Chapter 37
Dependency Analysis Tools

Revision 1.0, December 2025.

Objectives

- Understand Software Supply Chain Resource Management.
- Learn about Software Bill of Materials.
- Understand sources of vulnerability data.
- Learn about the CVE, CPE, CWE and CVSS.
- Understand practical issues in using dependency analysis tools.

37.1 Introduction

This chapter presents an important class of tools that you can use to understand problems in your software supply chain. These tools are typically called "dependency analysis tools" or "dependency tools" for short. They are also known as "software composition analysis tools".

The main function of these tools is to tell you if there are any known vulnerabilities in the packages or modules that you are including in your software project.

37.2 Supply Chain Resource Analysis

We will start by reviewing the concept of supply chain resource management or SCRM. And to do so, we will look at a particularly complex machine, a jumbo Boeing 747, as shown in Figure 1.

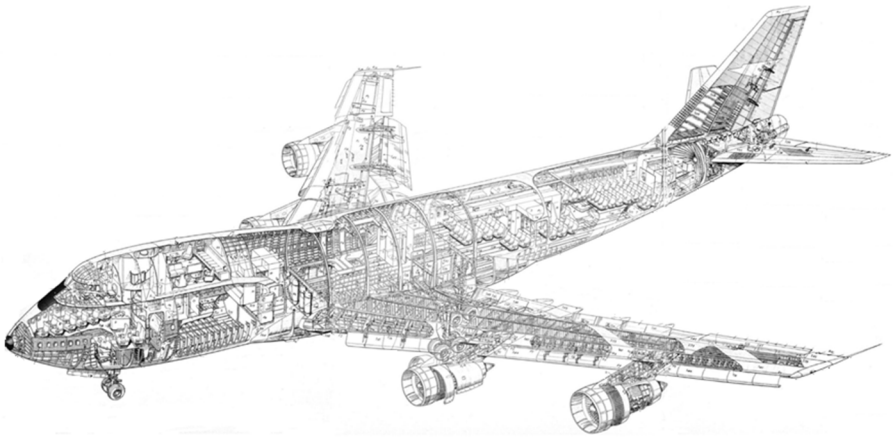

Figure 1: Structural Drawing of a Boeing 747

The complexity of this impressive machine is obvious when we see that it is made up of 6 million parts from hundreds of different suppliers. Just ordering these parts and assembling this plane produces significant logistical challenges.

Now, supposed that we hear from one our suppliers that a certain type of screw has a serious manufacturing flaw that introduces a high potential for failing during installation or operation. We are now faced with the challenge of finding all the flawed screws in the completed (or partially completed) aircraft, understanding the risk of failure in each case, and then organizing a strategy for replacing these flawed screws.

We first need to understand which aircraft were affected. Many makes and models of aircraft might have included this type of screw. So, we need to figure out which aircraft included this particular batch of flawed screws.

Understanding the risk associated with each flawed component is also a critical step, as it tells us the urgency of the repair. For example, does the FAA, EASA or JCAB need to issue an emergency Airworthiness Directive immediately grounding all the affected aircraft?

37.3 Software Supply Chain Resource Management

As you can imagine, the software world has many of the same supply chain issues and needs as the aviation world. Let's Look at how this might work for the "myapp" Python application who package structure is illustrated in Figure 2. As we can see, myapp makes use of several modules, including the four listed here. So, any security risk that appears in one of these modules potentially puts the entire application at risk.

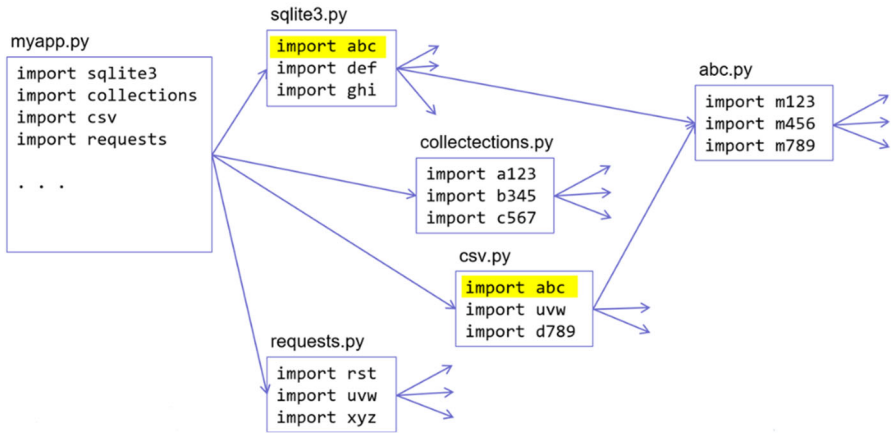

Figure 2: Software Dependence Graph for a Python Application

And we can see that each of the included modules may, themselves, include other modules, thereby introducing vulnerabilities when one of these packages has a serious flaw. Any of these second-level imported modules can, themselves, have their dependencies. We have to detect the full transitive closure of dependencies to understand all the sources of vulnerabilities in our application.

