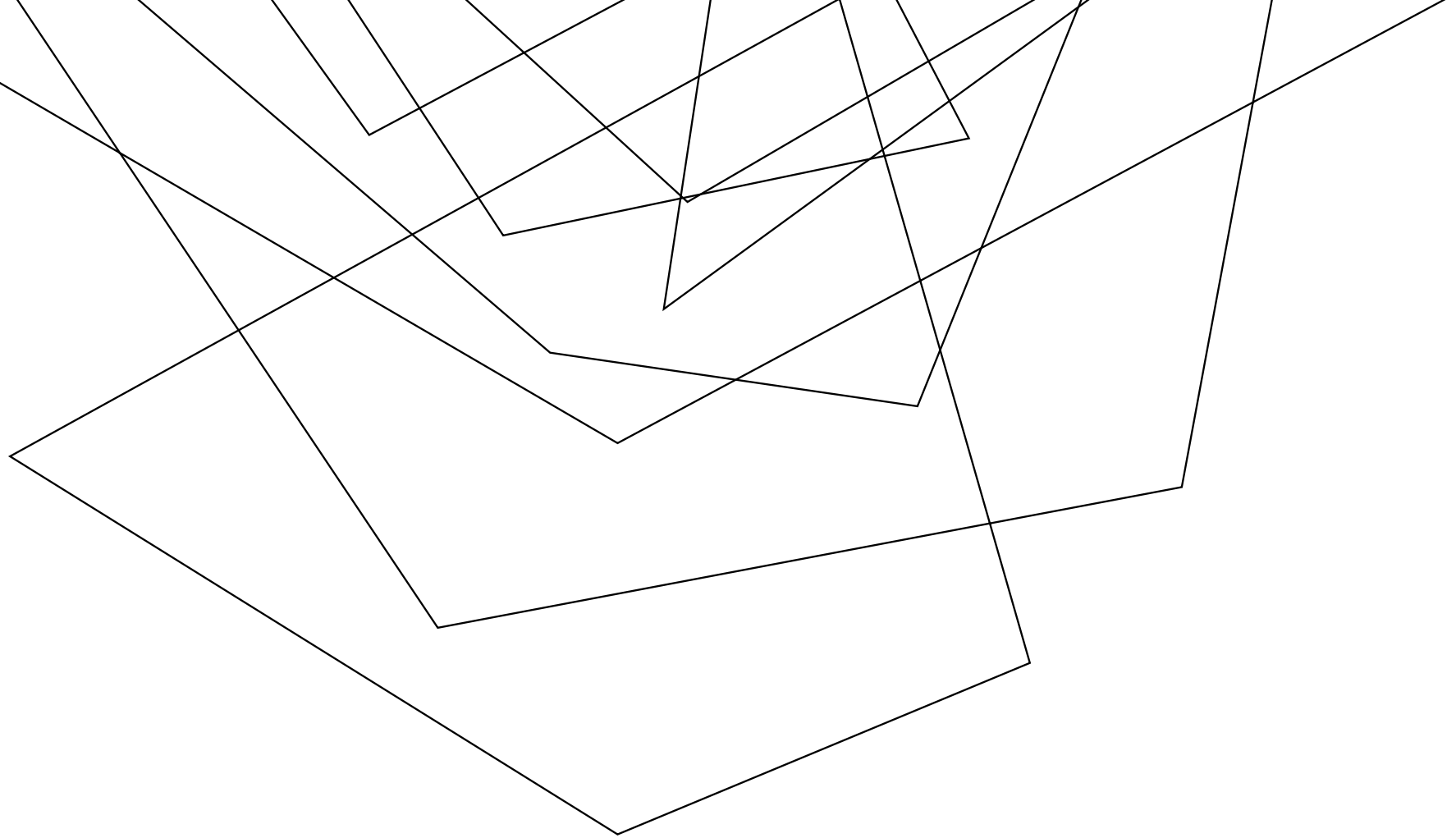
EXERCISE #35

BUG ISOLATION REVIEW

Write your name and answer the following on a piece of paper

Cooperative Bug Isolation (CBI) avoids the use of a periodic global counter to report bugs. It also avoids the use of a random test at each report point. Why?

