

Chapter 17. Exploiting Third-Party Dependencies

It's no secret that the software of today is built on top of OSS. Even in the commercial space, many of the largest and most profitable products are built on the back of open source contributions by a large number of developers throughout the world.

Some products built on top of OSS include:

- Reddit (BackBoneJS, Bootstrap)
- Twitch (Webpack, nginx)
- YouTube (Polymer)
- LinkedIn (EmberJS)
- Microsoft Office Web (Angular)
- Amazon DocumentDB (MongoDB)

Beyond simply being OSS reliant, many companies now make their core products available as open source software and make revenue with support or ongoing services instead of by selling the products directly. Some examples of this are:

- Automattic Inc. (WordPress)
- Canonical (Ubuntu)
- Chef (Chef)
- Docker (Docker)
- Elastic (Elasticsearch)
- Mongo (MongoDB)
- GitLab (GitLab)

BuiltWith is an example of a web application that fingerprints other web applications in an attempt to determine what technology they are built on

top of (Figure 17-1). This is useful for quickly determining the technology behind a web application.

Reliance on OSS, while convenient, often poses a significant security risk. This risk can be exploited by witty and strategic hackers. There are a number of reasons why OSS can be a risk to your application’s security, and all of them are important to pay attention to.



Figure 17-1. BuiltWith web application

First off, relying on OSS means relying on a codebase that probably has not been audited to the same stringent lengths that your own code would

be. It is impractical to audit a large OSS codebase, as you would first need to ramp up your security engineers enough to become familiar with the codebase, and then you would need to perform an in-depth, point-in-time analysis of the code. This is a very expensive process.

A point-in-time analysis is also risky because OSS codebases are constantly being updated. Ideally, you would also perform a security assessment of each incoming pull request. Unfortunately, that would also be very expensive, and most companies would not support that type of financial loss and would rather shoulder the risk of using relatively unfamiliar software.

For these reasons, OSS integrations and dependencies are an excellent starting point for a hacker looking to break into someone's software. Remember, a chain is only as strong as its weakest link, and often the weakest link is the one that was subjected to the least-rigid quality assurance.

As a hacker, the first step in finding OSS integrations or dependencies to exploit is recon. After recon, exploitability of these integrations can come from a number of different angles.

Let's investigate OSS integrations a bit further. First, we want to gain some understanding of how web applications integrate with OSS.

Once we understand the basics as to how these integrations take place, we can perform further investigations into the risks of OSS integrations. We can then learn how to take advantage of OSS integrations in a web application.

Methods of Integration

When the developer of a web application wishes to integrate with an OSS application, there are often a few ways they can go about it from an architectural perspective. It is important to know how an integration between a web application and an OSS package is structured, as this often dictates the type of data moving between the two, the method by which

the data moves, and the level of privilege the OSS code is given by the main application.

Integrations with OSS can be set up many different ways. An extremely centralized case involves direct integration into the core application code. Or it can involve running the OSS code on its own server and setting up an API for one-way communication from the main application to the OSS integration (this is the decentralized approach). Each of these approaches has pros and cons, and both bring different challenges to anyone attempting to secure them.

Branches and Forks

Most of today's OSS is hosted on Git-based version control systems (VCSs). This is a major difference between modern web applications and legacy web applications, as 10 years ago the OSS might have been hosted in Perforce, Subversion, or even Microsoft's Team Foundation Server.

Unlike many legacy VCSs, Git is distributed. That means that rather than making changes on a centralized server, each developer downloads their own copy of the software and makes changes locally. Once the proper modifications are made on a "branch" of the main build, a developer can merge their changes into the main branch (single source of truth).

When developers take OSS for their own use, sometimes they will create a branch against that software and run the branch they created instead of the main branch. This workflow allows them to make their own modifications, while easily pulling in changes pushed to the main branch by other developers.

The branching model comes with risks. It can be much easier for a developer to accidentally pull unreviewed code from the main branch into their production branch.

