Thou Shalt *Not* Depend on Me

BY TOBIAS LAUINGER, ABDELBERI CHAABANE, AND CHRISTO B. WILSON

A LOOK AT JAVASCRIPT LIBRARIES IN THE WILD

This article is based on original research by Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda, first published in the Proceedings of the 2017 Network and Distributed System Security Symposium (Thou shalt not depend on me: analysing the use of outdated JavaScript libraries on the web; <https://seclab.ccs.neu.edu/static/publications/ndss2017jslibver.pdf>).

Many websites use third-party components such as JavaScript libraries, which bundle useful functionality so that developers can avoid reinventing the wheel. jQuery (<https://jquery.com/>) is arguably the most popular open-source JavaScript library at the moment—found on 84 percent of the most popular websites as determined by Amazon's Alexa (<https://www.alexa.com/topsites>). But what happens when libraries contain security issues? Chances are that websites using such libraries inherit these issues and become vulnerable to attacks.

Given the risk of using a library with known

vulnerabilities, it is important to know how often this happens in practice and, more importantly, who is to blame for the inclusion of vulnerable libraries—the developer of the website, or maybe a third-party advertisement or tracker code loaded on the website?

We set out to answer these questions and found that with 37 percent of websites using at least one known vulnerable library, and libraries often being included in quite unexpected ways, there clearly is room for improvement in library handling on the web. To that end, this article makes a few recommendations about what can be done to improve the situation.

JAVASCRIPT VULNERABILITIES

Before delving into how to detect the use of vulnerable libraries on the web, we need to agree on what constitutes a vulnerability. First, we are interested only in code that will run on the client side—that is, in a web browser. JavaScript is the de facto standard language for that purpose, and it has become notorious for security vulnerabilities such as XSS (cross-site scripting), which allows an attacker to inject malicious code (or HTML) into a website. In particular, if a JavaScript library accepts input from the user and does a poor job validating it, an XSS vulnerability might creep in, and all websites using this library could become vulnerable.

As an example, consider jQuery's `$( )` function. It has different behavior depending on which type of argument is passed: if the argument is a string containing a CSS (Cascading Style Sheets) selector, the function searches the DOM (Document Object Model) tree for

corresponding elements and returns references to them; if the input string contains HTML, the function *creates* the corresponding elements and returns the references. As a consequence, developers who pass improperly sanitized input to this function may inadvertently allow attackers to inject code into the page even though the programmer's intent is to select an existing element. While this API design places convenience over security considerations, and the implications could be better highlighted in the documentation, it does not automatically constitute a vulnerability in the library.

In older versions of jQuery, however, the `$()` function's leniency in parsing string parameters could lead to complications by misleading developers to believe, for example, that any string beginning with `#` would be interpreted as a selector and could be safe to pass to the function, as `#test` selects the element with identifier `test`. Yet, jQuery considered parameters containing an HTML `<tag>` *anywhere* in the string as HTML (<https://bugs.jquery.com/ticket/9521>), so that a parameter such as `#<img src=/ onerror=alert(1)>` would lead to code execution rather than a selection. This behavior was considered a vulnerability and fixed.

Other vulnerabilities in JavaScript libraries include cases where libraries fail to sanitize inputs that are expected to be pure text but are passed to `eval()` or `document.write()` internally, which could cause them to be executed as script or rendered as markup. Attackers could exploit these capabilities to steal data from a user's browsing session, initiate transactions on the user's

behalf, or place fake content on a website. Therefore, it is important that JavaScript libraries do not introduce any new attack vectors into the websites where they are used.

At the time of our research, there was no single “authoritative” public database of JavaScript vulnerabilities. We manually searched the OSVDB (Open Source Vulnerability Database), the NVD (National Vulnerability Database), public bug trackers, GitHub comments, blog posts, and the list of vulnerabilities detected by Retire.js (<https://retirejs.github.io/retire.js/>) to gather metadata about vulnerable and fixed versions for the 11 popular libraries shown in figure 1. As a result, given the name of one of these 11 libraries and a specific release version, we can say whether we know about any publicly disclosed vulnerability— but there are likely more vulnerabilities that we do not know about. Thus, what we report here should be seen as a lower bound.

