Is The Internet Dependent on Dependencies?

John Carroll
Computer Science Dept.
Worcester Polytechnic Institute
Worcester, MA, USA
jccarroll@wpi.edu

Shikha Pandey
Computer Science Dept.
Worcester Polytechnic Institute
Worcester, MA, USA
spandey3@wpi.edu

I. Proposal

A. Problem Description

As the internet has developed, many web pages have transitioned to use JavaScript libraries in order to enhance the user experience of their website. JavaScript allows the web page to offload a large percentage of work to be rendered locally on the user's machine. By doing this, the web page will experience significant improvements to its responsiveness and allow for much more graphically complex content to be displayed. One benefit to JavaScript is that it can be compiled into libraries that may be accessed remotely, locally, or a combination of the two. Many websites, including high profile pages such as Apple.com and Facebook.com, use a number of remote JavaScript libraries in order to enhance the user experience while also keeping the libraries up to date. Past works have analyzed the qualities and level of maintenance needed to utilize these remote JavaScript libraries, although there is little research into their necessity to host a responsive and functional web page. In this project we will analyze the importance of these library dependencies and consider whether a large number of websites depending on the same library could create several failure scenarios should these libraries become unavailable.

First, we must analyze the inclusion of library dependencies and measure the impact that they have on a website's functionality and stability. In simple terms, a library dependency is a component to the source code of a web page that depends on an external library or component. The inclusion of libraries through dependencies simplifies the development process of a web site, as the developer does not have to implement every necessary component or library that is needed to provide functionality. It is commonly known that library dependencies and the inclusion of external code expose a web page to a number of security vulnerabilities. Although these vulnerabilities are the primary concern of network developers, a question that is less considered is how a web page would function if these dependencies became inaccessible. The inclusion of library dependencies is a very common practice in web development, further emphasizing the importance of analyzing the impact and risks that these libraries create. A study done by White Source [1] has shown that the average web project includes an average of 64 open source dependencies. In many developers' eyes, this figure represents 64 vulnerabilities to a web page's security, functionality, and performance. Despite all of these risks, it is still common practice to include library dependencies in web development projects due to the extensive functionality that they can enable.

One of the most common library dependencies seen on modern web pages is the inclusion of external JavaScript libraries. At its core, JavaScript is designed to piece together all of the elements to a web page. The twenty-first century has seen rapid growth with the amount of content on the internet, and we are now reaching a point where interactive web content is limited by the ability to render in real time from the server. This expresses the importance of JavaScript to modern, interactive web content. In order to function correctly, a web developer must include the necessary JavaScript libraries either locally or fetch them from a remote server. Third party JavaScript libraries, such as jQuery or Google Analytics, have become essential elements to many website's functionality. The most common approach to including these libraries is through a Content Distribution Network (CDN). "A CDN is a group of servers spread out over many locations. These servers store duplicate copies of data so that servers can fulfill data requests based on which servers are closest to the respective end-users" [2]. The primary benefit to a CDN is that it provides very fast service and is affected very little by the amount of traffic on the web page. Using a CDN also lowers the number of bandwidth being used by a network host. The alternative to using a CDN would be to include the JavaScript library locally. The benefit of including a JavaScript library locally is that the website is no longer reliant on the library's host in order to function correctly. In addition to this, the web page developers have full control over the JavaScript code that is distributed to the page's viewers and can modify and store this information as they choose. The primary drawback to this approach is that it requires a lot of maintenance, as these libraries must be manually updated by the website developer. In addition to this, the viewers of the page must download these scripts from the host every time that they visit the page, even if they are identical to scripts that are already downloaded and stored in the browser cache. Most CDN's are already configured in your browser's cache settings and the JavaScript inclusions will already be downloaded, providing the user with a more responsive experience.

