Chapter 6. Identifying Third-Party Dependencies

Most web applications today are built on a combination of in-house code and external code integrated internally by one of many integration techniques. External dependencies can be proprietary from another company, which allows integration under a certain licensing model, or free—often from the OSS community. The use of such third-party dependencies in application code is not risk free, and often third-party dependencies are not subject to as robust a security review as in-house code.

During reconnaissance you will likely encounter many third-party integrations, and you will want to pay a lot of attention to both the dependency and the method of integration. Often these dependencies can turn into attack vectors; sometimes vulnerabilities in such dependencies are well known and you may not even have to prepare an attack yourself but will instead be able to copy an attack from a *Common Vulnerabilities and Exposures* (CVE) database.

Detecting Client-Side Frameworks

Often, rather than building out complex UI infrastructure, developers take advantage of well-maintained and well-tested UI frameworks. These often come in the form of SPA libraries for handling complex state, JS-only frameworks for patching functionality holes in the JavaScript language across browsers (Lodash, JQuery), or as CSS frameworks for improving the look and feel of a website (Bootstrap, Bulma).

Usually all three of these frameworks are easy to detect. If you can pin down the version number, you can often find a combination of applicable

ReDoS, prototype pollution, and XSS vulnerabilities on the web (in particular with older versions that have not been updated).

Detecting SPA Frameworks

The largest SPA frameworks on the web as of 2023 are (in no particular order):

- EmberJS (LinkedIn, Netflix)
- Angular (Google)
- React (Facebook)
- VueJS (Adobe, GitLab)

Each of these frameworks introduces very particular syntax and order as to how they manage DOM elements and how a developer interacts with the framework. Not all frameworks are this easy to detect. Some require fingerprinting or advanced techniques. When the version is given to you, always make sure to write it down.

EmberJS

EmberJS is quite easy to detect because when EmberJS bootstraps, it sets up a global variable `Ember` that can easily be found in the browser console (see Figure 6-1).

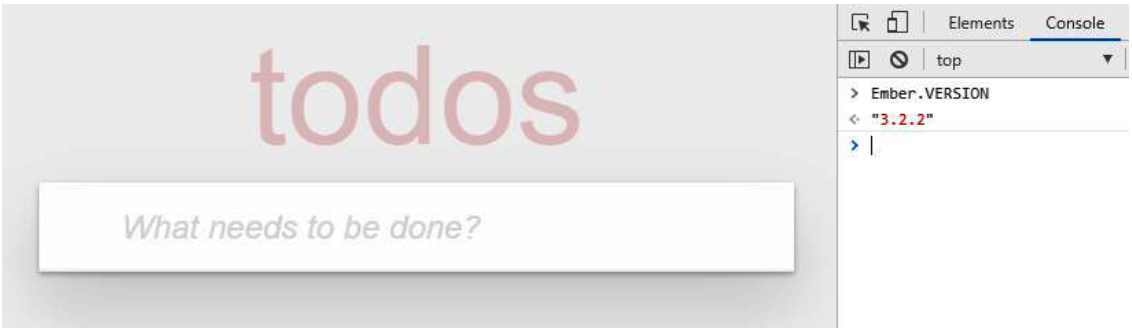

Figure 6-1. Detecting the EmberJS version

EmberJS also tags all DOM elements with an `ember-id` for its own internal use. This means that if you look at the DOM tree in any given web page using EmberJS via the Developer tools → Elements tab, you should see a number of divs containing `id=ember1, id=ember2, id=ember3`, etc.

Each of these divs should be wrapped in a `class="ember-application"` parent element, which is usually the `body` element.

EmberJS makes it easy to detect the version running. Simply reference a constant attached to the global `Ember` object:

```
// 3.1.0
console.log(Ember.VERSION);
```

Angular

Older versions of Angular provide a global object similar to EmberJS. The global object is named `angular`, and the version can be derived from its property `angular.version`. Angular 4.0+ got rid of this global object, which makes it a bit harder to determine the version of an Angular app. You can detect if an application is running Angular 4.0+ by checking to see if the `ng` global exists in the console.

