section, we will provide a tutorial that provides an end-to-end example of applying RLBox to a simple application.

Why RLBox

Work on RLBox began several years ago while attempting to add fine grain isolation to third party libraries in the Firefox renderer. Initially we attempted this process without any support from a framework like RLBox, instead attempting to manually deal with all the details of sandboxing such as sanitizing untrusted inputs, and reconciling ABI differences between the sandbox and host application.

This went poorly; it was tedious, error prone, and did nothing to abstract the details of the underlying sandbox from the developer. We had basically no hope that this would result in code that was maintainable, or that normal Mozilla developers who were unfamiliar with the gory details of

Managing Risk: Version Locking

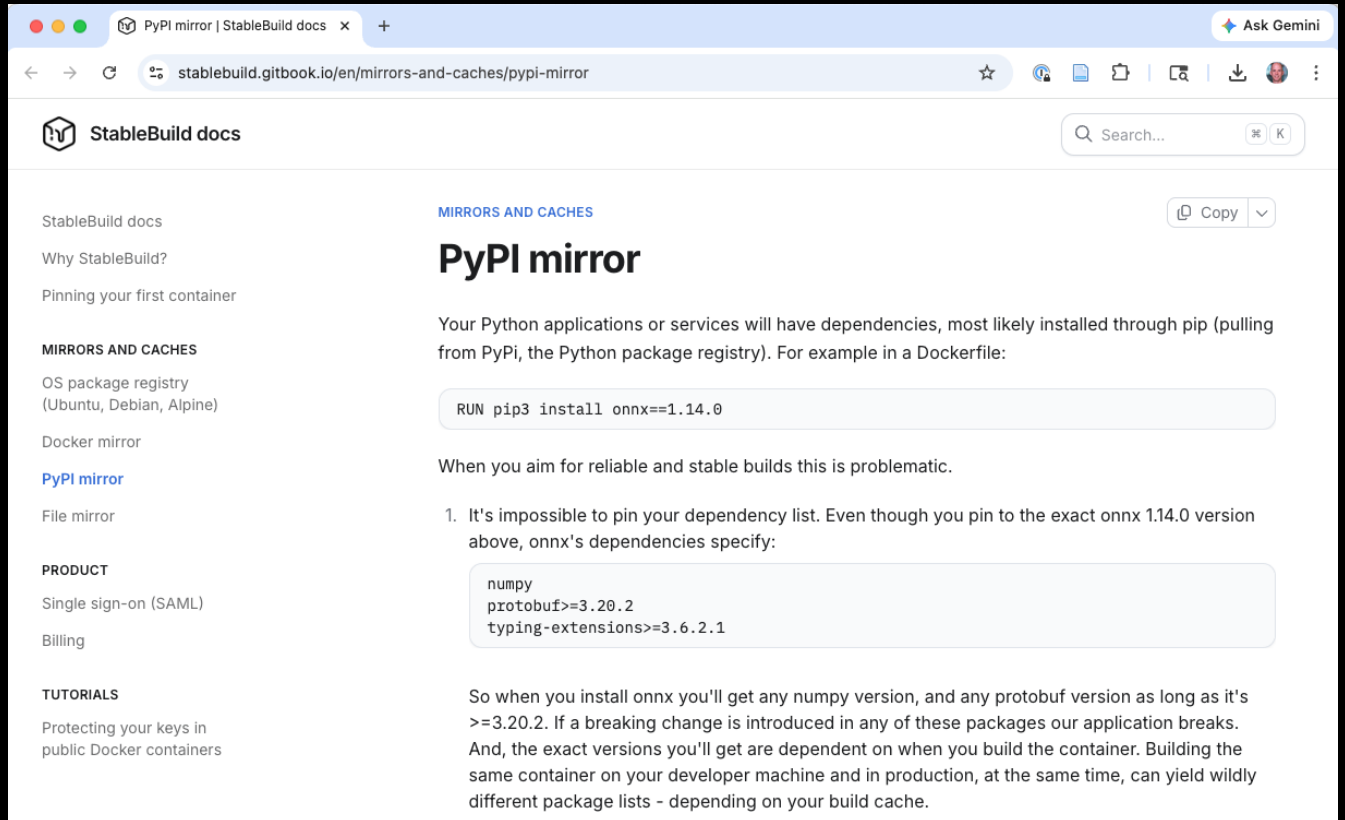
```
[package]
name = "cedar-policy"
...
repository = "https://github.com/cedar-policy/cedar"


[dependencies]
cedar-policy-core = { version = "=4.8.2", path = "../cedar-policy-core" }
cedar-policy-formatter = { version = "=4.8.2", path = "../cedar-policy-formatter" }
ref-cast = "1.0"
serde = { version = "1.0", features = ["derive", "rc"] }
serde_json = "1.0"
itertools = "0.14"
miette = "7.6.0"
thiserror = "2.0"
smol_str = { version = "0.3", features = ["serde"] }
dhat = { version = "0.3.2", optional = true }
serde_with = "3.16.1"
nonempty = { version = "0.12", optional = true }
prost = { version = "0.14", optional = true }
linked-hash-map = { version = "0.5.6", features = ["serde_impl"] }
```