From these descriptions alone, one would assume that smaller web pages would likely utilize remote inclusions to improve performance meanwhile larger web pages would use local inclusions to maintain control and security. The reality is actually quite the opposite. The vast majority of high traffic websites, including Amazon, Apple, and more, utilize content distribution networks to include a variety of third-party JavaScript libraries. These libraries provide critical components to the functionality of each of these pages. Google AdSense, for example, is a component of a third-party library that provides a primary source of income to many smaller web pages. We plan to analyze both low and high traffic web pages and study the effect that these libraries have on the web page. In a worst case scenario, an entire CDN would go down and the web page would no longer have access to the necessary JavaScript libraries. There are a number of fallback procedures in place to ensure that the web page will still render without the inclusion of the necessary libraries. For example, if a web page requested the jQuery library and host was offline, Google has an API that provides the same functionality but with a much less efficient run-time. Our plan is to benchmark a variety of websites that are rendered with missing library dependencies and determine if they are able to render at all, if they still provide the intended functionality, and if they appear and perform as if the required libraries were properly included.

B. Limitations of Current Approaches

In today's modern era of the internet, library dependencies essentially exist to make life easier for programmers. The arduous manual process of implementing every single component of the system, can now instead be imported into a library automatically within seconds. Offloading work to prewritten components allows developers to work efficiently and effectively, but this does not come without its caveats. Library dependencies can be the cause of many problems simply due to the total number of lines of code involved, and the reliance on third-party libraries versus in-house code. This makes it hard to track all of the dependencies, leading to vulnerable and outdated code, which in turn can have disastrous consequences.

Website Failovers are bound to occur when a network comprises software homogeneity. Software homogeneity is when a network has all systems running the same operating system, same dependencies, or other piece of software in a network. [3] This can be disastrous in circumstances where malware can exploit the same vulnerability in all nodes on the network. As Jorgen Tellness, from the University of Bergen states, 54.8% of all web sites depend on the Javascript framework jQuery, including mega-sites such as Amazon.com, Microsoft.com, Wikipedia and Tumblr—and the trend is rising. [3] A vulnerability in jQuery would impact over 50% of the world's websites, as the malware distributed through an exploited jQuery vulnerability would lead to massive website failover on the internet. [3] To combat this predicament, most Javascript libraries benefit greatly from being served from a Content Delivery Network (CDN), mostly due to the increased chance of the user already having the library in their browser cache, thus decreasing loading times and bandwidth usage. [3] Furthermore, another modern day approach to combating this issue is through the use of a dependency management system. These systems act as tools that assist in ensuring licence compliance, handling mapping of explicit library dependencies and notifying developers about new releases and known vulnerabilities. [3] Yet, as many researchers including Tellness, have stated," the current state of dependency management systems is that there are no systems that satisfy our needs", as these systems are not effectively able to control, protect, and efficiently run library dependencies. [3]

Furthermore, the dangers of malicious modules are not unknown. A malicious package has the ability to damage the code base regardless of where it is in the required hierarchy and or even if it isn't passed sensitive data. [4] Third party code can be dangerous, and even well-meaning package authors can become a victim to phishing attempts and password leaks. Adding two factor authentication as well as being conscious of the number of modules/dependencies you have installed is the best way to lessen an organization's attack surface. [4] Moreover, there are auditing tools that currently exist that can scan installed dependencies and compare them to a blacklist of modules that carry known vulnerabilities. [4] Yet these tools, while powerful, are reactive approaches, and will not proactively detect and contain threats.

Lastly, "dependency hell" has the ability to impact thousands of npm users and websites. Dependency hell is defined as "the condition when one or more pieces of software need two or more conflicting dependencies (usually transitive ones) installed in the same environment". [5] In March of 2016, thousands of developers saw website outages wipe out the internet, and it was found that the root cause of the problem was a developer pulling his packages out of npm, a dependency manager for JavaScript. [6] The unpublished packages included an 11-line left-pad function that was downloaded 2.5 million times in the month prior to the incident, an indication that the function was extensively used in other applications. [6] One of those applications turned out to be Babel, a JavaScript compiler of many popular websites. Babel contained a module that depended on the left-pad function. When JavaScript applications stopped compiling altogether in Babel, thousands of developers started panicking and saw outages wipe across the internet.[6] To combat this issue from happening again, developing teams that are using open source libraries are told to keep track of their directive and transitive dependencies, and softwares such as Whitesource exist that can provide better visibility into open source usage, but these software solutions are reactionary at best. [6] Therefore, it can be argued that current approaches to optimizing library dependencies are not sufficient and sustainable for ensuring website stability.