Forks, on the other hand, offer a greater level of separation, as forks are new repositories that start at the last commit pushed to the main branch prior to the fork's creation. As a new repository, a fork can have its own


permissions systems, its own owner, and implement its own Git hooks to ensure that accidentally insecure changes are not merged.

A con of using a forking model for deploying OSS is that merging code from the original repo can become quite complex as time goes on, and the commits need careful cherry-picking. Sometimes commits from the main repo will no longer be compatible with the fork if significant refactoring occurred after the fork was created.

Self-Hosted Application Integrations

Some OSS applications come prepackaged, often with simple setup installers. A prime example of this is WordPress (see [Figure 17-2](#)). It started out as a highly configurable PHP-based blogging platform, and it now offers simple one-click installation on most Linux-based servers.

Rather than distributing WordPress by source code, WordPress developers suggest downloading a script that will set up a WordPress installation on your server automatically. Run this script, and the correct database configuration will be set up, and files will be generated specifically based on the configuration presented to you in a setup UI.



Below you should enter your database connection details. If you're not sure about these, contact your host.

Database Name	<input type="text" value="wordpress"/>	The name of the database you want to use with WordPress.
Username	<input type="text" value="username"/>	Your database username.
Password	<input type="text" value="password"/>	Your database password.
Database Host	<input type="text" value="localhost"/>	You should be able to get this info from your web host, if localhost doesn't work.
Table Prefix	<input type="text" value="wp_"/>	If you want to run multiple WordPress installations in a single database, change this.

Figure 17-2. WordPress—the most popular CMS on the internet

These types of applications are the most risky to integrate into your web application. It may sound like a simple one-click-setup blogging software could not cause a lot of trouble, but more often than not, this type of system makes it much more difficult to find and resolve vulnerabilities later on (you won't know the location of all the files without significant effort in reverse engineering the setup script). Generally, you should stay away from this deployment method, but when you must go down this route, you should also find the OSS repository and carefully analyze the setup script and any code run against your system in this repository.

These types of packages require elevated privileges and could easily result in a backdoor RCE. This could be detrimental to your organization as a result of the script itself likely running as an admin or elevated user on your web server.

Source Code Integration

Another method by which OSS can be integrated with a proprietary web application is via direct source, code-level integration. This is a fancy way of saying copy/paste, but can often be more complex; a large library

might also require its own dependencies and assets to be integrated alongside it.

This method requires quite a bit of work up front when dealing with large OSS libraries, but it is very simple with smaller OSS libraries. For a short 50–100 line script, this is probably the ideal method of integration. Direct source code integration is often the best choice for small utilities or helper functions.

Larger packages are not only more difficult to integrate but also come with more risks. The forking and branching models bring risk; insecure upstream changes may accidentally be integrated into the OSS code that integrates with your web application. The direct integration method has its own risks; there is no easy way to be notified of upstream fixes, and pulling that patch into your software could be difficult and time-consuming.

Each of these methods has pros and cons, and there is no correct method for every application. Make sure to carefully evaluate the code you wish to bring in and integrate by a number of metrics, including size, dependency chain, and upstream activity in the main branch.

Package Managers

Today, many integrations between a proprietary web application and OSS happen as a result of an intermediary application called a package manager. Package managers are applications that ensure your software always downloads the correct dependencies from reliable sources on the web and sets them up correctly so they can be consumed from your application regardless of the device your application is run on.

Package managers are useful for a number of reasons. They abstract away complicated integration details, slim down the initial size of your repository, and if correctly configured, can allow only the dependencies you require for your current development work to be pulled in rather than pulling in every dependency for a large application. In a small application this may not be useful, but for a large enterprise software package

with over a hundred dependencies, this could save you gigabytes of bandwidth and hours of build time.

Every major programming language has at least one package manager, many of which follow similar architectural patterns to those in other languages. Each major package manager has its own quirks, security safeguards, and security risks. We cannot evaluate each and every package manager in this chapter, but we can analyze a few of the most popular ones.

JavaScript

Until recently, the JavaScript (and Node.js) development ecosystem was built almost entirely on a package manager called npm (see [Figure 17-3](#)).