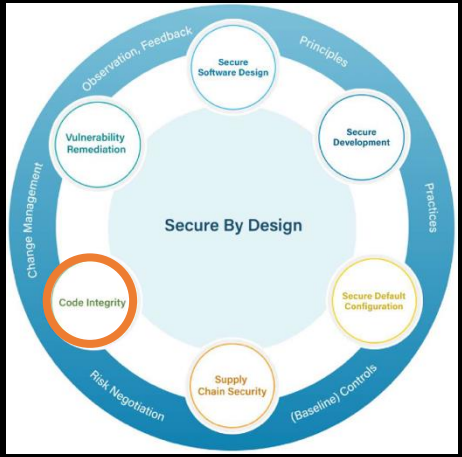
Locked version

SemVer compatibility
(not locked)

Cedar's
`Cargo.toml`

Managing Risk: Mirroring





Code Integrity

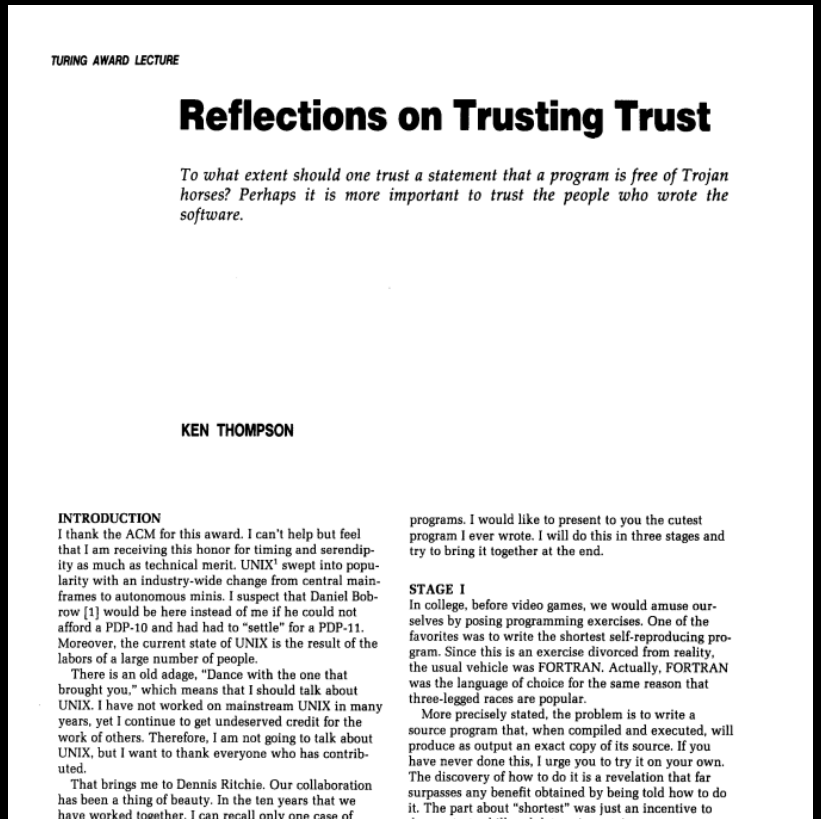
What is Code Integrity?

Ensuring that the software delivered to users is the software the organization intended to deliver

Example supply chain attack vectors:

1. **Misplaced/stolen trust** —
   malicious actor commits malware or vulnerable code
   - XZ Utils attack as a case study
2. **Tampering** — malicious actor compromises build system, artifacts

Reflections on Trusting Trust



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.