C. Main Challenges

There are many challenges that can be foreseen as we begin our analysis of library dependencies and the impact they have on web page functionality and stability. The first and most broad challenge that we must approach is deciding which websites we should analyze. In order to provide the most meaningful data, comparison, and analysis, we need to analyze websites persisting of many different categories. We will begin our approach by inspecting websites that experience a great amount of traffic, little traffic, as well websites that consist of many dependencies versus very few dependencies. We also aim to examine websites that include dependencies locally as well as those that fetch them remotely. It will also be fruitful yet challenging in our analysis to examine the version of local dependencies that are being used, as well as figuring out the process in which we can view certain websites but block out any remote dependencies present. Lastly, it will be a difficult process to collect website benchmarks, on performance, access to data, etc., but we seek to overcome this challenge through extensive research and test trials. We believe that the greatest challenge that lies before us, is the fact that this topic has not been a prominent and thoroughly researched theme in Computer Networks Academia, thus making this proposal a difficult trail to overcome as lack of background information has led to gaps in knowledge. Yet through this proposal, we aim to bring to light the issues that library dependencies have in order to make the Internet a more safe and secure environment.

II. Literature Review

A. Introduction

As previously discussed in the main challenges of this project, there are several methods of both collecting the necessary information and simulating the failure scenarios that may occur from a large JavaScript library becoming unavailable. There are a number of works that have collected data on the inclusion of JavaScript dependencies on strictly high profile web pages, and others that have used scrapers to analyze millions of pages and provide a broad datasets.

B. Existing Works

Nikiforakis et al. [7] performed a large-scale evaluation of Remote JavaScript Inclusions, evaluating more than three million pages of the top 10,000 Alexa sites and identifying how remote JavaScript libraries, due to their untrustworthy library providers, have the ability to compromise Top Internet websites.The team of researchers argued that there were many possible failure scenarios that could occur, as remotely included code can not only have bugs, which can include vulnerabilities, but that these remote hosts could also be malicious and use scripts to attack users as well as exfiltrate data from websites [7]. After conducting the largest to data web crawl of over 3 million pages and recording 8.5 million remote inclusions, where each set of pages was obtained by querying the Bing search engine for popular pages within each domain and, and utilizing HtmlUnit, a headless browser, which they used in their experiments as a placeholder for Mozilla Firefox 3.6., they were able to fully execute the inline JavaScript code of each page, and accurately process all remote script inclusion requests [7]. The researchers concluded that this method has its limitations as the visited pages that included remote scripts based on non-Firefox user-agents, were missed by the crawler. In order to account for these user agents, a crawler would need to be implemented that is able to fetch and execute each page with multiple user agents and JavaScript environments. An alternative option would be to use Rozzle, which is a system that explores multiple execution paths within a single execution, in order to uncover environment-specific malware [7].

Hejderup's [8] study focused on exploring why JavaScript modules relied on vulnerable components from dependency point of views. He argued that although a source code perspective could identify potential security vulnerabilities in the code sections of the software, both static and dynamic analysis are important [8]. And while on the server-side of JavaScript where dynamic analysis requires semi-manual intervention to explore all code execution paths, the pure static analysis has low accuracy on call-graph construction [8]. Along with being a very arduous process, the rate of false positives and false negatives in source code analysis are immense [8]. Therefore, the author argues that the best course of study for third-party components is from a dependency perspective. A crawling strategy was devised to extract dependency information modules from the npm registry and a vulnerability scanner was implemented with a low false positive and false negative rate [8]. From there, an extension of the vulnerability scanner was implemented that would assess the depth and width of the "dependencies of dependencies" chain for the cascading effect that would measure the number of vulnerable modules for each level in the chain and measure the time latency of updating from a vulnerable version to a non-vulnerable version [8].