FIGURE 1: **POPULAR LIBRARIES WITH KNOWN VULNERABILITIES**

LIBRARY DETECTION

Collecting vulnerability metadata manually was feasible because we restricted ourselves to 11 of the most popular libraries. For detection of libraries used on websites, however, an automated approach was needed. At first, detecting a library on a website does not sound too complicated: check how the library file is called in the official distribution, such as `jquery-3.2.1.js`, and look for that name in the URLs loaded by websites. Unfortunately, it's rarely that easy. Web developers can rename files, and they do. Using this simple strategy rather than the more complex detection methodology would miss 44 percent of all URLs containing the Modernizr library, for example. This is not acceptable.

Our approach uses a combination of static and dynamic methods. The static method is a slight improvement over the name-based approach: instead of detecting library files by their name, we detect them by the file hash. This required a comprehensive catalogue of library file hashes, compiled from download links found on the libraries' websites, and on JavaScript CDNs (content delivery networks) maintained by Google, Microsoft, and Yandex, as well as the community-based CDNs jsDelivr, cdnjs, and OSS CDN. Some libraries, such as Bootstrap and jQuery, maintain their own branded CDNs, which were included as well. All versions and variants of each library were downloaded. Variants typically included the "debug" version of the source code with comments, and a "minified" production version that had whitespace removed and internal identifiers shortened for smaller file size and faster page-load times.

A drawback of detecting a library by its hash is that

it cannot be detected when there is no corresponding reference file in the catalogue. This can happen, for example, when web developers modify the source code of the file. Source-code modifications such as addition or removal of comments, or custom minification, occur quite frequently in practice. Out of a random sample of scripts encountered in our crawls that were known to contain jQuery, only 15 percent could be detected based on the file hash. Therefore, we complemented the static detection with a dynamic detection method.

Dynamic detection examines the runtime environment when the library is loaded in a web browser. Many libraries register as a window-global variable and make available an attribute that contains the version number of the library. On a website using jQuery, for example, typing `$.fn.jquery` into the developer console of the browser returns a version number such as 3.2.1. Only detections returning a standard three-component `major.minor.patch` version number as used in semantic versioning (<http://semver.org/>) are counted. By convention, the major version component is increased for breaking changes, the minor component for new functionality, and the patch component for backwards-compatible bug fixes. Discarding detections with invalid or empty version attributes reduces the number of false-positive detections—that is, detections that do not actually correspond to the use of a library.

Furthermore, for the purposes of our data analysis, the version number of each detected library instance is needed to look up whether any vulnerabilities are known. Unfortunately, some libraries do not programmatically export version attributes, some libraries added this

feature only in more recent versions, and some library loading techniques such as Browserify or Webpack may prevent the library from registering its window-global variable. Furthermore, since only one instance of a window-global variable can exist at any time, when a library is loaded multiple times in the same page, only the last instance is visible at runtime. All these cases result in false-negative detections—that is, the dynamic-detection signature does not detect the library, even though it is present in a website.

Combining the static and dynamic detection methods overcomes their respective limitations. Our research paper also describes an offline variant of dynamic detection, used for the corner case of duplicate library inclusions.

CAUSALITY TREES

An important aspect of our research was finding out who is to blame for the inclusion of vulnerable libraries. To that end, we needed to model causal resource inclusion relationships in websites in order to represent how a library was included in a page. For example, a library may be referenced directly in a web page, or it can be included transitively when another referenced script loads additional resources. We call this model *causality trees*.

A causality tree contains a directed edge $A \rightarrow B$ if and only if element A causes element B to load. The elements modeled for this study are scripts and embedded HTML documents. A relationship exists whenever an element creates another element or changes an existing element's URL. Examples include a script creating an iframe, and a script changing the URL of an iframe.