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[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

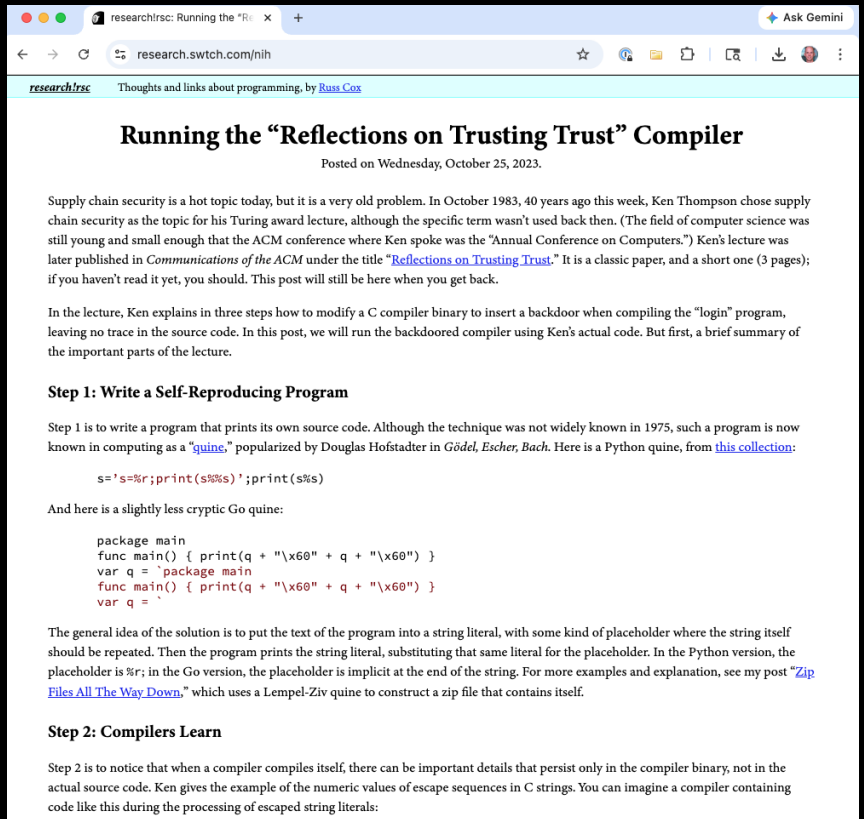
That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of

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

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to



research!rsc: Running the "R

Ask Gemini

research.swtch.com/nih

research!rsc Thoughts and links about programming, by Russ Cox

Running the "Reflections on Trusting Trust" Compiler

Posted on Wednesday, October 25, 2023.

Supply chain security is a hot topic today, but it is a very old problem. In October 1983, 40 years ago this week, Ken Thompson chose supply chain security as the topic for his Turing award lecture, although the specific term wasn't used back then. (The field of computer science was still young and small enough that the ACM conference where Ken spoke was the "Annual Conference on Computers.") Ken's lecture was later published in *Communications of the ACM* under the title "Reflections on Trusting Trust." It is a classic paper, and a short one (3 pages); if you haven't read it yet, you should. This post will still be here when you get back.

In the lecture, Ken explains in three steps how to modify a C compiler binary to insert a backdoor when compiling the "login" program, leaving no trace in the source code. In this post, we will run the backdoored compiler using Ken's actual code. But first, a brief summary of the important parts of the lecture.

Step 1: Write a Self-Reproducing Program

Step 1 is to write a program that prints its own source code. Although the technique was not widely known in 1975, such a program is now known in computing as a "quine," popularized by Douglas Hofstadter in *Gödel, Escher, Bach*. Here is a Python quine, from this collection:

```
s='s=%r;print(s%%s)';print(s%s)
```

And here is a slightly less cryptic Go quine:

```
package main
func main() { print(q + "\x60" + q + "\x60") }
var q = `package main
func main() { print(q + "\x60" + q + "\x60") }
var q = `
```

The general idea of the solution is to put the text of the program into a string literal, with some kind of placeholder where the string itself should be repeated. Then the program prints the string literal, substituting that same literal for the placeholder. In the Python version, the placeholder is `%r`; in the Go version, the placeholder is implicit at the end of the string. For more examples and explanation, see my post "Zip Files All The Way Down," which uses a Lempel-Ziv quine to construct a zip file that contains itself.