Lauinger et al. [9] explored how third party JavaScript libraries such as jQuery, although able to enhance the functionality of websites, are also able to create attack vectors that can allow websites to be compromised. Their study focused on using data from over 133,000 websites, and discussed how in 37% of those websites, at least one library with a known vulnerability [9]. The research team, first identified client-side JavaScript libraries, and collected metadata about popular JavaScript libraries, including a list of available versions, the corresponding release dates, code samples, and known vulnerabilities. [9]. Their approach next was to determine if JavaScript code found in the wild, was a known library or not. Lastly, websites were crawled while keeping track of causal resource inclusion relationships and matching them with detected libraries. Casual trees were generated by observing resource requests through the network view of the Chrome Debugging Protocol. [9]. Furthermore, the research team decided to divide their crawling parameters into two different datasets [9]. The Alexa Top 75k domains were crawled along with 75k randomly sampled domains from a snapshot of the ".com zone" to diversify the crawling data [3].

Zerouali et al. [10] researched the impact of npm JavaScript package vulnerabilities in Docker images. The research team looked at 961 images from three official repositories that use Node.js, and 1,099 security reports of packages available on npm. They argue that outdated npm packages in Docker images heavily increased the risk of potential security vulnerabilities in Docker maintainers [10]. In their research, they chose to begin by identifying candidate docker images, and then working to extract npm package data. They focused on node image2, which contained Node.js and the candidate images were run locally while identifying the installed npm package versions. [10]. They next focused on collecting security vulnerabilities that came up and computing the technical lag that succeeded based on the npm packages that were found [10].

Liu [11] argues that developers have pushed the boundaries of what JavaScript was originally intended for. Entire desktop applications are now being rebuilt using JavaScript and cited Google Docs office suite as just one example [11]. His research states that large applications make it hard to manage the complexity of loading the required JavaScript files and their dependencies and this problem can be aggravated when introducing multivariate A/B testing, a concept that has built Netflix's API, since its adoption of Node.js [11]. Multivariate testing makes it very hard to manage conditional dependencies. The author in his study uses modules to "encapsulate features that provide the ability to build an abstraction layer on top of the personalization facets that gate the A/B tests" [11]. He maps the eligibility for a test to a specific feature, which is then mapped to a module. This causes the JavaScript payload to effectively be resolved for a given subscriber, as it is then able to determine the subscriber's eligibility for each of the currently active tests [11].

III. Methodology

A. Creating a Dataset

In order to best analyze the potential failure scenarios from missing JavaScript dependencies, it is important to look at websites that handle high amounts of traffic and would create the largest failure scenario if it were limited in functionality. For this reason we chose to analyze a list of the Top 50 Websites in the United States, provided by Alexa.com. These websites are ranked by curating several statistics, including the daily time that is spent on the website, the average number of pages viewed on the site per visitor per day, the percent of web traffic that came from search, and the total number of sites that link to the given website. One potential limitation to this approach is that the list provided by Alexa.com only includes the host names of the sites. Many of these websites have critical elements that are nested in pages that are not immediately accessible from the index page. For this reason, we have chosen to utilize a tool called Wappalyzer. This tool allows us to analyze all of the JavaScript libraries that are included on a website and all of its pages. With access to this information, we can simulate the failure scenarios of the entire website when removing a required dependency.

B. Identifying Libraries

As previously mentioned, we chose to use a tool called Wappalyzer to identify all of the required JavaScript libraries that are included in the high-traffic websites in the dataset. Wappalyzer is a broad tool that was built to analyze the technology stack of a website. Although this tool is primarily used for market analysis, research, and lead generation, it also provides an accurate and thorough list of JavaScript libraries that are used on a site and all of its included pages. In addition to libraries, Wappalyzer is also able to identify which Content Delivery Networks are being utilized on a host as well. Since many high-profile websites utilize CDN's to fetch the most updated versions of its JavaScript dependencies, this piece of information could be used to analyze the potential failure scenarios from a much broader scope than just a single JavaScript library. If an entire Content Delivery Network was to go offline, many websites would resort to use outdated versions of its JavaScript dependencies that are stored locally, while other pages would lose these libraries entirely. This may pose a number of security vulnerabilities and also risk the stability of the entire page if the libraries are incompatible or missing. Analyzing the impact of all of these elements will provide crucial information to understand the possible failure scenarios at stake.