While the nodes in a causality tree correspond to nodes in the website's DOM, their structure is entirely unrelated to the hierarchical DOM tree. Rather, nodes in the causality tree are snapshots of elements in the DOM tree *at a specific point in time* and may appear multiple times if the DOM elements are repeatedly modified. For example, if a script creates an iframe with URL U_1 and later changes the URL to U_2 , the corresponding script node in the causality tree will have two document nodes as its children, corresponding to URLs U_1 and U_2 but referring to the same HTML `<iframe>` element. Similarly, the predecessor of a node in the causality tree is not necessarily a predecessor of the corresponding HTML element in the DOM tree; they may even be located in two different HTML documents, such as when a script appends an element to a document in a different frame.

Figure 2 shows a synthetic example of a causality tree. The large white circle is the document root (main document), filled circles are scripts, and squares are HTML documents (e.g., embedded in frames). Edges denote

FIGURE 2: **GENERIC EXAMPLE OF A CAUSALITY TREE**

“created by” relationships; for example, in figure 2 the main document includes the gray script, which in turn includes the blue script. Dashed lines around nodes denote inline scripts, while solid lines denote scripts included from an URL. Thick outlines denote that a resource was included from a known ad network, tracker, or social widget.

The color of nodes in figure 2 denotes which document they are attached to in the DOM: gray corresponds to resources attached to the main document, while one of four colors is assigned to each further document in frames. Document squares contain the color of their parent location in the DOM, and their own assigned color. Resources created by a script in one frame can be attached to a document in another frame, as shown by the gray script that has a blue child in figure 2 (i.e., the blue script is a child of the blue document in the DOM).

Figure 3a shows a LinkedIn widget as included in the causality tree of [mercantil.com](https://seclab.ccs.neu.edu/static/projects/javascript-libraries/). (An interactive version is available online at <https://seclab.ccs.neu.edu/static/projects/javascript-libraries/>.) Note that the web developer embedded code provided by the social network into the main document, which in turn initializes the widget and creates several scripts in multiple frames.

WEB CRAWL

Causality trees are generated using an instrumented version of the Chromium web browser. Its Chrome DevTools Protocol (<https://chromedevtools.github.io/devtools-protocol/>) allows detection of most resource-inclusion relationships; for some corner cases, we had to resort to source code modifications in the browser. We

FIGURE 3: CAUSALITY TREE OF MERCANTIL.COM

also link library detections to nodes in the causality tree and run a modified version of Adblock Plus to label (but not block) advertisement, tracking, and social media nodes in the causality trees. While visiting a page, the crawler scrolls downward to trigger loading of any dynamic content. As page-loaded events proved to be unreliable, our crawler remains on each page for a fixed delay of 60 seconds before clearing its entire state, restarting, and then proceeding to the next site.

To gain a representative view of JavaScript library usage on the web, we collected two different data sets. First, we crawled Alexa's top 75,000 domains, which represent popular websites. Second, we crawled 75,000 domains randomly sampled from a snapshot of the .com zone—that is, a random sample of all websites with a .com

address, which was expected to be dominated by less popular websites. The two crawls, conducted in May 2016, successfully generated causality trees for the homepages of 71,217 domains in Alexa and 62,086 domains in .COM. Failures resulted from timeouts and unresolvable domains, which were expected especially for .COM, since the zone file contains domains that may not have an active website.

HOW WEBSITES USE LIBRARIES...

Overall, our study used static and dynamic signatures for 72 open-source libraries. We found at least one library on the homepage of 87 percent of the Alexa sites and 65 percent of the .COM sites. Figure 4 shows the 12 most common libraries in Alexa. jQuery is by far the most popular, used by 84 percent of the Alexa sites and 61

FIGURE 4: TOP 12 LIBRARIES BY FREQUENCY IN ALEXA

	ALEXA	COM
jQuery	84%	61%
jQuery-UI	24%	8%
Modernizr	21%	9%
Bootstrap	13%	5%
jQuery-Migrate	11%	11%
Underscore	6%	2%
SWFObject	4%	2%
Moment	4%	1%
RequireJS	3%	2%
jQuery-Form	3%	3%
Backbone	3%	2%
Angular	2%	2%

percent of the .COM sites. In other words, nearly every website that's using a library is using jQuery. SWFObject, a library used to include Adobe Flash content, is ranked seventh (4 percent) and tenth (2 percent), despite being discontinued since 2013. On the other hand, several relatively well-known libraries such as D3, Dojo, and Leaflet appear below the top 30 in both crawls, possibly because they are less commonly used on the homepages of websites.