This list of software dependencies for our application has come to be termed the *software bill of materials* (SBOM, pronounced ess-bomb). The SBOM includes all the packages on which this application is dependent, along with several other useful details, including the provider of the software package, platform on which it can run, version number, and date of this version. Version number is particularly important so that we can determine if a flawed version of a package is the one that's actually included in my application.

There are other components that can be included in the SBOM. These include the build information for the application, licenses, and the tool chain that compiled and linked the application.

While software security practitioners have been worried about the software supply chain for quite a while, the issue gained significant public visibility in 2020 with the attack on Solar Winds Orion by the APT 29 ("Cozy Bear) attack team. As a result of this attack, there was the subsequent U.S. Presidential Executive Order 14028 in May 2021 requiring a focus on software supply chain issues.

Note that supply chain issues can appear in any programming language. While the syntax and terminology of including a package or module or library may be different, the concept is the same. Figure 3 shows the variety of terms for nine programming languages in common use.

Python modules	`import mod`	C/C++ libraries	`#include <lib>`
Perl modules	`use mod`	C# libraries	`using`
Ruby modules	`include mod`	Rust modules	`use mod`
Java packages	`import pack`	Go modules	`import "mod"`

Figure 3: Language Specific Terminology for Modules and Importing

It should be clear now why a software bill of materials is an important element to include in your development practices. We see the reasons called out explicitly in a U.S. Department of Commerce report[1]. As with our airplane example, this report says that we need to answer two key questions:

[1] `https://www.ntia.gov/SBOM`

First, are we affected by a known vulnerability and second, where is the flawed software being used?

Are we affected? To answer this first question, we have to know some information about the package or module that has the known vulnerability. This information includes the name of the package, the provider, and the version number (or numbers) that we are actually using. We are also interested in whether there is a new version of the package that fixes the vulnerability.

Where is the flawed software being used? To answer this second question, we need to understand our use of the flawed package. We need to know which version or versions have the vulnerability and is there a newer version that has fixed the vulnerability? Next, we need to know where in our application we are using this package and how much of the package are we using. If we are using only a small part of the package or there is another equivalent package out there, we might be able to code around the use of this package or replace it entirely.

37.4 Vulnerability Databases

A key resource for determining whether there is a vulnerability in a dependency is the vulnerability database. This database stores a list of known vulnerabilities, where each vulnerability is given a unique label in a standard form such as a CVE number. (More about CVEs in the next section.) The database is used by the dependency tool. The tool first constructs the list of dependencies (the bill of materials) and then looks for known vulnerabilities in each dependency.

The quality of a given vulnerability database is determined by its completeness. Certainly, the database that includes the most vulnerabilities is the best. It is also affected by the timeliness of the data. We would like the vulnerability data to appear in the database as close as possible to the time that the vulnerability is discovered. So a dependency tool can only be as good as the data that it is provided.

The earliest and most well-known of the vulnerability databases is the NVD, the National Vulnerability Database[2] operated by the U.S. National Institute of Standards and Technologies or NIST. The NVD is publicly available and used by a wide variety of tools, including OWASP Dependency Check and Snyk.

[2] `https://nvd.nist.gov`

More recently, GitHub created its own vulnerability database, the GitHub Advisory Database[3]. This data includes the NVD and vulnerabilities identified by GitHub.

And there are a variety of purely proprietary databases. These databases allow the different tool vendors to offer increased value over using just the NVD or over the data offered by their competitors. If profit was not the motive, then each of these vendors would just contribute their data to the NVD for all to use.

37.5 An Alphabet Soup of Key Acronyms

There is a collection of acronyms critical the accurate reporting of vulnerability data. Each of these acronyms represents a key concept in vulnerability reporting and defines a standard format for that concept. Standards are important so that people and tools can understand vulnerability information that is coming from a variety of sources around the world.