C. Classifying Functionality

If a website is forced to run with outdated or missing JavaScript dependencies, there is a chance that elements of the page will not perform as intended. With this being said, we must classify different levels of functionality in order to identify how a specific website is performing while in the failure case scenario of missing a JavaScript dependency. We chose to use four levels of functionality to classify a website in a descriptive manner, while still providing enough data to show meaningful trends. Starting from the highest level of functionality, there are "Fully Functional" websites. In order to receive this level of functionality, the website must perform, function, and appear exactly how it looked when all of the required libraries were included. All elements of the website must function correctly, and the website should show similar speed and performance when compared to its original state. Next, there are websites that have "Adequate Functionality". This would dictate that a user on the website can still perform the intended functionality, although there may be elements of the page that do not work properly. For example, if a user of an eCommerce site is still able to shop for items and successfully purchase them, then the website is functioning in the way that it was intended. Many users of ad-blockers already view websites in this state of functionality, as many ad-blockers completely block common JavaScript libraries that are used for targeted advertising, such as Google Analytics. A website that performs less than this would be classified as "Low Functionality". At this point, the website will still compile although it will not be able to provide the intended functionality of the page. There may be significant elements of the page that are no longer visible or interactive. For example, if a video-streaming platform were to render completely and show only the comments and not the video itself, then this would dictate a "Low Functionality" page. Lastly, there are pages that will have "No Functionality", dictating that the page no longer renders, returns an error, or provides absolutely no functionality to the user.

D. Simulating Failure Scenarios

Lastly, we must design a method to simulate a failure scenario of these JavaScript dependencies. For example, if an entire Content Delivery Network were to fail and a website did not have any version of the necessary JavaScript dependencies stored locally, then they would no longer be included in the page. In order to simulate this scenario, there are several methods of preventing the JavaScript dependencies from being included in the web page. The most thorough method of accomplishing this would be to use a Proxy to drop HTTP objects from the web site. We found that mitmproxy provided a free and open source method of accomplishing this. By searching for instances of common JavaScript files, such as www.google-analytics.com/ga.js (Google Analytics), and dropping these objects, we can then simulate how the page would interact without these dependencies. In addition to using a proxy, a much more simple solution would be to use a browser extension, such as Adblock Plus, that is capable of blocking certain page elements. By using Wappalyzer, it is already known what JavaScript libraries are being used on a website. By manually blacklisting each of these elements on the AdBlocker, we can simulate a failure scenario without using a proxy.

E. Evaluation

Finally, we will perform an evaluation on the specified web sites when put through a failure case scenario. We will use the previously mentioned tools to block specific JavaScript libraries, and determine the functionality of the web site by using the scale that was described in our section on classifying functionality. To begin, we must consider the fundamental purpose of each web site and the functionality that it is meant to provide to the user. We will then run our experiment and determine the level of functionality that the web site provides in the absence of certain JavaScript libraries in its 'failure scenario' state.

IV. Empirical Results

Our research to analyze potential failure scenarios of the internet from JavaScript dependencies began by compiling a dataset to best demonstrate the potential impact. For this reason, we have chosen to compile the list of the Top 50 Websites in the United States, provided by Alexa.com. Our decision to use the United States analytic over a global analytic was due to the fact that assessing the functionality of a website in English provided more meaningful results, as content was all readable without translation. In addition to this, browser translate extensions often use JavaScript libraries, so our experiment would have likely limited this functionality as well. Next, we needed to identify all of the JavaScript libraries included on each website so we could analyze the impact on functionality that each of these libraries provide. We used a browser extension, Wappalyzer, to record all of this data. Lastly, we needed a method of classifying functionality of each website when they are run with missing or outdated JavaScript dependencies. We chose to use a four-level scale to address functionality, as detailed in Figure 1 below.

Lastly, we must simulate a failure scenario in the delivery of these JavaScript libraries. Using Google Chrome's developer console, we were able to manually disable JavaScript elements and libraries for individual websites and then observe the functionality. The changes made in the developer console remained on all subpages as well, so we were able to navigate throughout the website within the simulated failure scenario to assess functionality.