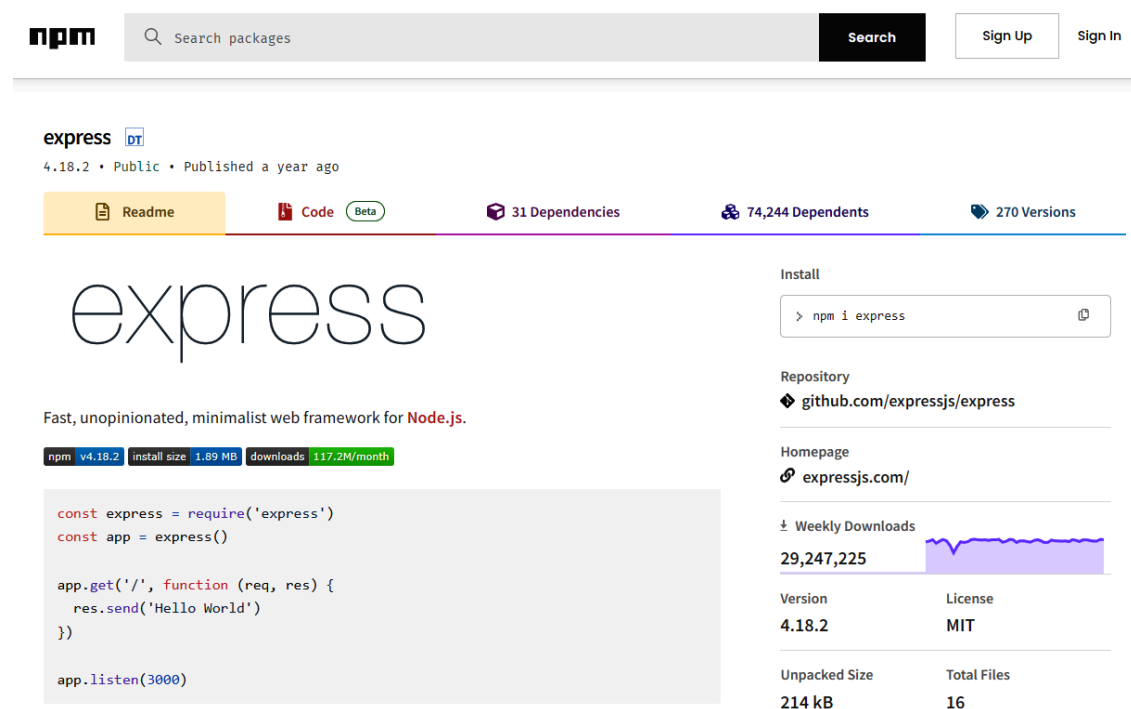


Figure 17-3. npm, the largest JavaScript-based package manager

Although alternatives have popped up on the market, npm still powers the vast majority of JavaScript-based web applications around the web. [npm](#) exists in most applications as a CLI for accessing a robust database of open source libraries that are hosted for free by npm, Inc.

You have probably run into an npm-based application by accident or on purpose. The key signs that an application brings in dependencies via

npm are the *package.json* and *package.lock* files in an application's root directory, which signal to the CLI which dependencies and versions to bring into the application at build time.

Like most modern package managers, npm not only resolves top-level dependencies, it also resolves recursive child dependencies. This means that if your dependency also has dependencies on npm, npm will bring those in at build time too.

npm's loose security mechanisms have made it a target for malicious users in the past. Due to its widespread usage, some of these events have affected the uptime of millions of applications.

An example of this was *left-pad*, a simple utility library maintained by one person. In 2016, *left-pad* was pulled from npm, breaking the build pipeline for millions of applications that relied on this one-page utility. In response, npm no longer allows packages to be removed from the registry after a certain amount of time has passed since they were published.

In 2018, the credentials of the owner of *eslint-scope* were compromised by a hacker who published a new version of *eslint-scope* that would steal local credentials on any machine it was installed on. This proved that npm libraries could be used as attack vectors for hackers. Since the incident, npm has increased documentation on security, but compromised package maintainer credentials are still a risk that, if exploited, could result in the loss of company source code, IP, or general malice as a result of malicious script downloads.