To detect the version, you need to put in a bit more work. First, grab all of the root elements in the Angular app. Then check the attributes on the first root element. The first root element should have an attribute `ng-version` that will supply you the Angular version of the app you are investigating:

```
// returns array of root elements
const elements = getAllAngularRootElements();
const version = elements[0].attributes['ng-version'];

// ng-version="6.1.2"
console.log(version);
```

React

React can be identified by the global object `React`, and like EmberJS, can have its version detected easily via a constant:

```
const version = React.version;

// 0.13.3
console.log(version);
```

You may also notice script tags with the type `text/jsx` referencing React's special file format that contains JavaScript, CSS, and HTML all in the same file. This is a dead giveaway that you are working with a React app, and knowing that every part of a component originates from a single *.jsx* file can make investigating individual components much easier.

VueJS

Similarly to React and EmberJS, VueJS exposes a global object `Vue` with a version constant:

```
const version = Vue.version;

// 2.6.10
console.log(version);
```

If you cannot inspect elements on a VueJS app, it is likely because the app was configured to ignore developer tools. This is a toggled property attached to the global object `Vue`. You can flip this property to `true` in order to begin inspecting VueJS components in the browser console again:

```
// Vue components can now be inspected
Vue.config.devtools = true;
```

Detecting JavaScript Libraries

There are too many JavaScript helper libraries to count, and some expose globals while others operate under the radar. Many JavaScript libraries use the top-level global objects for namespacing their functions. These libraries are very easy to detect and iterate through (see Figure 6-2).

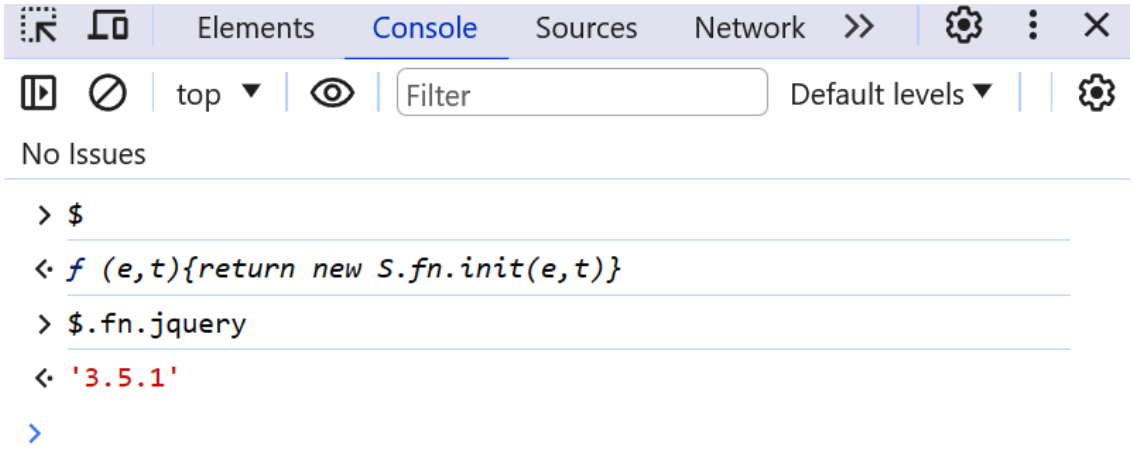

Figure 6-2. JavaScript library globals

Underscore and Lodash expose globals using the underscore symbol (`_`), and JQuery makes use of the `$` namespace. However, beyond the major libraries, you are better off running a query to see all of the external scripts loaded into the page.