Step 2: Compilers Learn

Step 2 is to notice that when a compiler compiles itself, there can be important details that persist only in the compiler binary, not in the actual source code. Ken gives the example of the numeric values of escape sequences in C strings. You can imagine a compiler containing code like this during the processing of escaped string literals:

Key idea (Fun Digression)

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
...
```

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return('\v');
...
```

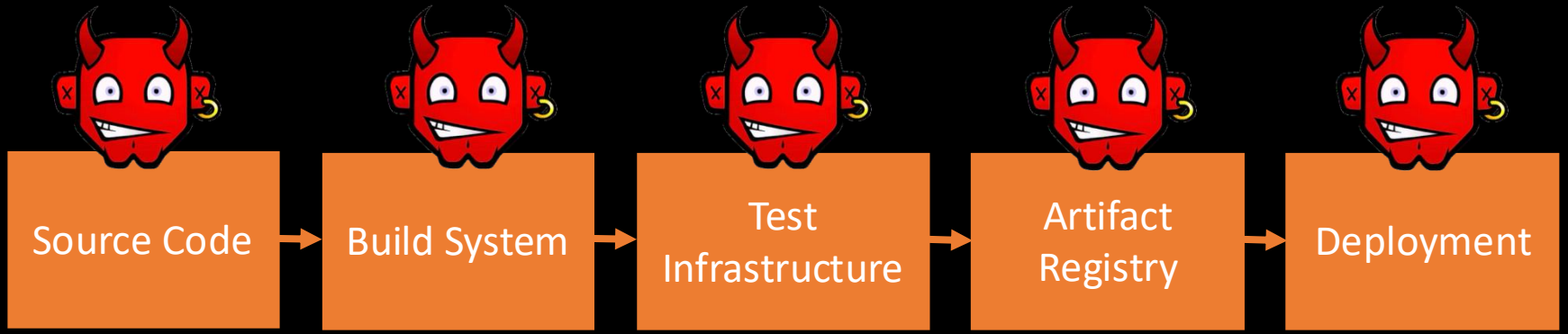
```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\ n');
if(c == 'v')
        return(11);
...
```

Ensuring Code Integrity

Two domains:

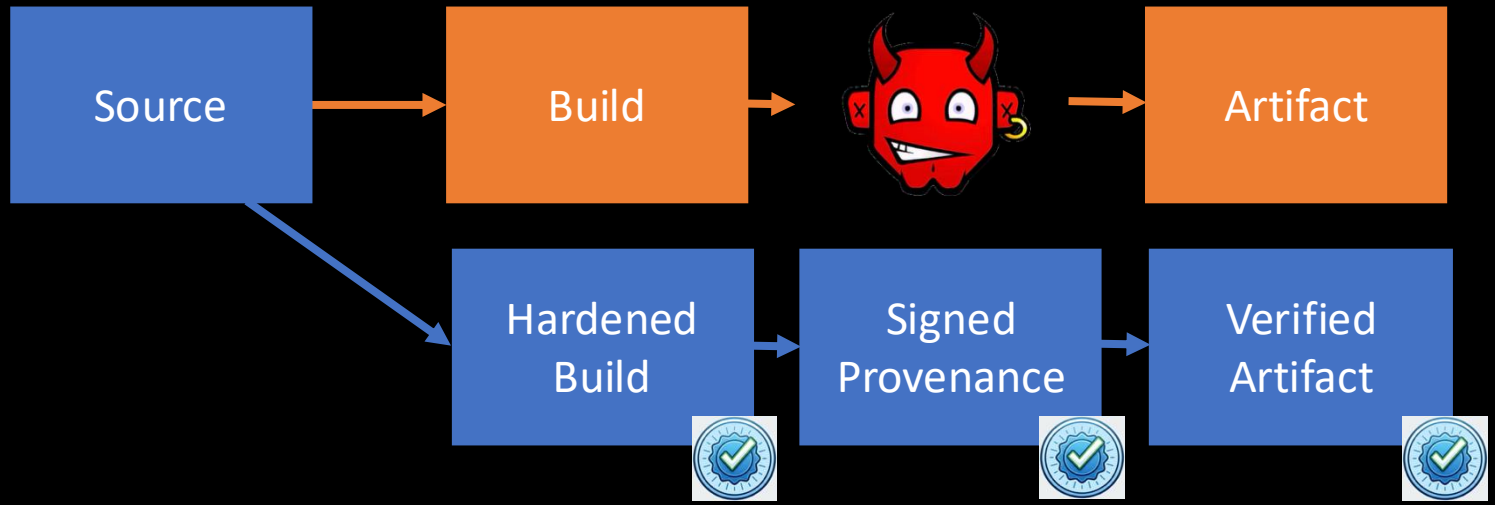
1. **Change management** — who can modify what → *Separation*
2. **Tamper detection** — proving nothing was modified (or later discovering that it was) → *Validity*