37.5.1 Common Vulnerabiliti3es and Exposures (CVE)

We start with the CVE, the Common Vulnerabilities and Exposures labeling. For a given vulnerability in a software package, the CVE provides a unique way to name that vulnerability and associate important information with it. For each vulnerability report, in other words each CVE, the information can include:

- Description of the vulnerability
- Standard name for the software and version affected (called the CPE)
- Standard name for the code flaw or weakness that allowed the vulnerability (called the CWE)
- Rating of the severity or seriousness of the vulnerability (called the CVSS)
- Possible workarounds for the vulnerability
- Description of the exploit for this vulnerability (though this item is often unsatisfyingly vague).

It would be great if every CVE had a complete report of the vulnerability, including enough detail to understand its exact cause (location in the code) and fix. However, the typical CVE report is woefully incomplete in this sense.

CVEs are added to the database as they are discovered and reported. As of the end of 2025, there were more than 323,000 CVEs. As an example CVE

[3] `https://github.com/advisories`

name, the highly publicized Log4J vulnerability[4] is identified as CVE-2021-44228.

The importance of the CVE is that two tools that report the same CVE are talking about the exact same problem.

37.5.2 Common Platform Enumeration (CPE)

The CPE, or Common Platform Enumeration, provides a way to precisely name the software package that contains the vulnerability. This description includes the vendor who produced the software, the vendor-provided name for the package, a vendor-specific version number, an edition (which can be something like a build number), and a language tag (if applicable).

So, two software packages with the same name should be the same package. However, two packages with different Project name fields can actually be the same package if the vendor is careless. For example, both "WindowsXP" (without a dash) and "Windows-XP" (with a dash) might name the same software package. Additionally, the name might appear in two different languages, such as US English and Taiwanese Chinese, making it appear that these two names refer to two different packages.

37.5.3 Common Weakness Enumeration (CWE)

When a vulnerability is found, it is due to some error, or flaw, or more properly, weakness in the code. As we discussed in our introductory unit on Basic Concepts and Terminology, a Common Weakness Enumeration, or CWE, is used the described the different types of coding errors that might lead to vulnerabilities. As of CWE version 4.19, there were 944 weaknesses identified, organized into a somewhat reasonable, but sometimes confusing hierarchy.

A CWE might include the unique ID, description, platform to which it applies, possible consequences of the weakness, examples of the weakness, and potential ways to mitigate it.

So, two weaknesses with the same CWE should describe the same type of error in the code. This allows results reported by different tools on different packages to be compared. However, we have to note that the choice of a CWE can sometimes seem confusing or arbitrary. For example, CWE-787 is an out-of-bounds write and CWE-121 is a stack-based buffer overflow. So, given two similar coding errors, two different programmers might choose different CWE numbers.

---

[4] https://nvd.nist.gov/vuln/detail/cve-2021-44228

One interesting use of CVE reports is to form a list of the most common coding mistakes found in software in a given year. This list is produced by OWASP by mining the available vulnerability databases and extracting the CWE for each reported vulnerability[5]. It provides interesting insights into the most common security coding mistakes that programmers are currently struggling with.

Rank	ID	Name
1	CWE-79	Cross-site Scripting (XSS)
2	CWE-89	SQL Injection
3	CWE-352	Cross Site Request Forgery (CSRF)
4	CWE-862	Missing Authorization
5	CWE-787	Out-of-bounds Write
6	CWE-22	Directory Traversal (Path Name Injection)
7	CWE-416	Use After Free
8	CWE-125	Out-of-bounds Read
9	CWE-78	OS Command Injection
10	CWE-94	Code Injection
11	CWE-120	Classic Buffer Overflow
12	CWE-434	Unrestricted Upload of File with Dangerous Type
13	CWE-476	NULL Pointer Dereference
14	CWE-121	Stack-based Buffer Overflow
15	CWE-502	Deserialization of Untrusted Data
16	CWE-122	Heap-based Buffer Overflow
17	CWE-863	Incorrect Authorization
18	CWE-20	Improper Input Validation
19	CWE-284	Improper Access Control
20	CWE-200	Exposing Sensitive Information to Unauthorized Actor
21	CWE-306	Missing Authentication for Critical Function
22	CWE-918	Server-Side Request Forgery (SSRF)
23	CWE-77	Command Injection
24	CWE-639	Authorization Bypass Through User-Controlled Key
25	CWE-770	Allocation of Resources Without Limits or Throttling