Later in 2018, a similar attack occurred with *event-stream*, which had added a dependency of *flatmap-stream*. *flatmap-stream* included some malicious code to steal the Bitcoin wallets of the computer it was installed on, hence stealing wallets from many users relying on *flatmap-stream* unknowingly.

As you can see, npm is ripe for exploitation in many ways and presents a significant security risk as it may be nearly impossible to evaluate each dependency and subdependency of a large application at a source-code level. Simply integrating your OSS npm package into a commercial appli-

cation could be an attack vector capable of resulting in a fully compromised company IP or worse.

I suggest that these package managers are a risk and provide examples only so that such risks can be properly mitigated. I also suggest that if you attempt to use npm libraries to exploit a business, you do so only with explicit written permission from the owners and on the basis of a red-team-style testing scenario only.

Java

Java uses a wide host of package managers, such as Ant and Gradle, with the most popular being Maven, supported by the Apache Software Foundation (see [Figure 17-4](#)). Maven operates similarly to JavaScript's npm—it is a package manager and is usually integrated in the build pipeline.

Because Maven predates Git version control, much more of its dependency management code is written from the ground up rather than relying on what is provided via Git. As a result, the underlying implementation between npm and Maven is different, although the function of the two is quite similar.

Welcome

[License](#)

ABOUT MAVEN

[What is Maven?](#)

[Features](#)

[Download](#)

[Use](#)

[Release Notes](#)

DOCUMENTATION

[Maven Plugins](#)

[Maven Extensions](#)

[Index \(category\)](#)

[User Centre](#)

[Plugin Developer](#)

[Centre](#)

[Maven Repository](#)

[Centre](#)

[Maven Developer](#)

[Centre](#)

[Books and Resources](#)

[Security](#)

COMMUNITY

[Community Overview](#)

[Project Roles](#)

[How to Contribute](#)

[Getting Help](#)

[Issue Management](#)

[Getting Maven Source](#)

[The Maven Team](#)

PROJECT

DOCUMENTATION

[Project Information](#)

MAVEN PROJECTS

[Maven](#)

[Archetypes](#)

[Extensions](#)

[Parent POMs](#)

[Plugins](#)

[Skins](#)

Welcome to Apache Maven

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

If you think that Maven could help your project, you can find out more information in the "About Maven" section of the navigation. This includes an in-depth description of [what Maven is](#) and a list of some of its main features.

This site is separated into the following sections, depending on how you'd like to use Maven:

Use	Download, Install, Configure, Run Maven	Maven Plugins and Maven Extensions
	Information for those needing to build a project that uses Maven	Lists of plugins and extensions to help with your builds.
Extend	Write Maven Plugins	Improve the Maven Central Repository
	Information for developers writing Maven plugins.	Information for those who may or may not use Maven, but are interested in getting project metadata into the central repository .
Contribute	Help Maven	Develop Maven
	Information if you'd like to get involved. Maven is an open source community and welcomes contributions.	Information for those who are currently Maven developers, or who are interested in contributing to the Maven project itself.

Each guide is divided into a number of trails to get you started on a particular topic, and includes a reference area and a "cookbook" of common examples.

You can access the guides at any time from the left navigation. If you are looking for a quick reference, you can use the [documentation index](#).

How to Get Support

Support for Maven is available in a variety of different forms.

To get started, search the documentation, [issue management system](#), the [wiki](#) or the [mailing list archives](#) to see if the problem has been solved or reported before.

If the problem has not been reported before, the recommended way to get help is to subscribe to the [Maven Users Mailing list](#). Many other users and Maven developers will answer your questions there, and the answer will be archived for others in the future.

You can also reach the Maven developers on [Slack](#).

Apache Software Foundation

Maven is a part of the Apache Software Foundation. We'd like to [thank the sponsors](#) that provide financial assistance to the foundation. For more information on how you can support the foundation, see the [sponsorship](#) page.