We can make use of the DOM's `querySelectorAll` function to rapidly find a list of all third-party scripts imported into the page:

```
/*
 * Makes use of built-in DOM traversal function
 * to quickly generate a list of each <script>
 * tag imported into the current page.
 */
const getScripts = function() {

  /*
   * A query selector can either start with a "."
   * if referencing a CSS class, a "#" if referencing
   * an `id` attribute, or with no prefix if referencing an HTML element.
   *
   * In this case, 'script' will find all instances of <script>.
   */
  const scripts = document.querySelectorAll('script');

  /*
   * Iterate through each `<script>` element, and check if the element
   * contains a source (src) attribute that is not empty.
   */
  scripts.forEach((script) => {
    if (script.src) {
```

```
        console.log(`i: ${script.src}`);
      }
    });
};
```

Calling this function will give us output like this:

```
getScripts();

VM183:5 i: https://www.google-analytics.com/analytics.js
VM183:5 i: https://www.googletagmanager.com/gtag/js?id=UA-1234
VM183:5 i: https://js.stripe.com/v3/
VM183:5 i: https://code.jquery.com/jquery-3.4.1.min.js
VM183:5 i: https://cdnjs.cloudflare.com/ajax/libs/d3/5.9.7/d3.min.js
VM183:5 i: /assets/main.js
```

From here we need to directly access the scripts individually in order to determine orders, configurations, etc.

Detecting CSS Libraries

With minor modifications to the algorithm to detect scripts, we can also detect CSS:

```
/*
 * Makes use of DOM traversal built into the browser to
 * quickly aggregate every `<link>` element that includes
 * a `rel` attribute with the value `stylesheet`.
 */
const getStyles = function() {
  const scripts = document.querySelectorAll('link');

  /*
   * Iterate through each script, and confirm that the `link`
   * element contains a `rel` attribute with the value `stylesheet`.
   *
   * Link is a multipurpose element most commonly used for loading CSS
   * stylesheets, but also used for preloading, icons, or search.
   */
  scripts.forEach((link) => {
```

```
    if (link.rel === 'stylesheet') {
      console.log(`i: ${link.getAttribute('href')}`);
    }
  });
};
```

Again, this function will output a list of imported CSS files:

```
getStyles();

VM213:5 i: /assets/jquery-ui.css
VM213:5 i: /assets/boostrap.css
VM213:5 i: /assets/main.css
VM213:5 i: /assets/components.css
VM213:5 i: /assets/reset.css
```

Detecting Server-Side Frameworks

Detecting what software is running on the client (browser) is much easier than detecting what is running on the server. Most of the time, all of the code required for the client is downloaded and stored in memory referenced via the DOM. Some scripts may load conditionally or asynchronously after a page loads, but these can still be accessed as long as you trigger the correct conditions.

Detecting what dependencies a server has is much harder, but often not impossible. Sometimes server-side dependencies leave a distinct mark on HTTP traffic (headers, optional fields) or expose their own endpoints. Detecting server-side frameworks requires more knowledge about the individual frameworks being used, but fortunately, just like on the client, there are a few packages that are very widely used. If you can memorize ways to detect the top packages, you will be able to recognize them on many web applications that you investigate.

Header Detection

Some insecurely configured web server packages expose too much data in their default headers. A prime example of this is the `X-Powered-By` header, which will literally give away the name and version of a web server. Often this is enabled by default on older versions of Microsoft IIS. Make any call to one of those vulnerable web servers and you should see a return value like this in the response:

```
X-Powered-By: ASP.NET
```

If you are very lucky, the web server might even provide additional information:

```
Server: Microsoft-IIS/4.5
X-AspNet-Version: 4.0.25
```

Smart server administrators disable these headers, and smart development teams remove them from the default configuration. But there are still millions of websites exposing these headers to be read by anyone.

Default Error Messages and 404 Pages

Some popular frameworks don't provide very easy methods of determining the version number used. If these frameworks are open source, like Ruby on Rails, then you may be able to determine the version used via fingerprinting. Ruby on Rails is one of the largest open source web application frameworks, and its source code is hosted on GitHub for easier collaboration. Not only is the most recent version available, but all historical versions using Git version control can be found. As a result, specific changes from commit to commit can be used to fingerprint the version of Ruby on Rails being used (see Figure 6-3).