Functionality Level	Description
Full Functionality	Website performs, functions, and appears exactly how it did when all of the JavaScript libraries were included and using the correct version.
Adequate Functionality	Website can still perform intended functionality, although there may be elements of the page that do not work properly. Ex. Amazon functions as intended by various menu bars are not working properly
Low Functionality	Website will still compile, although it is no longer capable of providing the intended functionality. Ex. Amazon's add to cart feature is no longer functioning, so users cannot purchase items on an eCommerce site.
No Functionality	The webpage no longer renders, returns an error, or provides no functionality to the user.

Website	Libraries	Functionality
Google.com	Boq; Wiz;	Full
Youtube.com	Polymer3.4.1; SPF	Adequate
Amazon.com	React; core-js	Adequate
Facebook.com	-	Full
Yahoo.com	Modernizr2.6.2; core-js	Adequate
Zoom.us	Bootstrap3.4.1; jQuery3.4.1; jQuery(Fast-path); Preact 8; Vue2.6.11-csp; Vue(Fast-path); core-js	Limited
Reddit.com	React; core-js	Limited
Wikipedia.org	jQuery3.4.1	-
Myshopify.com	jQuery2.0.3; jQuery (Fast-path)	Adequate
Office.com	jQuery3.3.1	-

V. Conclusion

As the internet continues to grow at unprecedented rates, so does the need for JavaScript libraries and frameworks, which act as the backbone of website development, security, and performance. Javascript Libraries and packages come with dependencies, which are defined as the third-party code that the application inherently depends on and cannot function without. Library Dependencies are undoubtedly the most important crutch to a website's features, functionality, and stability. JavaScript dependencies are used inside the source code of webpages to simplify the development process of a website, essentially making it easier for developers to create the website, while not having to go through the arduous process of implementing every single library that provides the functionality. External JavaScript libraries are omnipresent throughout the web due to their ability to piece multiple elements of web pages together. This same feature makes external JavaScript libraries one of the most common library dependencies and utilized in almost all of the Top 100 websites on the modern day internet. The primary aim of our study was to analyze the significance of these library dependencies as it was hypothesized in a scenario where these dependencies become unavailable, accessibility and functionality of the websites that rely on these dependencies would suffer as well. To simulate these results, the Top 50 Websites provided by Alexa.com were analyzed due to their capacity to handle high amounts of website traffic. Wappalyzer was utilized to identify JavaScript libraries in the website dataset, while Google Chrome's developer console was utilized to manually disable JavaScript libraries from the given websites. Next, different levels of functionality were identified to categorize the severity of the failure case scenario: Full functional, Adequately functional, Low functionality, and No functionality. Our results produced the following conclusions: The first being that, all websites analyzed were using JavaScript Dependencies, and almost all websites were affected by the deletion of these dependencies, leading to restricted and deficient capability, and specifically navigation errors. The results are in the early stages of analysis, but even with a limited quantity of the analysis complete, it can be assumed that most websites in the dataset will not reach Low or No Functionality levels with the deletion of selected JavaScript Dependencies. This study is a first of its kind, harboring into a never before studied area of the Internet Industry, and an often disregarded topic when it comes to Web Security. The impact that this study hopes to have is to bring to light the drawbacks associated with heavy website reliance on Third Party JavaScript dependencies, along with the devastating failure scenarios that these dependencies could give rise to. This study also hopes to encourage the review and inspection of how these dependencies can be implemented and optimized to prevent the loss of website functionality, stability and performance, and if potential solutions such as Dependency Management Systems could be applied to protect against any vulnerabilities, and hinder any sort of website accessibility.

VI. References

[1] https://www.drdobbs.com/open-source/indirect-dependencies-are-killing-open-s/240154702

[2] https://developer.mozilla.org/en-US/docs/Glossary/CDN\#:~:text=A\%20CDN\%20

[3] Dependencies: No Software is an Island

[4] https://medium.com/intrinsic/common-node-js-attack-vectors-the-dangers-of-malicious-modules-863ae949e7e8

[5] https://blog.tidelift.com/dependency-hell-is-inevitable

[6] https://resources.whitesourcesoftware.com/blog-whitesource/why-open-source-dependencies-are-your-blind-spot

[7] You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions

[8] In Dependencies We Trust: How vulnerable are dependencies in software modules?

[9] Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web

[10] On the Impact of Outdated and Vulnerable Javascript Packages in Docker Images

[11] JavaScript and the Netflix User Interface