While the majority of libraries used in Alexa are hosted on the same domain as the website, most inclusions are loaded from external domains in .COM. In the case of jQuery, 59 percent of all inclusions in Alexa websites are internal, and 39 percent are external. The remainder are inline inclusions where the source code of the library is not loaded from a file but directly wrapped in `<script>`
`// library code here </script>` tags. Only 30 percent of the websites in the .COM crawl host jQuery internally, whereas 68 percent rely on external hosting. This highlights a difference in how larger and smaller websites include libraries.

In both crawls, JavaScript CDNs are among the most popular domains from which libraries are loaded. In Alexa, almost 18 percent of library files are loaded from `ajax.googleapis.com`, Google's JavaScript CDN (13 percent in .COM), followed by jQuery's branded CDN `code.jquery.com` (4 percent in Alexa, 3 percent in .COM). The less popular sites in the .COM crawl, however, also frequently load libraries from domains related to domain parking and hosting providers.

When looking at *why* libraries are included, it turns out that around 3 percent of jQuery inclusions in Alexa and almost 26 percent in .COM are caused by advertisement, tracking, or social media widget code. For SWFObject, more than 42 percent of inclusions in Alexa come from ads. In other words, the blame for including a now-unsupported library does not go directly to those websites but to the ad networks they are using. Advertisement, tracking, or social media widget code is typically provided by an external service and loaded *as is* by the website developer—who may not be aware that the included code will load additional libraries and who has no say in which versions of these libraries will be loaded. Overall, libraries loaded by ads can be found on 7 percent of sites in Alexa, and on 16 percent of sites in .COM.

...AND HOW THEY INCLUDE *VULNERABILITIES*

We compiled metadata about vulnerable versions of the 11 libraries shown in figure 1. Among the Alexa sites, 38 percent use at least one of these 11 libraries in a version known to be vulnerable, and 10 percent use two or more different known vulnerable versions. In .COM, the vulnerability rates are slightly lower—37 percent of sites have at least one known vulnerable library, and 4 percent have two or more—but the sites in .COM also have a lower rate of library use in general. As a result, those .COM sites that *do* use a library have a higher probability of vulnerability than those in Alexa.

Looking at individual libraries shows that known vulnerable versions can make up a majority of all uses of those libraries in the wild. jQuery, for example, has around

37 percent known vulnerable inclusions in Alexa, and 55 percent in .COM. Angular has 39-40 percent vulnerable inclusions in both crawls, and Handlebars has 87-88 percent. This does not mean, however, that Handlebars is “more vulnerable” than jQuery; it means only that web developers use known vulnerable versions more often in the case of Handlebars than for jQuery. The emphasis here is on *known vulnerable*, as each library may contain vulnerabilities that are *not* known. In that sense, these results are a lower bound on the use of vulnerable libraries.

So far, we have examined whether sites are potentially vulnerable—that is, whether they include one or more known vulnerable libraries—and how that adds up on a per-library level. Now let’s return to our analysis of *how* libraries are included by sites. Figure 5 shows two prominent factors that are connected to a higher fraction of vulnerable inclusions:

➔ Inline inclusions of jQuery have a clearly higher fraction of vulnerable versions than internally or externally hosted copies.

FIGURE 5: **VULNERABLE FRACTION OF JQUERY INCLUSIONS**

	ALEXA	COM
All Inclusions	37%	55%
Internal	38%	42%
External	35%	63%
Inline	55%	90%
Ad/Widget/Tracker	38%	89%
No Ad/Widget/Tracker	37%	46%

➔ Library inclusions by ad, widget, or tracker code appear to be more vulnerable than unrelated inclusions. While the difference is relatively small for jQuery in Alexa, the vulnerability rate of jQuery associated with ad, widget, or tracker code in .COM—89 percent—is almost double the rate of unrelated inclusions. This may be a result of less reputable ad networks or widgets being used on the smaller sites in .COM as opposed to the larger sites in Alexa.