Figure 4: OWASP CWE Top 25 List as of 2025

[5] https://cwe.mitre.org/top25/

The Top 25 list for 2025 is shown in Figure 4. We have color coded it to group the CWE reports into a few interesting categories:

Web:	1, 3, 22
Injection:	2, 6, 9, 10, 15, 23
Memory:	5, 7, 8, 13, 14, 16

There are a couple of things to note about this Top 25 data. First, you will find chapters in this book on almost every CWE type. Second, easy to fix things like SQL Injection (number two on the list!) are still quite prominent in the real world code. Third, the memory weakness category is primarily due to C and C++ code. Even in this age of interpreted languages (like Java, C#, Python, Ruby, and Perl) and modern systems languages (like Go and Rust) with strong bounds checking, memory errors are too common.

37.5.4 Common Vulnerability Scoring System (CVSS)

For each vulnerability that is found, there may be a numeric report (or score) that allows us to understand various characteristics of the vulnerability. The Common Vulnerability Scoring System or CVSS is the most common way to do this. The CVSS can include information on what kind of access was needed to conduct the attack; the privilege level of the attacker; the difficulty of conducting the attack; the impact of the attack on confidentiality, integrity and availability; whether there is example exploit code; and whether there is a known fix for the vulnerability.

Figure 5 contains the main elements from CVSS version 3.1. Once you have assigned a value for each CVSS element, you can use a score calculator[6] to assign a numerical value. These scores also allow a dependency tool to sort their output in descending order of severity.

Attack vector:	Physical / Local / Adjacent network / Network
User interaction:	None / Required
Privileges required:	None / Low / High
Attack complexity:	None / Low / High
Confidentiality impact:	None / Low / High
Integrity impact:	None / Low / High
Availability impact:	None / Low / High
Exploit code maturity:	Unproven / Proof-of-concept / Functional / High
Remediation level:	Official fix / Temporary fix / Workaround / Unavailable
Report confidence:	Unknown / Reasonable / Confirmed

Figure 5: Element of CVSS Version 3.1

[6] `https://www.first.org/cvss/`

37.6 Practical Issues for Dependency Analysis Tool Use

Dependency analysis tools work by first building the SBOM and then looking for known vulnerabilities in each component of the SBOM.

We note that these tools should be run on a regular basis, even more often than you build your software. New vulnerabilities can appear at any time and can be found in even well-established software packages at any time.

37.6.1 Example of Source of SBOM Data: pom.xml File from Maven

One of the main sources of information used to build the bill of materials is a "manifest" file. This file contains information about a project's dependencies and the version (or versions) of the packages used. Some examples of manifest files are the pom.xml for Maven, Gemfile.lock for Ruby and yarn.lock for JavaScript.

Figure 6 is a poml.xml file used by Maven to build the Signal-Server messaging server package. In most cases, you will rarely need to look at it yourself, but it is not hard to understand.

While there are a lot of details in this file, the most interesting part is the list of dependencies along with the version numbers for each dependency. In Figure 6, we can see this list of dependencies starting at ❶ and the dependency version numbers at ❷.

Other types of manifests encode this data in their own ways, but the basic idea is the same.

37.6.2 Local vs. Repository Tool Execution

When you want to run a dependency tool, you have two basic ways to do it. The easiest way is to run the tool against your code repository. For example, GitHub has integrated the Dependabot tool[7] so that you do not have to do any set up. You can either trigger the tool yourself or GitHub will run it from time to time and send you a report if any vulnerable dependencies are found.

A second way to run a tool is download it to your own computer and run it against the code in your development directory. Note that since the code is actually being built and run on your computer, there may be more information available to a locally run tool.