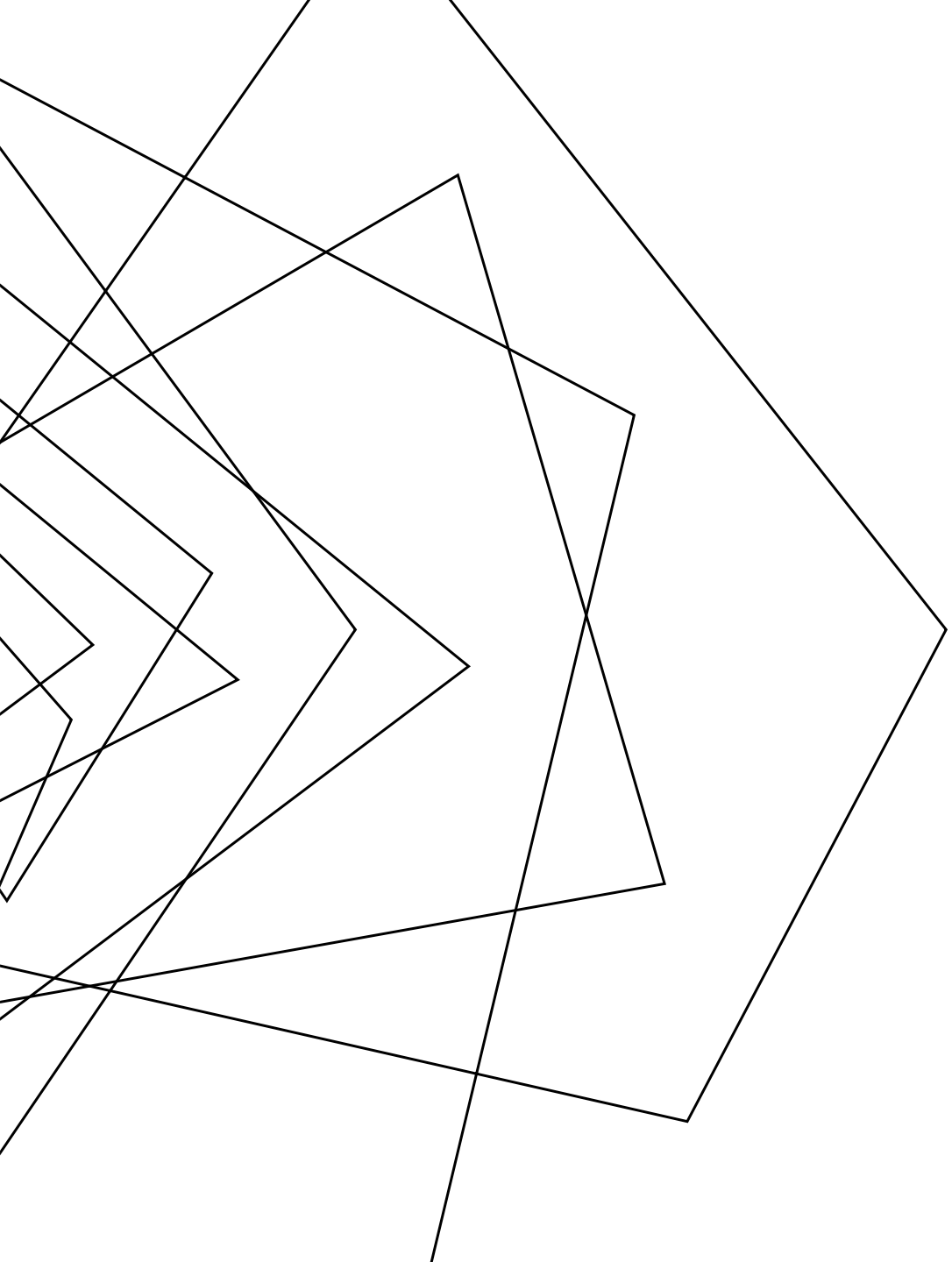
ADMINISTRIVIA AND ANNOUNCEMENTS



SUPPLY CHAIN SECURITY

EECS 677: Software Security Evaluation

Drew Davidson



WHERE WE'RE AT

GRAB-BAG TOPICS!

PREVIOUSLY: BUG ISOLATION

LAST LECTURE REVIEW

ISOLATING CAUSE-EFFECT CHAINS IN PROGRAM MISBEHAVIOR

- Why we isolate bugs
- How we isolate bugs

THIS LECTURE
BUG ISOLATION

Software Supply Chain Security

- Supply chain overview
- Threats
- Defenses

WHAT IS A SOFTWARE SUPPLY CHAIN?

SOFTWARE SUPPLY CHAINS: OVERVIEW

DEFINITION

The components, libraries, tools, and processes used to develop, build, and publish a software artifact.

– wikipedia

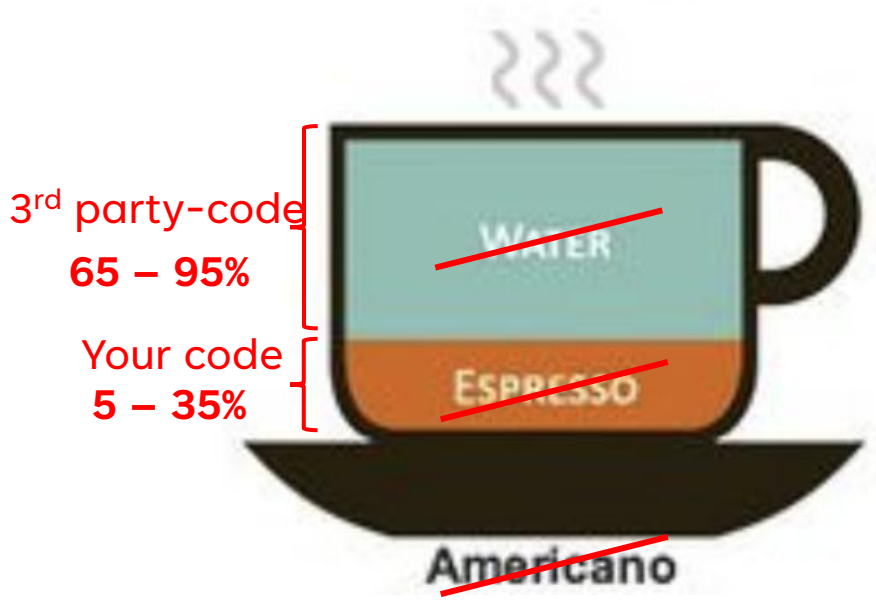
RELATED TERMINOLOGY

(somewhat inspired by traditional supply chains)

Software Bill of Materials (SBOM) – A declaration of the ~~the~~ the inventory of components used to build a software artifact

SOFTWARE SUPPLY CHAIN – IMPACT

SOFTWARE SUPPLY CHAINS: OVERVIEW



MODERN SOFTWARE ECOSYSTEMS

SOFTWARE SUPPLY CHAINS: OVERVIEW

LBEs (Language-Based ecosystems)