The Build Pipeline is Part of Your Supply Chain



- GitHub Actions runners, Jenkins agents, CI plugins
- Build tools, compilers, container base images
- **Every tool in the pipeline is a dependency you must trust**

Build Pipeline Integrity



Source Control Protections

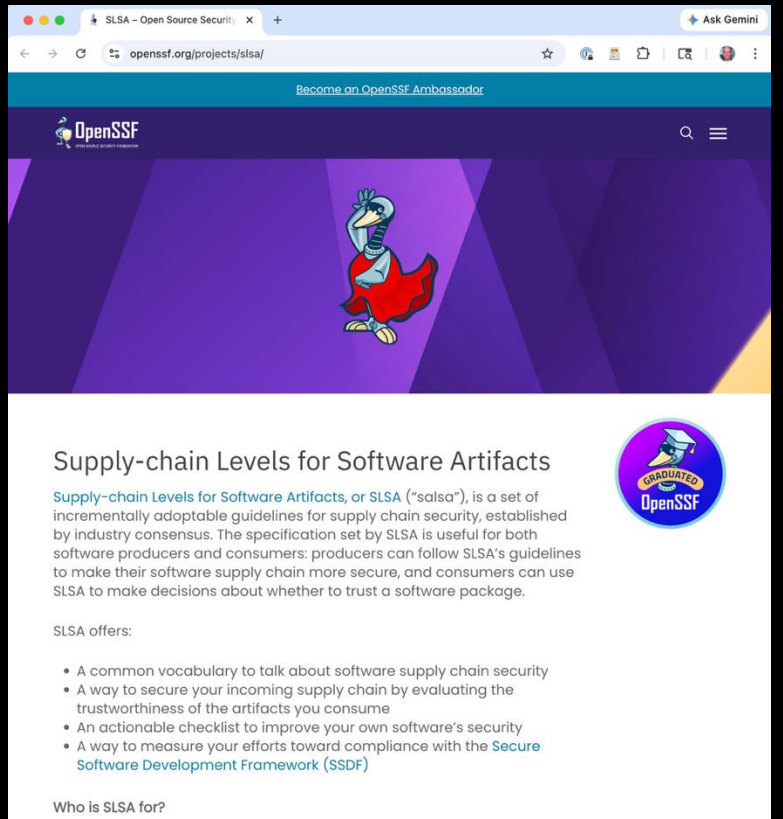
- **Branch protection rules** — require reviews before merge
- **Signed commits** — GPG or SSH signatures prove authorship
- **Principle of least privilege** — limit who has write access
- **Audit logs** — who changed what, when

Code Signing and Artifact Verification

- **Sigstore** is now the standard for open-source signing
- **Keyless**: authenticate via OIDC
  → Fulcio issues short-lived certificate
  → Rekor logs it
- No long-lived keys to manage, lose, or have stolen

SLSA: Track-Based Build Integrity

Build Level	Requirements
L1	Provenance exists — build process documented
L2	Hosted build service; authenticated/signed provenance
L3	Hardened build platform; unforgeable provenance; isolation between builds



The Identity vs. Trustworthiness Gap

What our tools **can** verify today:

- This package was published by **GitHub account X** (Sigstore OIDC)
- Account X has **MFA enabled** (registry policy)
- The build ran in **GitHub Actions from repo Y** (SLSA provenance)

What our tools **cannot** verify:

- Is the human behind account X **actually trustworthy**?
- Has a trusted maintainer been **socially engineered or coerced**?

Case Study: “Glassworm” — Invisible Code Attack (March 2026)

- **Aikido Security** found **151 malicious packages** on GitHub (March 3–9), plus more on npm and VS Code marketplace
- Malicious payloads encoded using **invisible Unicode characters** (variation selectors)
- Code reviewers and static analysis tools see **blank lines** — the JavaScript runtime sees executable code

Glassworm Returns: Invisible

Ask Gemini

aikido.dev/blog/glassworm-returns-unicode-attack-github-npm-vscode

← Blog / Vulnerabilities & Threats

Glassworm Is Back: A New Wave of Invisible Unicode Attacks Hits Hundreds of Repositories

Written by Ilyas Makari

Published on: Mar 13, 2026

Table of Contents

The invisible threat we've been tracking for nearly a year is back. While the PolinRider campaign has been making headlines for compromising hundreds of GitHub repositories, we are separately seeing a new wave of Glassworm activity hitting GitHub, npm, and VS Code.