[7] `https://github.com/dependabot`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <packaging>pom</packaging>
  <prerequisites>
    <maven>3.0.0</maven>
  </prerequisites>

  <properties>
    <dropwizard.version>2.0.21</dropwizard.version>
    <aws.sdk.version>1.11.939</aws.sdk.version>
  </properties>

  <groupId>org.whispersystems.textsecure</groupId>
  <artifactId>TextSecureServer</artifactId>
  <version>1.0</version>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>io.dropwizard</groupId>
        <artifactId>dropwizard-dependencies</artifactId>
        <version>${dropwizard.version}</version>
        <type>pom</type>
        <scope>import</scope>
```

Figure 6: Example Manifest, a pom.xml File for Maven

Tools integrated with a repository are usually added as an application to your project's repository. This is easy to set up, but you have limited control over configuring the tool and how it runs. Locally run tools scan code stored on your computer. These tools often come with plugins for a variety of programming languages. While they offer greater flexibility than integrated tools, they also can be harder to set up.

37.6.3 What the Tools Report

The main job of these tools is to provide a list of dependencies that are vulnerable, along with information on how to fix the vulnerabilities, including newer versions of the packages in which the vulnerability has been fixed.

Many of these tools label the vulnerabilities with their corresponding CVE. For example, we can get CVE numbers from OWASP Dependency Check, Snyk, and Dependabot. However, for vulnerabilities that appear only in a proprietary database, there will instead be a vendor-specific vulnerability ID.

Some tools will also try to describe the impact or risk associated with the vulnerability by including a score such as a CVSS.

For dependency analysis tools, and other tools such as static analysis tools, we know that no tool is perfect. Formally, we can say that they are neither *sound* nor *complete*. Sound means that anything that the tool tells you is true. Complete means that the tool did not miss anything. This means that tools may present us with *false positives*, which are reports of a vulnerability where none really exists. There may also be *false negatives*, which is when there is no report even though a vulnerability does exist. And information can simply be wrong in the database or in the labeling of the package or vulnerability.

The good news is that these tools are correct much more often than they are incorrect, so do not let these issues dissuade you from using them. Use these issues to make yourself more effective in using the tools.

37.6.4 Best Practices in Using Dependency Analysis Tools

We have learned some lessons over the years and share these lessons with you in the hope that they will make you more effective in using dependency tools.

1. The tool that reports the most issues for your package isn't necessarily the best tool. You need to look carefully to make sure that all the reports are true ones. Then compare the tools based on how many true reports they present.
2. Make sure to keep your tools up to date, as they are improving all the time. We have seen some dramatic improvements in a tool in just one new release.
3. As with any tool, no one tool may be best in all situations. You can increase your confidence in the results by running more than one tool.
4. Pay attention to the tool's configuration options and settings. These can have a big impact on their effectiveness.
5. Sometimes the results look worse than they really area. There may be several places in your code base that use the same vulnerable package, so you might get multiple reports for the same vulnerability.

6. Not all vulnerabilities will have a CVE. A vulnerability report without a CVE is just as valid as one with a CVE. It just indicates that the tool vendor did not share this information with the NVD.
7. Know that the tools are not perfect and will miss things. Dependency analysis tools are more effective when used in conjunction with static analysis tools, strong secure coding practices, and code reviews.

37.7 Summary

We have learned how dependency tools work and about the databases that provide vulnerability data. As part of that data, we have learned about the standard formats used for reporting vulnerabilities, including the CVE, CWE, CPE, and CVSS. We then discussed best practices for using these tools. These tools are so easy to use that f you are not already using them, today is a good day to start!

37.8 Exercises

1. Visit the NVD, `https://nvd.nist.gov/`, and browse through the list of recent vulnerabilities. Select one that interests you.
    a. Study the components of your selected vulnerability. Are the CWE, CPE, and CVSS identifiable and complete?
    b. Is there enough information contained in the report to identify exactly what was wrong in the code that allowed the vulnerability? (The answer will probably be "no". In that case, move on to the next part of this question.)
    c. Search the Web for the CVE number and read the related sources and articles. Look for reports from security blogs and security companies, and for commit logs for the updates that fixed the vulnerability. The goal is to see if you can find and identify the exact code in the application that contained the weakness.
2. Visit the CWE website maintained by the MITRE Corporation, `https://cwe.mitre.org/`.
    a. Look at the CWE entries related to buffer overflow. Compare CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), CWE-121 (Stack-based), CWE-122 (Heap-based), and CWE-120 (Classic). When would you use each one? Are all these needed or are any of them redundant. Justify your answer.
    b. Look at CWE-77 (Command Injection) and CWE-78 (OS Command Injection). Are they both needed? Go to the NVD and find a vulnerability report that is based on each

CWE. Are the CWE entries assigned correctly to the CVE? Explain.

3. Visit the CVSS website, `https://www.first.org/cvss/`.
    a. Look at the two most recent version of CVSS, v3.1 and v4.0. How do they differ? What changed between these two versions?
    b. Why were these changes made? List positive and negative responses to these changes.