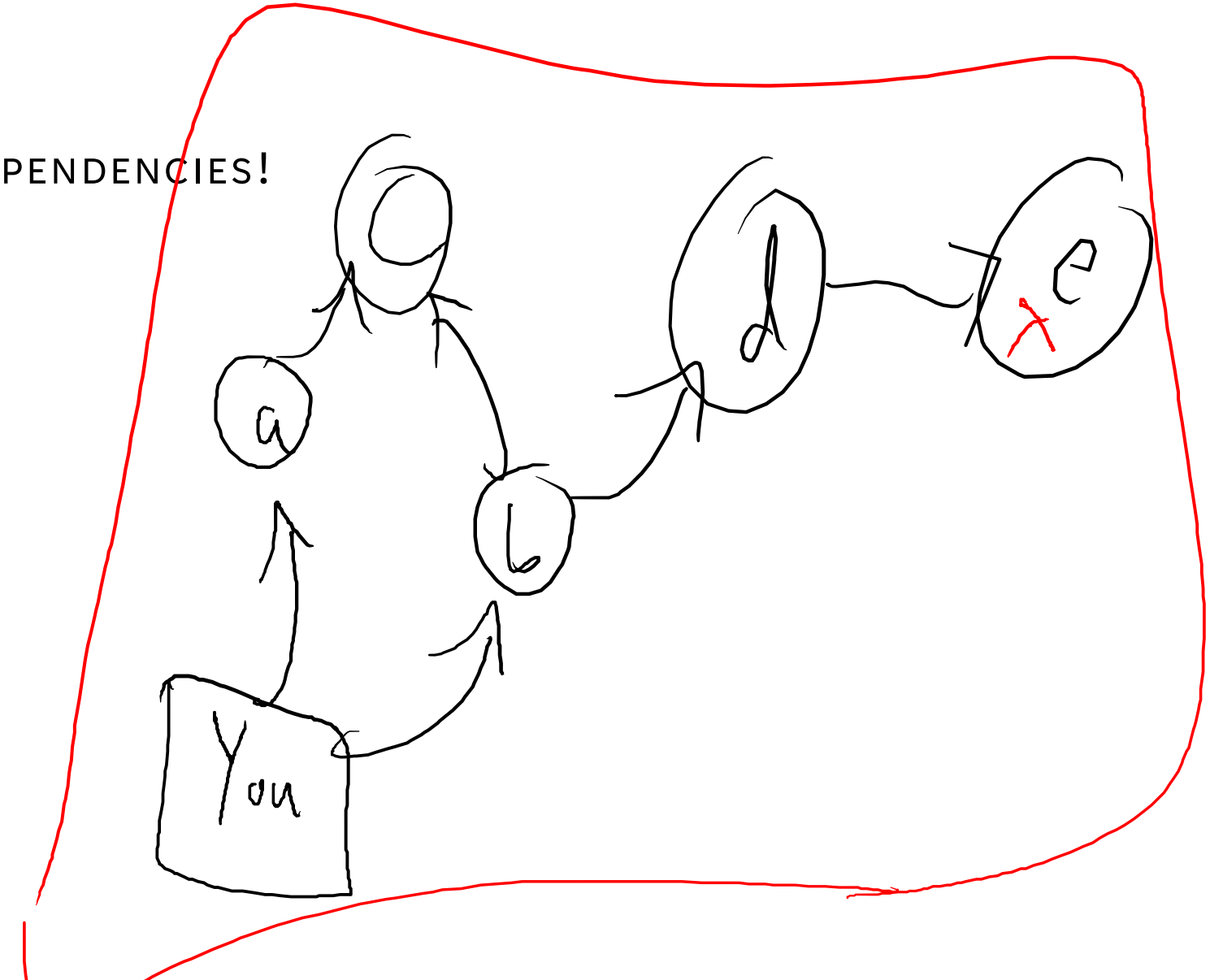
The enabling infrastructure for delivering software supply chains

- Repositories
- Dependencies
- Packages
- Frontend tools

DEPENDENCY WEBS

SOFTWARE SUPPLY CHAINS: OVERVIEW

DEPENDENCIES... HAVE DEPENDENCIES!



PACKAGE MANAGERS FRONTENDS

SOFTWARE SUPPLY CHAINS: OVERVIEW

VASTLY SIMPLIFY THE TASK OF DEPENDENCY MANAGEMENT

Finding, fetching and installing the dependency web

BENIGN-MODEL CHALLENGES

SOFTWARE SUPPLY CHAINS: THREATS

EVEN LACKING MALICIOUS INTENT, SUPPLY CHAIN SECURITY CHALLENGES ARISE

The components, libraries, tools, and processes used to develop, build, and publish a software artifact.

Software rot – The world changes, software needs to change with it

Incompatibilities – A dependency might change

WHO UPLOADS PACKAGES?

SOFTWARE SUPPLY CHAINS: THREATS

ANYBODY!

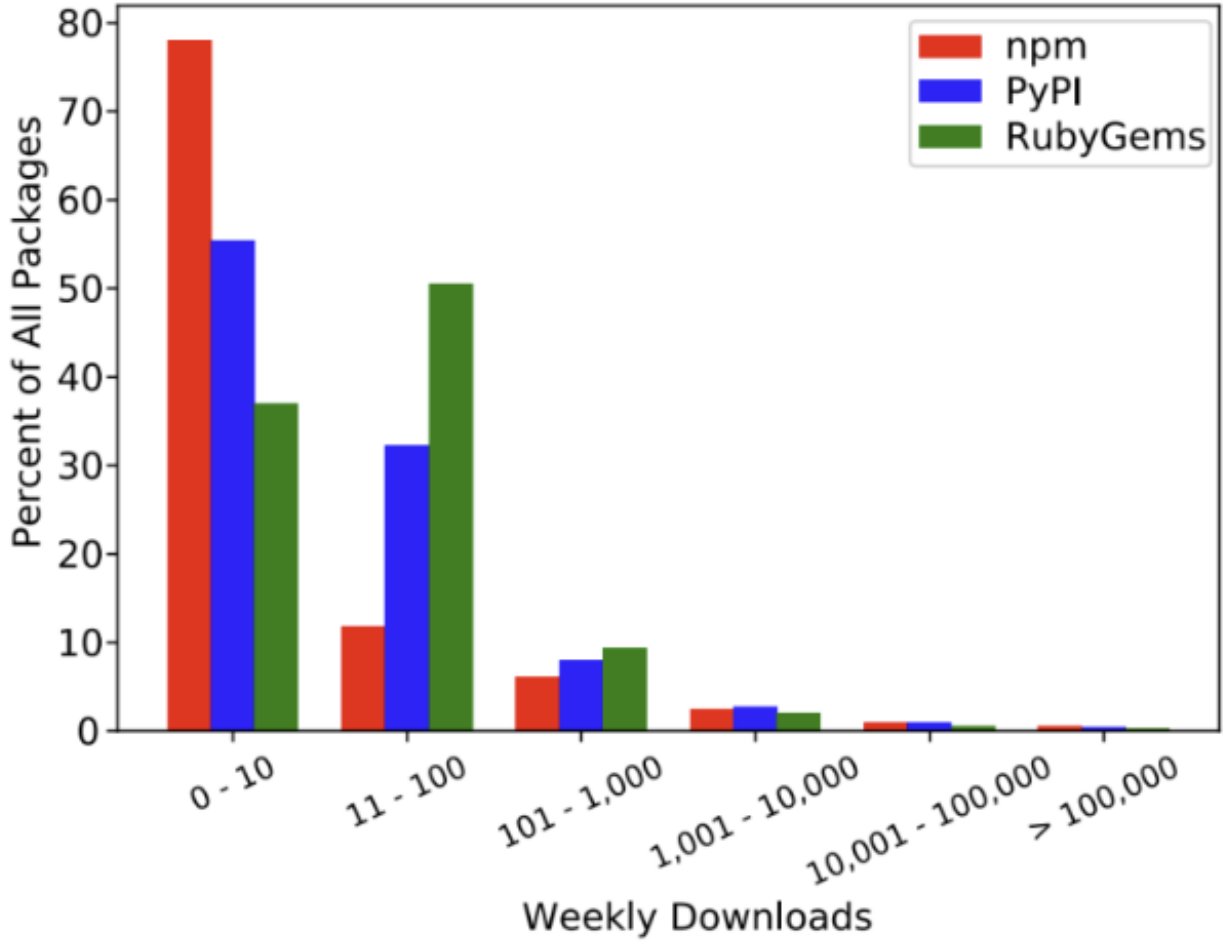
Most package repositories are completely free to use