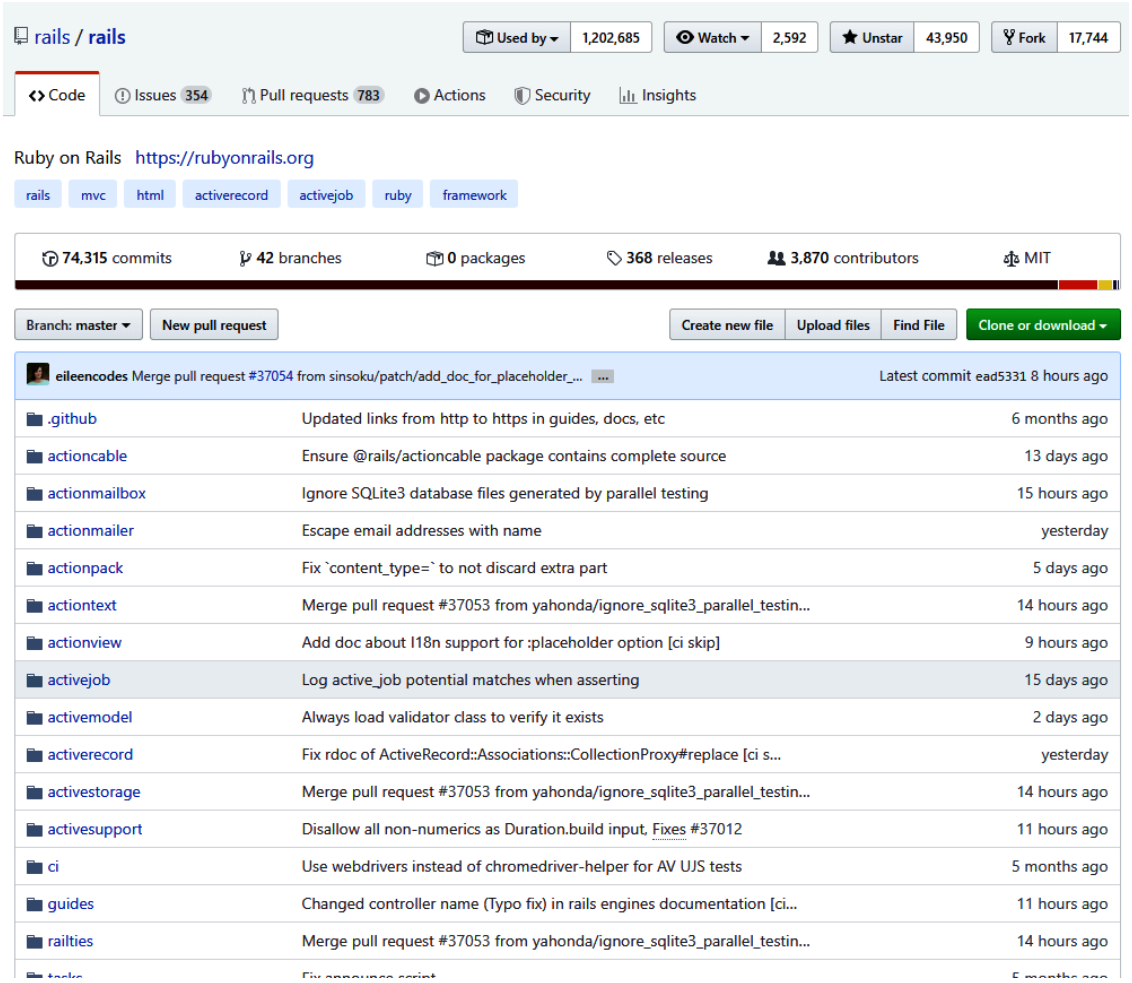

Figure 6-3. Fingerprinting the version of Ruby on Rails being used

Have you ever visited a web application and been presented with a standard 404 page or had an out-of-the-box error message pop up? Most web servers provide their own default error messages and 404 pages, which continue to be presented to users until they are replaced with a custom alternative by the owner of the web application.