You can also attend [Apache Events](#). Don't hesitate to ask on the [Maven User mailing list](#) if Maven team members will be there. It can be a great opportunity to meet them.

Figure 17-4. Maven—the oldest and most popular package manager for Java-based applications

Maven, too, has been the target of attacks in the past, though typically these have received less media attention than npm. Just like npm, Maven projects and plug-ins can be compromised and imported into a legitimate application. Such risks are not isolated to any one package management software.

Other Languages

C#, C, C++, and most other large mainstream programming languages all have similar package managers to JavaScript or Java (NuGet, Conan, Spack, etc.). Each of these can be attacked with similar methods, either by the addition of a malicious package that is then incorporated into a legitimate application's codebase, or by the addition of a malicious dependency, which is then incorporated into a legitimate package and then incorporated into a legitimate application's codebase.

Attacking via a package manager may require a combination of social engineering and code obfuscation technique. Malicious code must be out of plain site, so that it is not easily identified, but still capable of execution.

Ultimately, package managers present a similar risk to any method of OSS integration. It is difficult to fully review the code in a large OSS package, especially when you take into consideration its dependencies.

Common Vulnerabilities and Exposures Database

Generally speaking, deploying a package to a package manager and getting it integrated into an application could be an attack vector, but it would require a significant amount of long-term effort and planning. The most popular way of exploiting third-party dependencies quickly is by determining known vulnerabilities in the application's dependencies that have not yet been patched and attacking those dependencies.

Fortunately, vulnerabilities are disclosed publicly when found in many packages. These vulnerabilities often make it to an online database like the US Department of Commerce National Vulnerability Database (NVD) (see [Figure 17-5](#)) or Mitre's [Common Vulnerabilities and Exposures \(CVE\) database](#), which is sponsored by the US Department of Homeland Security.

This means that popular third-party applications will likely have known and documented vulnerabilities as a result of many companies collaborating and contributing research from their own security analysis for others to read.

CVE databases are not incredibly useful when attempting to find known vulnerabilities in smaller packages, such as a GitHub repo with two contributors that has been downloaded three hundred times. On the other hand, major dependencies like WordPress, Bootstrap, or JQuery that have millions of users have often been scrutinized by many companies prior to being introduced in a production environment. As a result, the majority

of serious vulnerabilities have likely already been found, documented, and published on the web.

JQuery is a good example of this. As one of the top 10 most commonly used libraries in JavaScript, JQuery is used on over 10 million websites, has over 18,000 forks on GitHub in addition to over 250 contributors, and has around 7,000 commits comprising 150 releases.