Allow pseudonymous identification

THE SEA OF GARBAGE

SOFTWARE SUPPLY CHAINS: THREATS

MOST PACKAGES ARE NEVER INSTALLED



MALICIOUS-MODEL CHALLENGES

SOFTWARE SUPPLY CHAINS: THREATS

ANYBODY CAN UPLOAD... DESPITE THEIR INTENT

Very little vetting on whether the package is ok

THREAT VECTORS

SOFTWARE SUPPLY CHAINS: THREATS

INSTALL-TIME ATTACKS

Insert bad code that impacts your installation

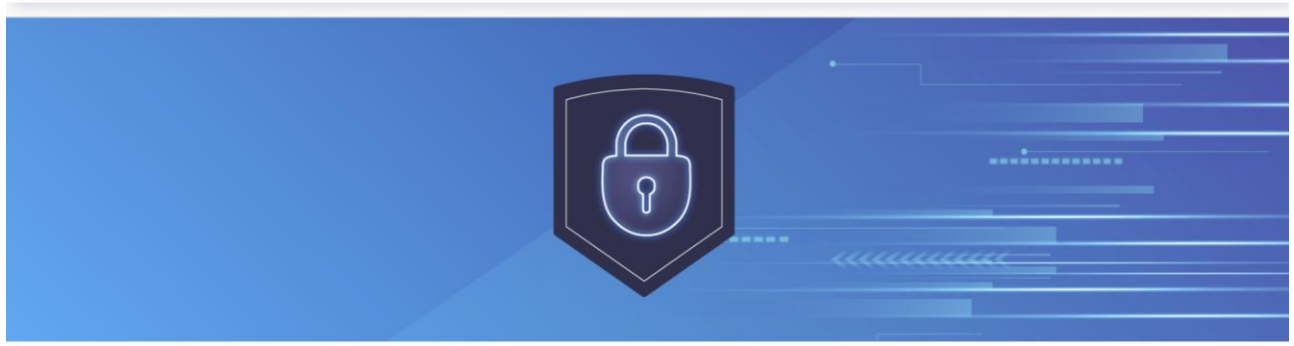
RUN-TIME ATTACKS

Insert bad code that impacts your users

PROTESTWARE

SOFTWARE SUPPLY CHAINS: THREATS

snyk



APPLICATION SECURITY | ECOSYSTEMS | OPEN SOURCE

Open source maintainer pulls the plug on npm packages colors and faker, now what?

Upon digging deeper, it turns out that the developer himself introduced an infinite loop in colors, thereby sabotaging its functionality, and purged the functional code from the **'faker' package in version 6.6.6**.

PROTESTWARE

SOFTWARE SUPPLY CHAINS: THREATS

BIG sabotage: Famous npm package deletes files to protest Ukraine war

By Ax Sharma March 17, 2022 05:51 AM 12

Protestware: Ukraine's ongoing crisis bleeds into open source

Select versions (10.1.1 and 10.1.2) of the massively popular 'node-ipc' package were caught containing malicious code that would overwrite or delete arbitrary files on a system for users based in Russia and Belarus. These versions are tracked under CVE-2022-23812.

MALWARE / SPYWARE

SOFTWARE SUPPLY CHAINS: THREATS

NPM packages posing as speed testers install crypto miners instead

By Bill Toulas

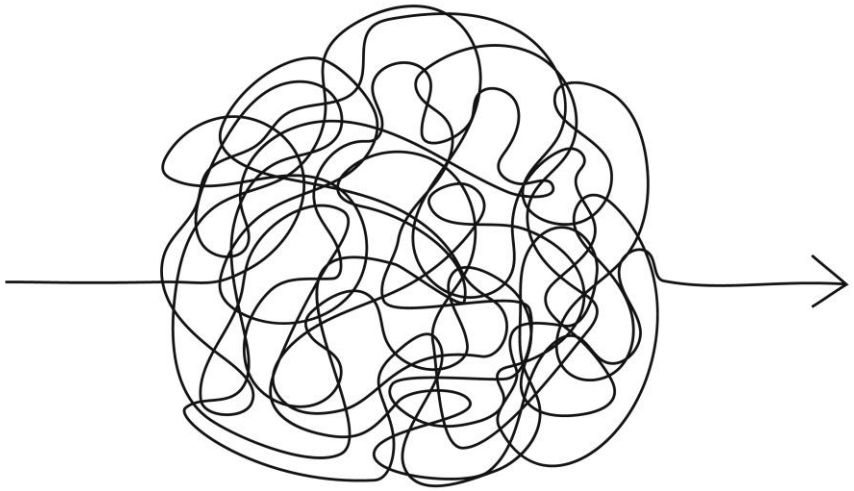
February 14, 2023 12:25 PM 0

CONFUSION ATTACKS

SOFTWARE SUPPLY CHAINS: THREATS

How do you get malicious packages installed

Trick the user!



THIS LECTURE
BUG ISOLATION

SOFTWARE SUPPLY CHAIN SECURITY

- Supply chain overview
- Threats
- Defenses

What are we supposed to do about all this trouble?