In October last year, we wrote about how hidden Unicode characters were being used to compromise GitHub repositories, tracing the technique back to a threat actor named *Glassworm*. This month, the same actor is back, and among the affected repositories are some notable names: a repo from Wasmer, Reworm, and opencode-bench from anomalyco, the organization behind OpenCode and SST.

A Year of the Invisible Code Campaign

- March 2025: Aikido first discovers malicious npm packages hiding payloads using PUA Unicode characters.
- May 2025: We publish a blog detailing the risks of invisible Unicode and how it can be abused in supply chain attacks.
- October 17, 2025: We uncover compromised extensions on Open VSX using the same technique.
- October 31, 2025: We discover that the attackers have shifted focus to GitHub repositories.
- March 2026: A new mass wave emerges: hundreds of GitHub repositories compromised, with npm and VS Code also affected.

How the Invisible Code Works

```
const s = v => [...v].map(w =>
  w = w.codePointAt(0),
  w >= 0xFE00 && w <= 0xFE0F ? w - 0xFE00 :
  w >= 0xE0100 && w <= 0xE01EF ? w - 0xE0100 + 16 : null
)).filter(n => n !== null);

eval(Buffer.from(s(``)).toString('utf-8'));
//                    ^^ looks empty -- actually has invisible chars
```

- The backtick string **looks empty** in every editor
- The decoder maps invisible variation selectors back to ASCII bytes
- Surrounding code is realistic: version bumps, doc tweaks, refactors

Mapping Glassworm to the Four-Stage Pattern

Stage	How Glassworm Does It
Compromise	Exploit weak package vetting on npm/GitHub (no actor verification)
Alteration	Inject invisible Unicode payloads into packages, AI-generated cover code
Propagation	Developers install via `npm install`; CI pipelines pull dependencies automatically
Exploitation	Decoder runs `eval()`, steals tokens/credentials, fetches second-stage via Solana

How Open-Source Malware Targets Developers

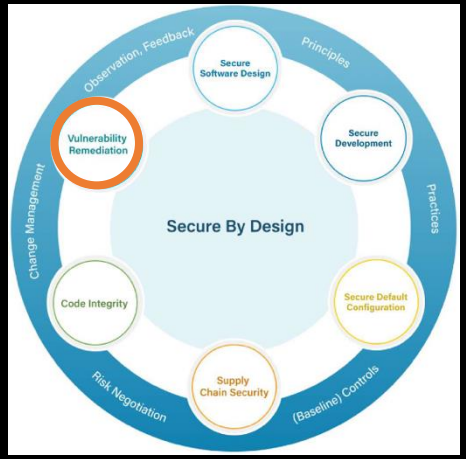
- **Installation-time execution**: malicious code runs on `npm install`, *before* the app even builds
- One download is enough to harvest **API keys, tokens, and deployment credentials**
- Packages mimic familiar plugins with standard documentation
- One campaign **spread without manual republishing** — hundreds of downstream packages compromised in days

Insider Threat and Personnel Controls

- **Separation of duties** — developer ≠ release engineer
- **Personnel vetting** for sensitive/critical software
- **Least-privilege access** to production environments

Zero Trust in the Dev Environment

- **MFA now mandatory** on major registries (npm for high-impact packages 2022; PyPI for all maintainers 2023)
- **Trusted Publishers**: OIDC-based publishing replaces long-lived API tokens entirely
- **Hardware security keys** (YubiKey, etc.) resist phishing
- **Endpoint security** on developer workstations



Vulnerability Remediation

SBOMs as a Remediation Tool

When a malicious package campaign is discovered:

1. Security researchers publish a **list of known-bad packages**
   - e.g., the 151 Glassworm packages
2. Query your SBOMs: **“Do any of our products or build environments contain these?”**
3. Identify affected products, **triage by exposure**, and remediate

Without an SBOM, this is a manual, error-prone search across every project

I found a vulnerability: Now what?