These 404 pages and error messages can expose quite a bit of intelligence regarding your server setup. Not only can these expose your server software, but they can often expose the version or range of versions as well.

Take, for example, the full stack web application framework Ruby on Rails. It has its own default 404 page, which is an HTML page containing a box with the words “The page you were looking for doesn’t exist” (see Figure 6-4).

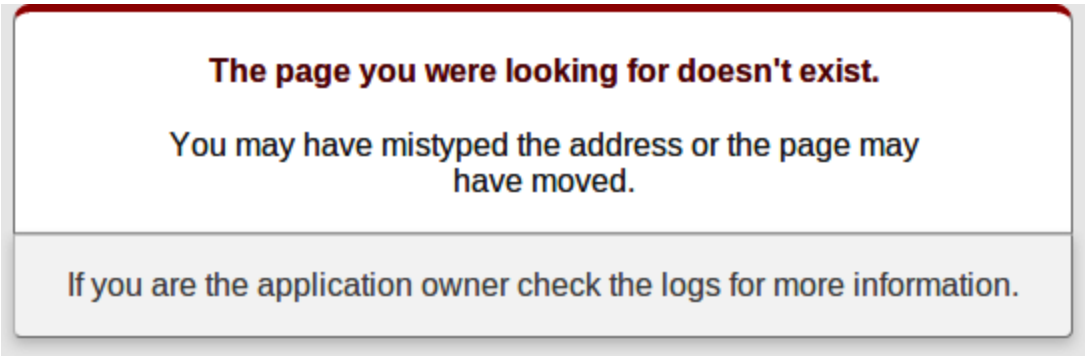

Figure 6-4. Ruby on Rails default 404 page

The HTML powering this page can be found at the public GitHub repository for Ruby on Rails under the file location *rails/railties/lib/rails/generators/rails/app/templates/public/404.html*. If you clone the Ruby on Rails repository on your local machine (using `git clone https://github.com/rails/rails`) and begin sifting through the changes to that page (using `git log | grep 404`), you may find some interesting tidbits of information, such as:

- April 20, 2017—Namespaced CSS selectors added to 404 page
- November 21, 2013—U+00A0 replaced with whitespace
- April 5, 2012—HTML5 type attribute removed

Now if you are testing an application and you stumble upon its 404 page, you can search for the HTML5 type attribute `type="text/css"`, which was removed in 2012. If this exists, you are on a version of Ruby on Rails shipped April 5, 2012, or earlier.

Next, you can look for the U+00A0 character. If that exists, then the application's version of Ruby on Rails is from November 21, 2013, or earlier.

Finally, you can search for the namespaced CSS selectors, `.rails-default-error-page`. If these do not exist, then you know the version of Ruby on Rails is from April 20, 2017, or earlier.

Let's assume you get lucky and the HTML5 type attribute was removed, and the U+00A0 was replaced with whitespace, but the namespaced CSS selectors are not yet in the 404 page you are testing. We can now cross-reference those time frames with the official release schedule listed on

the Ruby Gems package manager website. As a result of this cross-referencing, we can determine a version range.

From this cross-referencing exercise we can determine that the version of Ruby on Rails being tested is somewhere between version `3.2.16` and `4.2.8`. It just so happens that Ruby on Rails version `3.2.x` until `4.2.7` was subject to an XSS vulnerability, which is well documented on the internet and in vulnerability databases (CVE-2016-6316).

This attack allowed a hacker to inject HTML code padded with quotes into any database field read by an Action View Tag helper on the Ruby on Rails client. Script tags containing JavaScript code in this HTML would be executed on any device that visited the Ruby on Rails-based web application and interacted with it in a way to trigger the Action View helpers to run.