PREVENTING CONFUSION: PACKAGE NAMES

SOFTWARE SUPPLY CHAINS

Defending Against Package Typosquatting

Matthew Taylor[1], Ruturaj Vaidya[1], Drew Davidson[1], Lorenzo De Carli[2], and Vaibhav Rastogi[3⋆]

[1] University of Kansas, Lawrence, KS, USA
{mjt, ruturajkvaidya, drewdavidson}@ku.edu,
[2] Worcester Polytechnic Institute, Worcester, MA, USA
ldecarli@wpi.edu
[3] University of Wisconsin-Madison, Madison, WI, USA
vaibhavrastogi@google.com

Abstract. Software repositories based on a single programming language are common. Examples include npm (JavaScript) and PyPI (Python). They encourage code reuse, making it trivial for developers to import external packages. Unfortunately, the ease with which packages can be published also facilitates *typosquatting*: uploading a package with name similar to that of a highly popular package, with the aim of capturing some of the popular package's installs. Typosquatting frequently occurs in the wild, is difficult to detect manually, and has resulted in developers importing incorrect and sometimes malicious packages.
We present TypoGard, a tool for identifying and reporting potentially typosquatted imports to developers. TypoGard implements a novel detection technique, based on the analysis of npm and PyPI. It leverages a model of lexical similarity between names, and incorporates the notion of package popularity. It flags cases where unknown/scarcely used packages would be installed in place of popular ones with similar names, before installation occurs. We evaluated TypoGard on both npm, PyPI and RubyGems, with encouraging results: TypoGard flags up to 99.4% of known typosquatting cases while generating limited warnings (up to 0.5% of package installs), and low overhead (2.5% of package install time).

1 Introduction

Package managers automate the complex task of deploying 3rd-party dependencies into a codebase, by transitively resolving and installing all code upon which a given package—which the user wishes to install—depends. One of the most common uses of package managers is in the context of large repositories of code packages based on a single programming language. Package managers are undeniably useful, with open, free-for-all repositories like npm for Node.js, PyPI for Python, and RubyGems for Ruby, collectively serving billions of packages per week. However, they also come with problems.

The open, uncurated nature of these repositories means that any developer can upload a package with a name of their choosing. This circumstance gives rise to *typosquatting*, whereby a developer uploads a "perpetrator" package that is *confusable* with an existing "target" package due to name similarity. As a result the user, intending to install the target package, may accidentally request the confusable perpetrator package. Determining why perpetrator packages are created is a challenging and ill-defined problem, as solving it requires inferring

⋆ Currently employed at Google.

Beyond Typosquatting: An In-depth Look at Package Confusion

Shradha Neupane
Worcester Polytechnic Institute

Grant Holmes
University of Kansas

Elizabeth Wyss
University of Kansas

Drew Davidson
University of Kansas

Lorenzo De Carli
University of Calgary

Abstract

Package confusion incidents—where a developer is misled into importing a package other than the intended one—are one of the most severe issues in supply chain security with significant security implications, especially when the wrong package has malicious functionality. While the prevalence of the issue is generally well-documented, little work has studied the range of mechanisms by which confusion in a package name could arise or be employed by an adversary. In our work, we present the first comprehensive categorization of the mechanisms used to induce confusion, and we show how this understanding can be used for detection.

First, we use qualitative analysis to identify and rigorously define 13 categories of confusion mechanisms based on a dataset of 1200+ documented attacks. Results show that, while package confusion is thought to mostly exploit typing errors, in practice attackers use a variety of mechanisms, many of which work at semantic, rather than syntactic, level. Equipped with our categorization, we then define detectors for the discovered attack categories, and we evaluate them on the entire npm package set.

Evaluation of a sample, performed through an online survey, identifies a subset of highly effective detection rules which (i) return high-quality matches (77% matches marked as potentially or highly confusing, and 18% highly confusing) and (ii) generate low warning overhead (1 warning per 100M+ package pairs). Comparison with state-of-the-art reveals that the large majority of such pairs are not flagged by existing tools. Thus, our work has the potential to concretely improve the identification of confusable package names in the wild.

1 Introduction

Modern, language-based software ecosystems (LBEs) contain expansive repositories of third-party code that can be conveniently downloaded and installed by developers. The *packages*[1] of code contained in these repositories supply ready-made, diverse functionality to be used as part of a larger codebase. The popularity of package repositories is apparent through their usage: The package ecosystems npm for node.js, RubyGems for Ruby, and PyPI for Python collectively host millions of distinct packages and serve billions of package downloads weekly [66].

Tooling and automation has eased the task of finding and deploying packages. A simple invocation of the install command for the package manager frontend tool can be responsible for the cascading download of hundred of distinct packages, as (transitive) dependencies are discovered, fetched, and installed. The ease of use built into package ecosystems also increases the likelihood of a developer completing the entire installation process on a package that they did not intend to download. Should an error be made when invoking the name of an intended package, a completely different package name will downloaded and deployed. This set of circumstances allows the use of the LBE as a vector for software supply chain attacks. An adversary might publish a malicious package that attacks a developer when the package is installed, or delivers malicious functionality to end-users when the malicious package is used as part of a larger project.

In order to realize the type of incident described above, a victim developer needs to download the malicious package. Thus, the adversary's goal is to carry out a *package confusion attack*, in which a malicious package is created that is designed to be confused with a legitimate target package and downloaded by mistake. Such attacks have been shown to occur in practice [56], effecting developers that mistakenly install the package directly, and any other deployments that includes a malicious package in its transitive dependencies.

Confusion attacks can leverage the long tail distribution of packages. The top 1% most popular packages are responsible for over 99% of downloads [58]. Since LBE package managers fetch a package based on its name, the attacker can upload a package with a name that is easily confused with a legitimate popular package and passively launch the attack

[1] Although various LBEs use specialized names for the units of code that they serve, such as "gems" [2] or "crates" [11], we refer generically to each distinct unit as a package in this paper.

SCRUTINY

SOFTWARE SUPPLY CHAINS: DEFENSES

CAN WE TELL IF A PACKAGE IS BEING VETTED?

SCRUTINY
SOFTWARE SUPPLY CHAINS

CAN WE TELL IF A PACKAGE IS BEING VETTED?

(Nothing But) Many Eyes Make All Bugs Shallow

Elizabeth Wyss
University of Kansas
Lawrence, KS, USA
ElizabethWyss@ku.edu

Lorenzo De Carli
University of Calgary
Calgary, CA
Lorenzo.DeCarli@ucalgary.ca

Drew Davidson
University of Kansas
Lawrence, KS, USA
DrewDavidson@ku.edu

ABSTRACT

Open source package repositories have become a crucial component of the modern software supply chain since they enable developers to easily and rapidly import code written by others. However, low quality, poorly vetted code residing in such repositories exposes developers and end-users to dangerous bugs and vulnerabilities at a large scale.

Such issues have recently led to the creation of government-backed verification standards pertaining to packages, as well as a significant body of developer folklore regarding what constitutes a reliable package. However, there exists little academic research assessing the relationships between recommended development practices and known package issues in this domain.