No software ships bug-free. The question is: **what happens when bugs are found?**

Three phases:

1. **Immediate remediation:** Fix the reported vulnerability, ship a patch
2. **Variant analysis:** Search for *similar* vulnerabilities proactively
3. **Process improvement:** Update tools, training, and practices to prevent recurrence

Vulnerability Intake

- Publish a **vulnerability disclosure policy**
  - `security.txt` (RFC 9116) on your website
  - Bug bounty program (HackerOne, Bugcrowd)
- Establish a **Product Security Incident Response Team (PSIRT)**
- Accept reports via standardized channels (CSAF, VEX)

Triage and Prioritization

Not all vulnerabilities are equal — **prioritize by risk**

System	What It Measures
CVSS	Severity of the vulnerability itself
EPSS	Probability of exploitation in the wild
CISA KEV	Known to be actively exploited

Prioritization in Practice

- Define **SLAs** for response times by severity class
- Example:
  Critical = patch within 24 hours,
  High = 7 days,
  Medium = 30 days

Exploitability

Low/High **"Fix in next release"**	**High/High** **"patch NOW"**
Low/Low "Schedule fix"	High/Low "Plan fix soon"

Business Impact

Remediation and Release

- Patch development, testing, and **staged rollout**
- Security advisories in standardized formats:
  - **CSAF** (Common Security Advisory Framework)
  - **GitHub Security Advisories**
- **Coordinated vulnerability disclosure** with reporters and downstream users

Root Cause Analysis and Feedback Loop

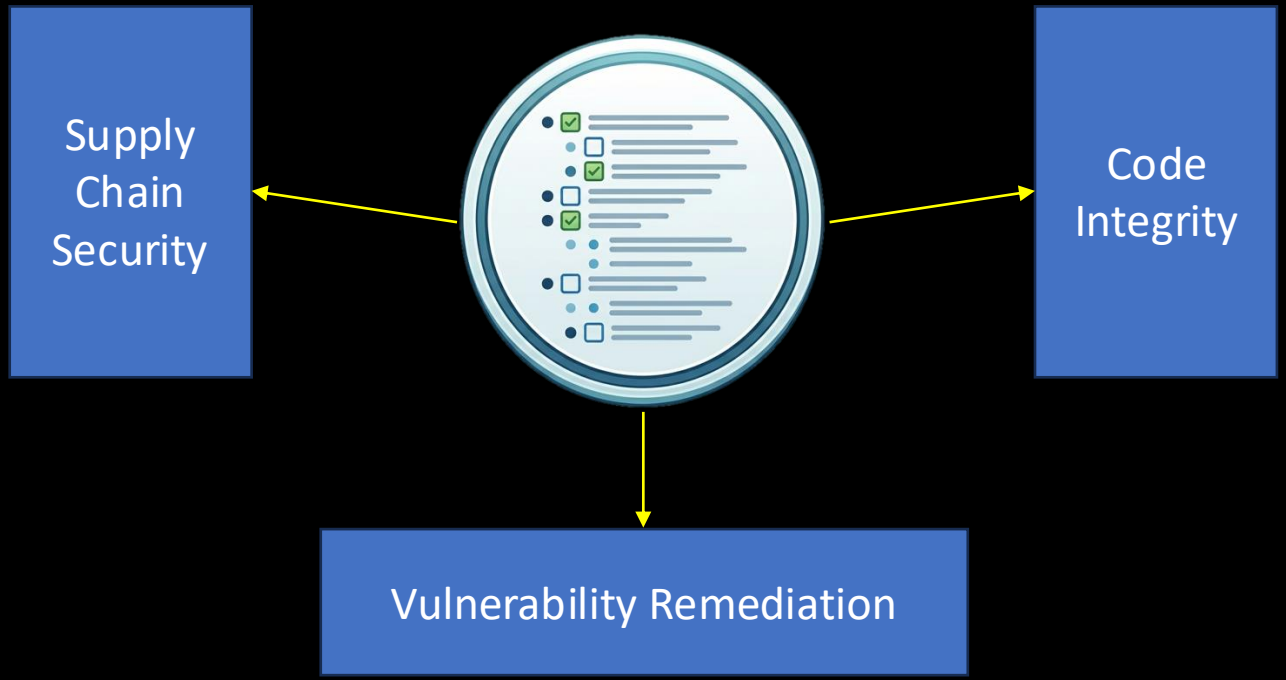
1. **Classify** the root cause (memory safety error, injection, logic flaw, …)
2. **Update** coding standards, training, and tools
3. **Variant analysis** — search for the same pattern elsewhere
4. **Feed findings** back into threat models

