Chapter 35. Securing Third-Party Dependencies

In Chapter 6, we investigated ways of identifying third-party dependencies in a first-party web application. In Chapter 17, we analyzed various ways that third-party dependencies are integrated in a first-party web application. Based on the integration, we were able to identify potential attack vectors and discuss ways of exploiting such integrations.

Because Part III is all about defensive techniques to stifle hackers, this chapter is all about protecting your application from vulnerabilities that could arise when integrating with third-party dependencies.

Evaluating Dependency Trees

One of the most important things to keep in mind when considering third-party dependencies is that many of them have their own dependencies. Sometimes these are called *fourth-party* dependencies.

Manually evaluating a single third-party dependency that lacks fourth-party dependencies is doable. Manual code-level evaluation of third-party dependencies is ideal in many cases.

Unfortunately, manual code reviews don't scale particularly well. In many cases it would be impossible to comprehensively review a third-party dependency that relied on fourth-party dependencies, especially if those fourth-party dependencies contain their own dependencies, and so on.

Third-party dependencies, their dependencies, the dependencies of those dependencies, etc., make up what is known as a dependency tree (see

Figure 35-1). Using the `npm ls` command in an npm-powered project, you can list an entire dependency tree out for evaluation. This command is powerful for seeing how many dependencies your application actually has because you may not consider the subdependencies on a regular basis.

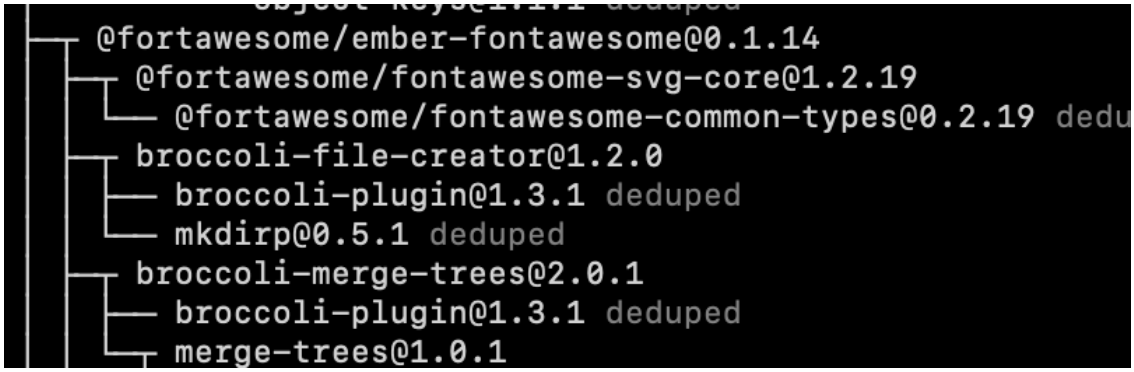

Figure 35-1. An npm dependency tree

Dependency trees are important in software engineering because they allow evaluation of an overarching application's code, which can result in dramatic file and memory size reduction.

Modeling a Dependency Tree

Consider an application with a dependency tree like this:

> *Primary Application → JQuery*
>
> *Primary Application → SPA Framework → JQuery*
>
> *Primary Application → UI Component Library → JQuery*

Being able to model a dependency tree would allow the application to identify that three parts of the dependency chain rely on JQuery. As a result, JQuery can be imported once and used in many places rather than imported three times (resulting in redundant file and memory storage).

Modeling dependency trees is also important in security engineering. This is because without proper dependency tree modeling, evaluating each dependency of the first-party application is quite hard.

In an ideal world, each component in an application that relied on JQuery to function (like the preceding example) would rely on the same version of JQuery. But in the real world, that is rarely the case. First-party applications can standardize on dependency versions, but it is unlikely the first-party application will standardize with the remainder of the dependency chain. This is because each item in the dependency chain may rely on functionality or implementation details that differ from version to version. The philosophy behind when and how to upgrade dependencies also differs from organization to organization.

Dependency Trees in the Real World

A real-world dependency tree often looks like the following: Primary Application v1.6 → JQuery 3.4.0 Primary Application v1.6 → SPA Framework v1.3.2 → JQuery v2.2.1 Primary Application v1.6 → UI Component Library v4.5.0 → JQuery v2.2.1

It is very much possible that version 2.2.1 of a dependency has critical vulnerabilities, while version 3.4.0 does not. As a result, each unique dependency should be evaluated in addition to each unique version of each unique dependency. In a large application with a hundred third-party dependencies, this can result in a dependency tree spanning thousands or even tens of thousands of unique subdependencies and dependency versions.

Automated Evaluation

Obviously, a large application with a ten thousand-item-long dependency chain would be nearly impossible to properly evaluate manually. As a result, dependency trees must be evaluated using automated means, and other techniques should be used in addition to ensure the integrity of the dependencies being relied on.