Motivated by this gap in understanding, we conduct a large-scale study that formally evaluates whether adherence to these guidelines meaningfully impacts reported issues and bug maintenance activity across the most widely utilized npm packages (encompassing **7,162** packages with over **100K** weekly downloads each), which unveiled wide disparities across package-level metrics.

We find that it is only recommendations pertaining to a broad notion of *scrutiny* that provide strong and reliable insights into the reporting and resolving of package issues. These findings pose significant implications for developers, who seek to identify well-maintained packages for use, as well as security researchers, who seek to identify suspicious packages for critical observation.

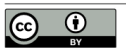
CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; **Software verification and validation**.

KEYWORDS

software supply chain, open-source, package repositories

ACM Reference Format:
Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2023. (Nothing But) Many Eyes Make All Bugs Shallow. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '23), November 30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3605770.3625216



This work is licensed under a Creative Commons Attribution International 4.0 License.

SCORED '23, November 30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0263-1/23/11.
https://doi.org/10.1145/3605770.3625216

1 INTRODUCTION

Modern software infrastructure often relies on third-party code dependencies, known as *packages*, residing in open source repositories. The rapid rise of these language-based package ecosystems highlights their widespread use and importance to the software development community. The largest of such language-based repositories is npm [70] for Node.js, which serves billions of weekly downloads of more than two million unique packages, all the while continually growing at a rate of nearly one thousand new packages per day [20].

npm's frontend package manager allows developers to easily import packages into their own codebases via a simple command line interface, which in turn has enabled wide-scale code sharing, extensive code reuse, and rapid software development cycles. Despite these advantages, this very process has also enabled the propagation of dangerous bugs and vulnerabilities, as illustrated by the almost three thousand common vulnerabilities and exposures (CVEs) which have been derived from npm packages alone [6].

Attempting to avoid these problems, the software development community surrounding npm asserts considerable claims regarding how to select a reliable package [9, 12, 54, 61, 74], but these claims are largely untested in empirical settings. In a similar vein, recent national cybersecurity initiatives [1] have led the U.S. government to issue official development standards [13], aimed at catching vulnerabilities and other software flaws.

This paper seeks to formally evaluate whether adherence to these guidelines meaningfully impacts the quality-driven outcomes of packages–including reported issues and bug maintenance activity. This particular notion of quality serves to aid users in avoiding software defects, which may expose attack surfaces to adversaries along the software supply chain.

We encounter several challenges in our work. A first challenge exists in defining a representative and effective dataset for our analysis. npm is a treasure trove of package data, but also an extremely noisy one, containing large amounts of empty and unused packages [68]. We opt to focus on the most frequently downloaded npm packages because they have the greatest overall impacts and represent the most typical use cases. Packages that garner more than 100K weekly downloads account for the majority of all package downloads in npm [68], and they are thus representative of general repository usage. Applying this insight, we extract a wide range of package metrics for 7,162 of the most widely-utilized npm packages, which each boast weekly download counts ranging from 100K to 191M. Finally, we utilize the bug maintenance activity and the issues reported against these packages to comparatively assess proposed advice and standards.

A second challenge of this study is that different software metrics may measure related, overlapping aspects, and thus may not be independent. It is important to distinguish metrics that truly

PREVENTING CONFUSION: CLONES

SOFTWARE SUPPLY CHAINS

What the Fork? Finding Hidden Code Clones in npm

Elizabeth Wyss
University of Kansas
Lawrence, KS, USA
ElizabethWyss@ku.edu

Lorenzo De Carli
Worcester Polytechnic Institute
Worcester, MA, USA
ldecarli@wpi.edu

Drew Davidson
University of Kansas
Lawrence, KS, USA
DrewDavidson@ku.edu

ABSTRACT

This work presents findings and mitigations on an understudied issue, which we term shrinkwrapped clones, that is endemic to the npm software package ecosystem. A shrinkwrapped clone is a package which duplicates, or near-duplicates, the code of another package without any indication or reference to the original package. This phenomenon represents a challenge to the hygiene of package ecosystems, as a clone package may siphon interest from the package being cloned, or create hidden duplicates of vulnerable, insecure code which can fly under the radar of audit processes.

Motivated by these considerations, we propose UNWRAPPER, a mechanism to programmatically detect shrinkwrapped clones and match them to their source package. UNWRAPPER uses a package difference metric based on directory tree similarity, augmented with a prefilter which quickly weeds out packages unlikely to be clones of a target. Overall, our prototype can compare a given package within the entire npm ecosystem (**1,716,061** packages with **20,190,452** different versions) in **72.85** seconds, and it is thus practical for live deployment. Using our tool, we performed an analysis of a subset of npm packages, which resulted in finding up to **6,292** previously unknown shrinkwrapped clones, of which up to **207** carried vulnerabilities from the original package that had already been fixed in the original package. None of such vulnerabilities were discoverable via the standard npm audit process.

ACM Reference Format:
Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2022. What the Fork? Finding Hidden Code Clones in npm. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510168

1 INTRODUCTION

The security and correctness of code stored in package repositories is an important concern because such repositories are crucial to modern software infrastructure. Indeed, language-based package repositories such as npm, pypi, and RubyGems collectively serve billions of packages each week [39]. Much of the popularity of package repositories is due to the package manager frontend, which allows a user to easily import a package by issuing a simple install directive on the command line. While seamless import of external code is convenient, it also creates problems: developers tend to assume code to be reliable rather than vetting prior to import [6], and once a package is imported, latent bugs and vulnerabilities become part of the final application. Importing the "wrong" package may cause significant supply-chain security issues [17].

This work presents findings and mitigations on an understudied issue within the npm package repository[1], which we term *shrinkwrapped clones*. We use this term to refer to packages which are uploaded to a package repository and contain code that is identical, or nearly-identical, to that of an existing (legitimate) package. Specifically, we discover two types of clones: (i) identical clones, which contain source code that is identical to that of an existing package, and (ii) close clones, which make potentially significant syntactic/semantic changes to the code, but generally localized to a small number of files (we refine this definition in the following). This phenomenon is characteristic to npm, as this ecosystem lacks the notion of forks, by which we mean copied code repositories that explicitly link back to their source repositories (as is common for example in GitHub [2]). Instead, shrinkwrapped clones in npm cannot be explicitly linked back their source packages since npm lacks official mechanisms for forking packages.