Wrapping up

Putting it all together

Techniques	Transparency			Validity			Separation		
	Artifacts	Operations	Actors	Artifacts	Operations	Actors	Artifacts	Operations	Actors
SBOM	✓	✓							
npm-audit [55]	✓			✓					
Code scanning [1]	✓			✓					
Dependabot features [29]	✓			✓					
GitHub Actions [28]		✓		✓	✓			✓	
Git Commit Signing [27]			✓	✓					
Scope [54]				✓			✓		✓
Multi-Factor Authentication						✓			
In-toto [73]	✓	✓		✓	✓			✓	✓
Containerization							✓	✓	✓
Version Locking							✓		
Sigstore [51]	✓	✓	✓	✓	✓				
Mirroring and Proxies [53]	✓			✓			✓	✓	

How SBOMs Enable the Three Activities



SBOM Challenges: Solved, Improving, and Open

Challenge	Status	How
Naming	Largely solved	Package URL (purl) + OSV database sidestep CPE inconsistencies
Consumption	Improving	GUAC, Dependency-Track, Grype+SBOM
Completeness	Improving	better tooling, but vendored code and build tools still problematic
Multi-tier correlation	Still open	your vendor's vendor's vendor
VEX adoption	Still open	specs exist but producing VEX is labor-intensive
Quality	Still open	garbage-in, garbage-out; no SBOM accuracy standards

State of Supply Chain Security

- In 2022, most security tools focused on **artifacts**
- Since then: **operations** much improved (SLSA v1.0, build attestations); **actor identity** partially addressed (Sigstore OIDC, mandatory MFA)
- But **actor trustworthiness** remains the hardest unsolved problem

AI is Changing the Supply Chain Threat Landscape

Attackers use AI to:

- Generate convincing malicious packages at scale (Glassworm)
- Create realistic commit messages, docs, and code style

AI tools inadvertently create risk:

- AI coding assistants recommend **non-existent packages** (27.76% hallucination rate)
- Attackers register those hallucinated names with malicious code
- AI agents autonomously installing unvalidated dependencies in CI

AI as Attack Multiplier

- Both Aikido and security firm Koi suspect **Glassworm is using LLMs** to generate packages
- “Manual crafting of 151+ bespoke code changes across different codebases simply isn’t feasible”
- AI enables attackers to produce **convincing, stylistically consistent** malicious contributions at scale

Key Takeaways

1. **Supply chain attacks** follow four stages: compromise → alteration → propagation → exploitation
2. **Three security properties** block these stages: transparency, validity, separation
3. Current tools mostly protect **artifacts** — operations and actors remain underserved
4. **SBOMs** are the primary tool for transparency, enabling rapid exposure assessment
5. **AI** is changing the threat landscape — both as an attack multiplier and an inadvertent risk

Tools Summary

Activity	Key Tools
SCA / Dependency scanning	Dependabot, Snyk, Grype, Socket.dev
Build integrity / provenance	SLSA v1.0, in-toto attestations, Tekton Chains
Code signing	Sigstore (cosign, Fulcio, Rekor); GPG
Registry provenance	npm provenance, PyPI attestations (PEP 740), GitHub Artifact Attestations
Actor security	Mandatory MFA, Trusted Publishers (OIDC), Sigstore identity binding
Source control security	Branch protection, signed commits, GitHub Vigilant Mode
Vulnerability tracking	CVE, NVD, OSV, GitHub Advisories
Severity scoring	CVSS, EPSS, CISA KEV
SBOM generation	Syft, cdxgen, Trivy, GitHub/GitLab native export
SBOM consumption	GUAC, Dependency-Track, OSV-Scanner
Malware detection	OpenSSF Package Analysis, Socket.dev, registry scanning

Further Reading

- **Okafor et al.**, “SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties” (SCORED ’22) — *assigned reading*
- CIS/SAFECode, “Secure by Design: A Guide to Assessing Software Security Practices” (Oct 2025)
- NIST SP 800-218, “Secure Software Development Framework (SSDF)”
- SLSA framework: slsa.dev
- Sigstore: sigstore.dev
- CISA “Secure by Design” initiative