If a dependency tree can be pulled into memory and modeled using a tree-like data structure, iteration through the dependency tree becomes quite simple and surprisingly efficient. Upon addition to the first-party application, any dependency and all of its subdependency trees should be

evaluated. The evaluation of these trees should be performed in an automated fashion.

The easiest way to begin finding vulnerabilities in a dependency tree is to compare your application's dependency tree against a well-known CVE database. These databases host lists and reproductions of vulnerabilities found in well-known OSS packages and third-party packages that are often integrated in first-party applications.

You can download a third-party scanner (like Snyk) or write a bit of script to convert your dependency tree into a list and then compare it against a remote CVE database. In the npm world, you can begin this process with a command like: `npm list --depth=[depth]`.

You can compare your findings against a number of databases, but for longevity's sake, you may want to start with the NIST; it is funded by the US government and likely to stick around for a long time.

Secure Integration Techniques

In "Methods of Integration", we evaluated different integration techniques, discussing the pros and cons of each from the perspective of an onlooker or an attacker.

Let's imagine we are now viewing an integration from the perspective of an application owner. What are the most secure ways of integrating a third-party dependency?

Separation of Concerns

Unfortunately, one risk of integrating with third-party code on your main application server is that that code may have side effects or (if compromised) be able to take over system resources and functionality if the principle of least authority is not correctly implemented. One way to mitigate this risk is to run the third-party integration on its own server (ideally maintained by your organization).

After setting up the integration on its own server, have your server communicate with it via HTTP—sending and receiving JSON payloads. The JSON format ensures that script execution on the application server is not possible without additional vulnerabilities (*vulnerability chaining*) and allows for the dependency to be considered more like a "pure function" as long as you do not persist state on the dependency server.

Note that while this reduces the risk on the primary application server, any confidential data sent to the dependency server could still be modified and potentially recorded (with an improperly configured firewall) if the package is compromised. Additionally, this technique will implement a reduction in application performance due to increased in-transit time for functions to return data.

But the concepts behind this can be employed elsewhere; for example, a single server could employ a number of modules with hardware-defined process and memory boundaries. In doing this, a "risky" package would struggle to get the resources and functionality of the main application.

Secure Package Management

When dealing with package management systems like npm or Maven, there is a certain amount of accepted risk that comes with each individual system and the boundaries and review required for published applications. One way of mitigating risk from third-party packages installed this way is to individually audit specific versions of the dependency, then "lock" the semantic version to the audited version number.

Semantic versioning uses three numbers: a "major" release, a "minor" release, and a "patch." Generally speaking, most package managers attempt to automatically keep your dependencies on the latest patch by default. This means that, for example, myLib 1.0.23 could be upgraded to myLib 1.0.24 without your knowledge.

npm will include a caret (`^`) prior to any dependency by default. If you remove this caret, the dependency will use the exact version rather than the latest patch (1.0.24 versus ^1.0.24).

This technique, unknown to most, does not protect your application if the dependency maintainer deploys a new version using an existing version number. Honoring the rule of new code → new version number is entirely up to the dependency maintainer in npm and several other package managers. Furthermore, this technique only forces the top-level dependency to maintain a strict version and does not apply to descendant dependencies.

This is where *shrinkwrapping* comes into play. Running the command `npm shrinkwrap` against an npm repo will generate a new file called *npm-shrinkwrap.json*. From this point forward, the current version of each dependency and subdependency (the dependency tree) will be used at the exact version level.

This eliminates the risk of a dependency updating to the latest patch and pulling in vulnerable code. It does not, however, eliminate the very rare risk of a package maintainer reusing a version number for its dependency. To eliminate this risk, modify your shrinkwrap file to reference Git SHAs or deploy your own npm mirror that contains the correct versions of each dependency.

Summary

Today's web applications often have thousands, if not more, of individual dependencies required for application functionality to operate as normal. Ensuring the security of each script in each dependency is a massive undertaking. As such, it should be assumed that any third-party integration comes with at least some amount of expected risk (in exchange for reduced development time). However, while this risk cannot be eliminated, it can be mitigated in a number of ways.

Applying the principle of least privilege, we can let specific dependencies run on their own server, or at least in their own environment with isolated server resources. This technique reduces the risk to the rest of your application in the case of a severe security bug being found or a malicious

script going unnoticed. For some dependencies, however, isolation is difficult or impossible.

Dependencies that very tightly integrate with your core web application should be evaluated independently at a particular version number. If these dependencies are brought in via a package manager like npm, they should be version-locked and shrinkwrapped. For additional security, consider either referencing Git SHAs or deploying your own npm mirror. The same techniques for dealing with npm apply to other, similar package managers used in other languages.

To conclude, third-party dependencies always present risk, but careful integration with some thought behind it can mitigate a lot of the upfront risk your application would otherwise be exposed to.