The phenomenon of shrinkwrapped clones represents a challenge to the hygiene of package repositories; in particular, it contributes to the problem of *confusability* of npm packages. npm contains more than 1.7 million packages, and while the ecosystem provides a robust search interface, it provides no assistance in choosing the most appropriate package to provide a desired functionality. Previous work on typosquatting attacks [39] suggests that it is fairly common for developers to install a package different from the one they intended.

Clones exacerbate these problems. Most obviously, a clone package causes confusion, as clone packages are oftentimes named similarly to the original package, and in many occasions they also reuse their metadata (such as the package description). Users of the package repository may thus misattribute the provenance of a package, giving credit to the wrong developer for creating a particular codebase. However, we also find that a non-trivial number of clone packages are rarely maintained and fail to include updates to the package being copied. The users of the clone are thereby locked into older versions of functionality and, crucially, forgo any bug fixes that are applied to the package being cloned. In effect, the users of the clone are subject to vulnerabilities which have already been patched. Moreover, clones can exist deep within package dependency trees, which means that installing a package that (transitively) depends on a clone also implicitly installs

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510168

[1] We choose npm as our repository of interest because it is the largest and most popular.

PREVENTING CONFUSION: CLONES

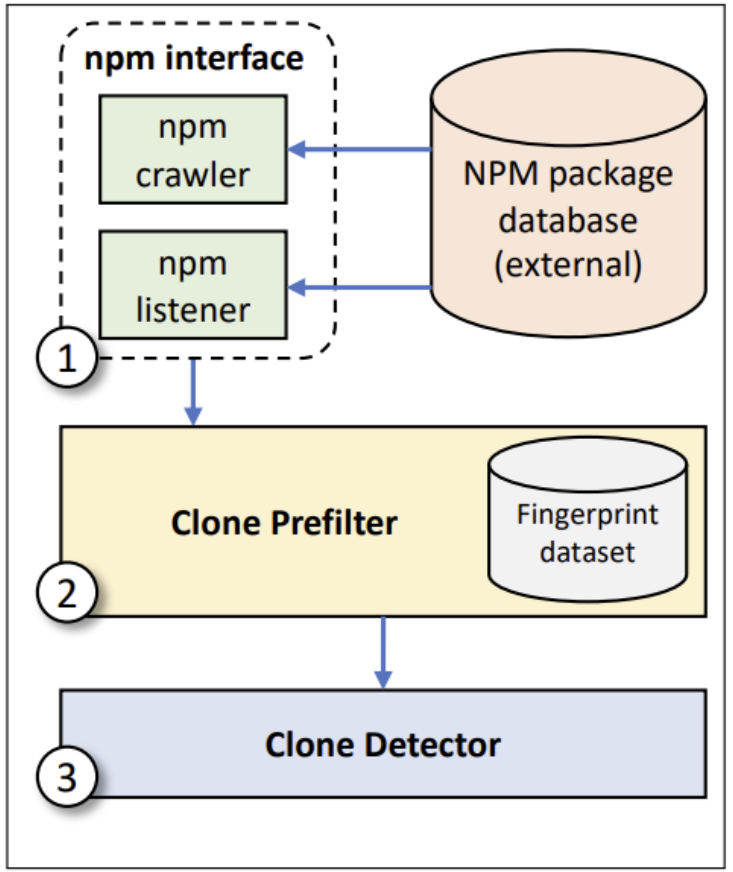
SOFTWARE SUPPLY CHAINS

Table 5	**Clone Vulnerability & Popularity**	
Clone Type	**Identical (348)**	**Close (5,944)**
Likely Downloaded	21	399
More Vulnerable	62	2,304
Likely Downloaded AND More Vulnerable	4	148
More Vulnerable AND Vulnerabilities Undetected by Audit	17	190
Likely Downloaded AND More Vulnerable AND Vulnerabilities Undetected by Audit	0	8

Table 5: Measured popularity and vulnerability statistics of identical clones and close clones

PREVENTING CONFUSION: CLONES

SOFTWARE SUPPLY CHAINS



PREVENTING INSTALL-TIME EXPLOITS

SOFTWARE SUPPLY CHAINS

Allow package users and repository maintainers to specify acceptable install-time behavior

Wolf at the Door:
Preventing Install-Time Attacks in npm with LATCH

Elizabeth Wyss
University of Kansas
Lawrence, KS, USA
ElizabethWyss@ku.edu

Alexander Wittman
University of Kansas
Lawrence, KS, USA
wittmanalex@gmail.com

Drew Davidson
University of Kansas
Lawrence, KS, USA
DrewDavidson@ku.edu

Lorenzo De Carli
Worcester Polytechnic Institute
Worcester, MA, USA
ldecarli@wpi.edu

ABSTRACT

The npm software ecosystem allows developers to easily import code written by others. However, manual vetting of every individual installed component is made difficult in many cases by the number of transitive dependencies brought in by installing popular packages. This has enabled attackers to propagate malicious code by hiding it deep into the dependency chains of popular packages. A particularly dangerous form of attack comes from malicious code embedded into package install scripts.

We tackle the problem of preventing undesirable install-time behavior by proposing LATCH, a system for mediating install-time capabilities of npm packages. LATCH generates permission manifests summarizing each package's install-time behavior and checks them against user-defined policies to ensure compliance. Policies in LATCH are expressed in a rich formal policy language that covers a broad range of use cases. Our key insight is that expressive LATCH policies empower users to define and enforce their own individualized security needs.

Evaluation of practical LATCH policies on all publicly available npm packages and on a number of real-world attack packages demonstrates that our approach is effective in identifying and stopping unwanted behavior while minimizing disruption due to undesired alerts.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Empirical software validation**.