This is just one example of how investigating the dependencies and versions of a web application can lead to easy exploitation. We will cover this type of exploitation in the next part of the book, but keep in mind that these techniques don't just apply to Ruby on Rails. They apply to any third-party dependency where you (the hacker or tester) can determine the software and versions of that software that the application integrates with.

Database Detection

Most web applications use a server-side database (such as MySQL or MongoDB) to store state regarding users, objects, and other persistent data. Very few web application developers build their own databases, as efficiently storing and retrieving large amounts of data in a reliable way is not a small task.

If database error messages are sent to the client directly, a similar technique to the one for detecting server packages can be used to determine the database. Often this is not the case, so you must find an alternative discovery route.

One technique that can be used is primary key scanning. Most databases support the notion of a "primary key," which refers to a key in a table (SQL) or document (NoSQL) that is generated automatically upon object creation and used for rapidly performing lookups in the database. The method by which these keys are generated differs from database to database and can at times be configured by the developer if special needs are required (such as shorter keys for use in URLs). If you can determine how the default primary keys are generated for a few major databases, unless the default method has been overwritten, you will likely be able to determine the database type after sifting through enough network requests.

Take, for example, MongoDB, a popular NoSQL database. By default, MongoDB generates a field called `_id` for each document created. The `_id` key is generated using a low-collision hashing algorithm that always results in a hexadecimal-compatible string of length 12. Furthermore, the algorithm used by MongoDB is visible in its open source documentation.

The documentation tells us the following:

- The class that is used to generate these `id`s is known as `ObjectId`.
- Each `id` is exactly 12 bytes.
- The first 4 bytes represent the seconds since the Unix epoch (Unix timestamp).
- The next 5 bytes are random.
- The final 3 bytes are a counter beginning with a random value.

An example `ObjectId` would look like this: `507f1f77bcf86cd799439011`.

The `ObjectId` spec also goes on to list helper methods like `getTimestamp()`, but since we will be analyzing traffic and data on the client rather than the server, those helper methods likely will not be exposed to us. Instead, knowing the structure of MongoDB's primary keys, we want to look through HTTP traffic and analyze the payloads we find for 12-byte strings with a similar appearance. This is often simple, and you will find a primary key in the form of a request like:

```
GET users/:id
```

Where `:id` is a primary key

`PUT users, body = { id: id }`

Where `id` again is a primary key

`GET users?id=id`

Where the `id` is a primary key but in the query params

Sometimes the `id`s will appear in places you least expect them, such as in metadata or in a response regarding a user object:

```
{
  _id: '507f1f77bcf86cd799439011',
  username: 'joe123',
  email: 'joe123@my-email.com',
  role: 'moderator',
  biography: '...'
}
```

Regardless of how you find a primary key, if you can determine that the value is indeed a primary key from a database, then you can begin researching databases and trying to find a match with their key generation algorithms. Often this is enough to determine what database a web application is using, but from time to time you may need to use this in combination with another technique (e.g., forcing error messages) if you run into a case where multiple databases use the same primary key generation algorithm (e.g., sequential integers or other simple patterns).

Summary

For many years, first-party application code was the most common attack vector as far as source code goes. But that is changing today due to modern web application reliance on third-party and open source integrations.

Developing a deep understanding of a target's third-party integrations may lead you to security holes in an application that are ripe for exploita-

tion. Often these vulnerabilities are also difficult for the owner of an application to detect. Beyond this, understanding the way third-party dependencies are being used in your own codebase allows you to mitigate risk otherwise brought on by shoddy integration techniques or integration with less secure libraries (when more secure options are available).

In conclusion, due to the amount of code running underneath most of today's applications, third-party integration is almost mandatory. Building an entire full stack web application from scratch would be a heroic effort. As a result, understanding the techniques used to find and evaluate dependencies in an application is becoming a must-have skill for anyone involved in the security industry.