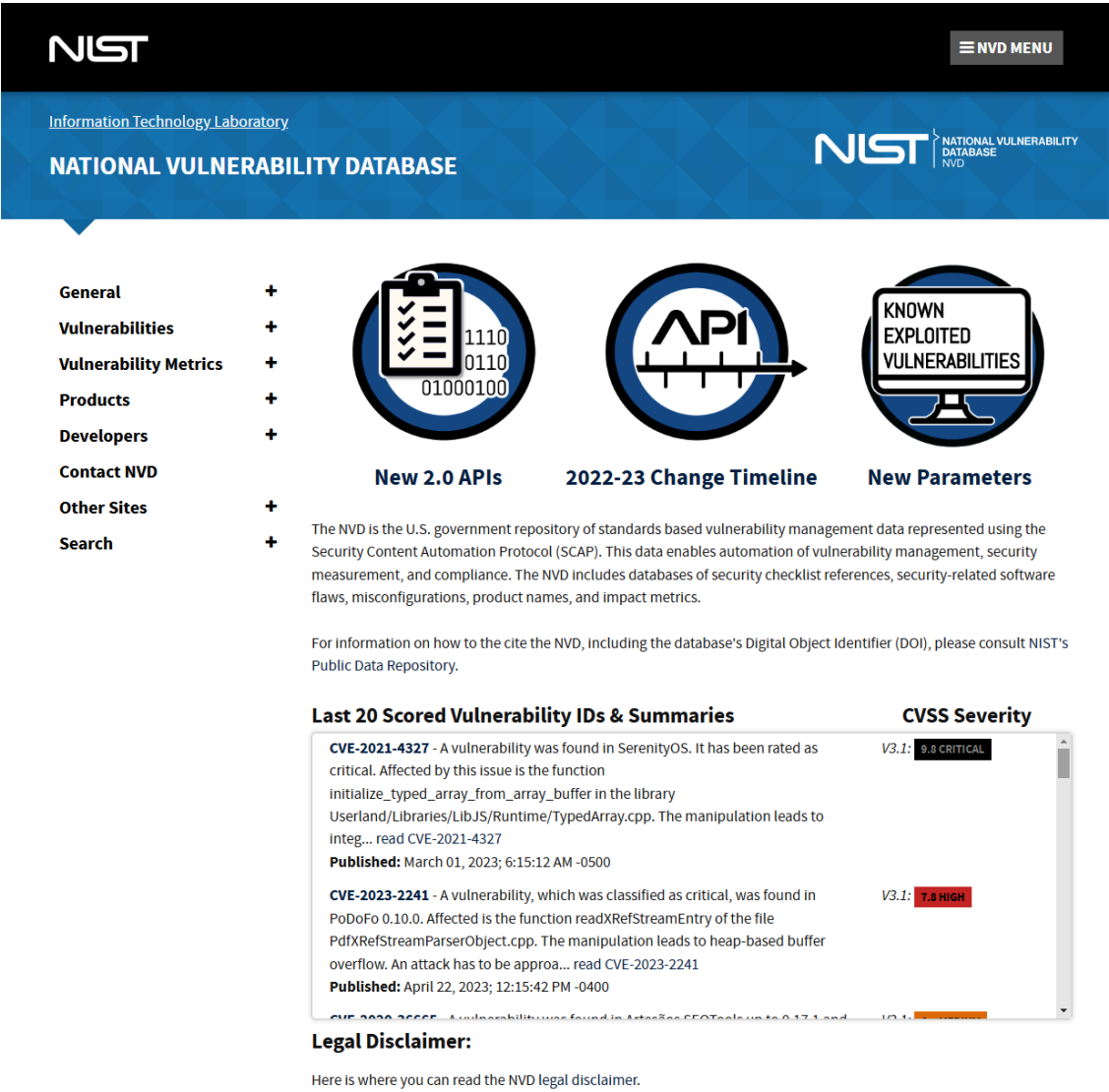


Figure 17-5. NVD, the national database of known vulnerabilities scored by severity

Due to its widespread usage and visibility, JQuery is constantly under high scrutiny for attention to secure coding and architecture. A serious vulnerability in JQuery could wreak havoc on some of the largest companies in the world—the damage would be widespread.

A quick scan of NVD's CVE database shows dozens of reported vulnerabilities in JQuery over the years. These include reproduction steps and

threat ratings to determine how easily exploitable the vulnerability is and what level of risk the vulnerability would bring to an organization.

As an attacker, these CVE databases can provide you with detailed methods of exploiting an application that contains a previously disclosed vulnerability. CVE databases make finding and exploiting vulnerabilities very easy, but you must still make use of reconnaissance techniques to properly identify dependencies, their integration with the primary application, and the versions and configurations used by those dependencies.

Summary

The rampant use of third-party dependencies, in particular from the OSS realm, has created an easy-to-overlook gap in the security of many web applications. A hacker, bug bounty hunter, or penetration tester can take advantage of these integrations and jump-start their search for live vulnerabilities. Third-party dependencies can be attacked a number of ways, from shoddy integrations to fourth-party code or just by finding known exploits discovered by other researchers or companies.

While the topic of third-party dependencies as an offensive attack vector is wide, and difficult to narrowly profile, these dependencies should always be considered in any type of offensive-style testing environment. Third-party dependencies can take a bit of reconnaissance effort to fully understand their role in a complex web application, but once that reconnaissance effort is complete, vulnerabilities in the dependencies often become visible quicker than those in first-party code. This is because these dependencies lack the same rigid review and assurance processes as first-party code, making them a great starting point for any type of web application exploitation.