At this point, a word about the limitations of our study is in order. We do not check whether a known vulnerability in a library can be exploited when used on a specific website. If web developers can ensure that a library vulnerability cannot be exploited on their site, they do not need to update to a newer version. Yet, as will be discussed in a moment, the release notes of libraries rarely contain enough information that would allow a non-expert to decide whether continuing to use a vulnerable library on a specific site is safe or not. Therefore, in practice, the safe course of action would be always to update when a vulnerability in a library is discovered.

Unfortunately, because of the release cycles and patching behavior of library maintainers, updating a library dependency is easier said than done. Only a very small fraction of sites using vulnerable libraries (less than 3 percent in Alexa, and 2 percent in .COM) could become free of vulnerabilities by applying only patch-level updates. Updates of the least significant version component, such as from 1.2.3 to 1.2.4, would generally be expected to be backwards compatible. In most cases, however, patch updates are not available. The vast majority of sites would

need to install at least one library with a more recent major or minor version to remove all vulnerabilities. Migrating to these newer versions might necessitate additional code changes and site testing because of incompatibilities in the API.

Beyond vulnerabilities and considering all 72 supported libraries, 61 percent of Alexa sites and 46 percent of .COM sites are at least one patch version behind on one of their included libraries. Even though such updates should be “painless,” they are often neglected. Similarly, the median Alexa site uses a version released 1,177 days (1,476 days for .COM) before the newest available release of the library. These results demonstrate that the majority of web developers are working with library versions released a long time ago. Time differences measured in years suggest that web developers rarely update their library dependencies once they have deployed a site.

Analyzing the use of JavaScript libraries on websites reveals that libraries are often used in unexpected ways. For example, about 21 percent of the websites including jQuery in Alexa, and 17 percent in .COM, do so two or more times in a single web page. That alone is no cause for concern; when a website contains `<iframe>`s with documents loaded from different origins, it may even be necessary to include the library multiple times because of the same-origin policy limiting scripts' access across origins. Yet, a closer look reveals that 4 percent of websites using jQuery in Alexa include the same version of the library two or more times in the *same* document (5 percent in .COM), and 11 percent (6 percent) include two or more versions of jQuery in the same document. No

benefit is derived by including the library multiple times in the same document because jQuery registers itself as a window-global variable. Unless special steps are taken, only the last loaded and executed instance in each document can be used by client code; the other instances will be hidden. Asynchronously included instances may even create a race condition, making it difficult to predict which version will prevail in the end.

As an illustration, consider the detail from the causality tree for mercantil.com in figure 3b. The site includes jQuery four times. All these inclusions are referenced directly in the main page's source code, some of them directly adjacent to each other. On other sites, duplicate inclusions were caused by multiple scripts transitively including their own copies of jQuery. While we can only speculate on why these cases occur, at least some of them may be related to server-side templating, or the combination of independently developed components into a single document. Indeed, we have observed cases where a web application (e.g., a WordPress plug-in) that bundled its own version of a library was integrated into a page that already contained a separate copy of the same library. Since duplicate inclusions of a library do not necessarily break any functionality, many web developers may not be aware that they are including a library multiple times, and even fewer may be aware that the duplicate inclusion may be potentially vulnerable.

WHAT CAN, AND *SHOULD*, BE DONE?

Our research has shown that vulnerable libraries are widely used on the web. A number of factors are at play,

and no single actor can be made responsible for the situation. Instead, let's look at it from three different angles.