KEYWORDS

npm, supply chain security, install-time attack, policy language

ACM Reference Format:
Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. 2022. Wolf at the Door: Preventing Install-Time Attacks in npm with LATCH.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASIA CCS '22, May 30-June 3, 2022, Nagasaki, Japan
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9140-5/22/05.
https://doi.org/10.1145/3488932.3523262

In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30-June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3488932.3523262

1 INTRODUCTION

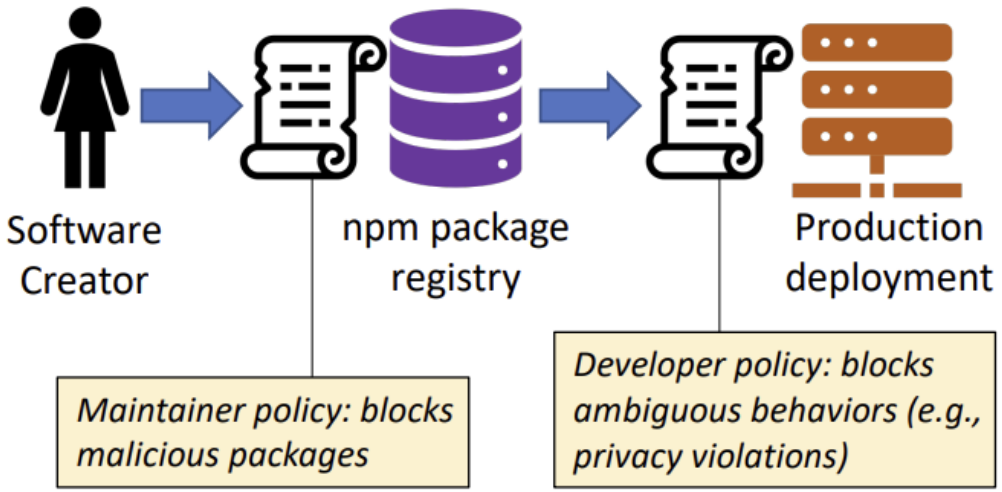
Many programming language ecosystems benefit from public repositories that allow any developer to upload a package that contains modular functionality for use in other software projects. Although these package repositories are undeniably useful, they can also be a vector for a *software supply chain* attack. In such an attack, the larger system is compromised through the functionality of an imported component, such as a package. Recently publicized software supply chain attacks have resulted in execution of crypto-mining code [51], exfiltration of credentials and other sensitive information [20], and other unwanted outcomes. While there are many ways in which software supply chain issues surface, one of the most dangerous involves exploiting the install-time package setup mechanism. In many repositories, packages are equipped with routines that allow for bootstrapping. Scripts embedded within these bootstrap mechanisms, under most conditions, execute when the package is installed. As a result, an attack can complete even if the actual package is *never run or imported* by the victim. In practice, setup scripts have been used as a vector for a variety of malicious behaviors [20, 29, 49–51], and there have been over one hundred documented attacks built upon this technique [13, 31].

Even when a package's installation scripts are not explicitly designed to do harm, they may still exhibit behavior that some developers would consider undesirable if they are poorly written or perform unnecessary operations. Potentially undesirable install-time operations have been discovered in many repository packages, including sending machine specifications, machine identifiers, and lists of installed packages to remote tracking APIs [24].

In this work, we tackle the risks of install-time software supply chain compromise by proposing LATCH (**L**ightweight inst**A**ll-**T**ime **CH**ecker), a system capable of (i) capturing in a succinct but expressive manner the operations performed at install-time by any given package, and (ii) matching those operations to user-defined security policies, flagging cases where package behavior violates the intended policy. Thus, our approach defines a set of install-time capabilities, and precisely bounds the install-time behavior of each package within the set of allowed capabilities.

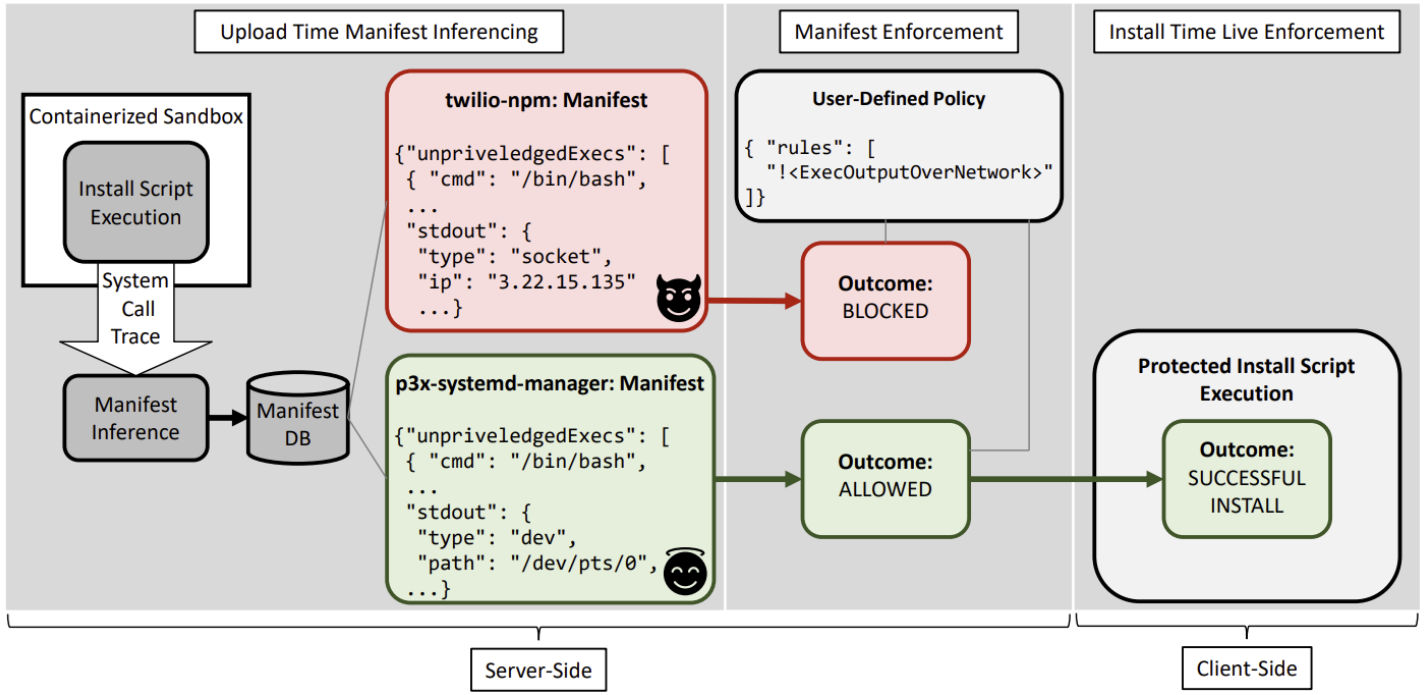
PREVENTING INSTALL-TIME EXPLOITS

SOFTWARE SUPPLY CHAINS



PREVENTING INSTALL-TIME EXPLOITS

SOFTWARE SUPPLY CHAINS



WRAP-UP

SOFTWARE SUPPLY CHAINS

An application is more than your code

Likely ***mostly*** somebody else's code!

- Very little vetting
- Very little protection