Dependency Management

Website developers need to be aware of which libraries they are using. It is too easy to forget about a library when it is manually copied into the codebase. Instead, we recommend explicitly declaring a project's dependencies in a central location. For client-side JavaScript, Bower (<https://bower.io/>) was one of the first dependency management tools. Yarn (<https://yarnpkg.com/>) is a more recent entry to the scene, backed by the repository of NPM (Node Package Manager; <https://www.npmjs.com/>), which contains not only server-side Node.js packages, but also client-side JavaScript libraries). Explicit dependencies make it easy to automatically include the library code of the declared version into the project. Additionally, tools such as Retire.js (<https://retirejs.github.io/retire.js/>), AuditJS (<https://github.com/OSSIndex/auditjs>), or Snyk (<https://snyk.io/>) can scan the declared dependencies for known vulnerable versions. Ideally, web developers should make such tools part of their build process, so that attempts to include a known vulnerable library cause a build to fail. For projects where such a proactive approach is not an option, Retire.js also has a browser extension that can detect vulnerable libraries in deployed websites.

Library Development

The development practices adopted by library maintainers have a big influence on how difficult it will be for library

users to keep their dependencies up to date. To that end, we conducted an informal survey of the 12 most frequently used libraries (figure 4).

Before developers can update the libraries they are using, they must be made aware that there is a need to update. None of these 12 libraries, however, seems to maintain a mailing list or other dedicated channel for security announcements. Some libraries have Twitter accounts, but these contain a lot of additional “noise” unrelated to new releases or security issues. None of the libraries appears to systematically allocate CVE (Common Vulnerabilities and Exposures) numbers or register security issues in popular vulnerability databases. Only Angular prominently highlights patched vulnerabilities in the release notes of new library versions; the other libraries often mention unspecific “security fixes” along with a long list of other changes, if they are mentioned at all.

In addition to the difficulty of finding out about vulnerabilities, it is very rare to find information about the range of versions affected by a vulnerability. Given this general lack of readily available information, security-conscious users of a library do not have much of a choice other than to update every time a new version is released. Updating is often “painful,” however, for a number of reasons ranging from the short release cycles common in web library development to breaking API changes and the need for testing after each library update.

To end this survey on a positive note, we highlight the security practices followed by Ember (<https://emberjs.com>). Its maintainers commit to patching long-term support

releases so that library users do not need to deal with frequent breaking API changes. Ember maintains a security announcement mailing list, registers CVE numbers, mentions security issues in release notes, lists the range of versions affected by a vulnerability, and provides a dedicated email address to report security issues. These practices ease the burden of dealing with vulnerabilities. Let's hope that other library maintainers will follow suit.

Third-Party Components

The previous paragraphs assumed that website developers directly include libraries, which makes it their responsibility to keep them up to date. The results of the web crawls, however, show that this assumption often does not hold in practice. In fact, many website developers load external scripts such as advertisements, tracker code, or social media widgets. These third-party components sometimes include libraries on their own. This study has shown that such behavior may cause duplicate inclusions of a library, and that these indirect inclusions come with a higher rate of vulnerability. Under some circumstances, sandboxing the third-party code in an iframe may be an option to limit the damage. In general, however, website developers must rely on the maintainers of these components to update their code.

CONCLUSION

Most websites use JavaScript libraries, and many of them are known to be vulnerable. Understanding the scope of the problem, and the many unexpected ways that libraries are included, are only the first steps toward improving the

Related articles

➡ Dismantling the Barriers to Entry

Rich Harris

We have to choose to build a web that is accessible to everyone.

<https://queue.acm.org/detail.cfm?id=2790378>

➡ JavaScript and the Netflix User Interface

Alex Liu

Conditional dependency resolution

<https://queue.acm.org/detail.cfm?id=2677720>

➡ MongoDB's JavaScript Fuzzer

Robert Guo

The fuzzer is for those edge cases that your testing didn't catch.

<https://queue.acm.org/detail.cfm?id=3059007>

situation. The goal here is that the information included in this article will help inform better tooling, development practices, and educational efforts for the community.

Tobias Lauinger is a Ph.D. student at Northeastern University with an interest in Internet-scale measurements of everything security and beyond.

Abdelberi Chaabane is a security researcher at Nokia Bell Labs whose work focuses on empirical large-scale studies to measure and understand online threats.

Christo Wilson is an associate professor at Northeastern University whose work focuses on security and privacy on the web, and algorithmic transparency.

Copyright © 2018 held by owner/author. Publication rights licensed to ACM.