When Third-Party JavaScript Meets Cache: Explosively Amplifying Security Risks on the Internet

Tao Hou*, Shengping Bi†, Mingkui Wei‡, Tao Wang†, Zhuo Lu§, and Yao Liu§
*Texas State University, San Marcos, TX, USA, taohou@txstate.edu
†New Mexico State University, Las Cruces, NM, USA, {sbi, taow}@nmsu.edu
‡George Mason University, Fairfax, VA, USA, mwei2@gmu.edu
§University of South Florida, Tampa, FL, USA, {zhuolu@, yliu@cse.}usf.edu

***Abstract*—Today's web applications feature the proliferation of third-party JavaScript inclusion, which incurs a range of security risks. Although attack strategies by manipulating third-party JavaScript files have been widely investigated, the adverse impact caused by third-party JavaScript inclusion and caching does not receive much attention. Specifically, when a malicious script is cached, it can revive and bite every time when a user visits any website that includes it, leading to a much worse effect of the attack. In this paper, we present the first comprehensive study on Alexa top one million websites to investigate how likely third-party JavaScript inclusion and caching can make an attack large-scale and long-lasting, and further to uncover insecure practices that carelessly or inadvertently exacerbate the attack impact. We also discuss potential solutions to improve current practices to minimize the security risk associated with third-party JavaScript inclusion and caching.**

I. Introduction

It has become a common practice for a website to include third-party JavaScript libraries on the Internet. Many service providers, such as jQuery and Google Analytics, create and provide third-party JavaScript libraries for the Internet to use. However, loading third-party scripts is potentially dangerous as they gain the same privileges as local scripts of the webpage during the execution [1], which yields to a range of web attacks, such as cross-site scripting (XSS) [2], [3], cross-site request forgery (CSRF) [4], and cross-site script inclusion (XSSI) [5].

Typically, resources from different websites should be isolated from each other to avoid potential cross-site attacks. Nevertheless, if two websites include the same third-party scripts, they become correlated to each other. Consequently, they will face the same risks if the third-party scripts are exposed to attacks mentioned above. Further, such a practice produces an inadvertently negative impact from the security perspective when it meets the browser caching mechanism, which saves previously fetched web resources at local storage to accelerate website loading performance. In particular, successfully compromising a third-party script is usually a one-shot deal and only limited websites may be affected. But if the compromised script is cached, it can be loaded locally whenever a user visits any website that includes the same script. In this way, caching such a script can in fact make the attack impact both large-scale and long-lasting.

For example, Alice goes to a coffee house and connects to a free public WiFi network, which is set up by Bob, an attacker. Alice knows that a public free WiFi may be insecure, and thus she visits websites cautiously and avoids inputting sensitive information like her password. Nevertheless, if Bob is able to impersonate the third-party JavaScript sever and sends a modified, malicious third-party JavaScript file to Alice, Bob can generate a post-attack effect. Specifically, this malicious script will be cached by the browser. The next day, when Alice connects to a secure network and inputs her password to any website that includes the same script. The cached malicious script will revive and steal the password from Alice.

This example shows that third-party JavaScript inclusion and caching opens a door for indirectly compromising a website. However, existing research mainly focuses on attack strategies that take advantage of the vulnerability of JavaScript to successfully compromise users' privacy [6]–[10]. The adverse impact caused by third-party JavaScript inclusion and caching seems not receiving as much attention. Nevertheless, such an impact should not be left untended, but be contained or minimized. According to our study on Alexa top one million websites, we find that as many as 824,290 websites include third-party scripts, and a third-party script can be included in up to 504,692 websites. This indicates that the aforementioned attack example could be common for Internet surfing. Further, our study reveals that 51.04% third-party scripts are set to cache for more than 100 days, indicating that this attack may impose a long-lasting effect on victims.

Hence, rather than presenting a particular attack against a third-party script, we provide the first comprehensive Internet-wide study to evaluate current practices of third-party JavaScript inclusion and caching from two perspectives:

- **Attack Scale:** Websites are becoming heavily correlated due to inclusion of widely-used third-party scripts. The transitivity of the inclusion (i.e., a script may include another script, which further includes more) may complicate the correlations between websites. It is essential to track each third-party script to quantitatively understand how current inclusion practices can inadvertently increase the potential scale of an attack launched from a cached script.
- **Risk Duration:** Caching is a mechanism primarily designed for efficiency but not for security. As caching may allow a malicious script to survive in a local browser for quite a while, it is necessary to revisit current Internet practices of third-party JavaScript caching setting from

the security perspective.

In this study, we meet following challenges and propose corresponding techniques to address them.

(1) A website may indirectly include a third-party script (e.g., a website includes a third-party script J_1, which further includes another one J_2). Indirectly included scripts complicate the inclusion relationships. We must be able to identify all indirectly included scripts and track the inclusion relationships between different scripts. To solve this, we develop a customized, automated toolkit on top of Chromium [11] that can intercept the requests and responses between clients and servers, and exactly track the scripts that initiate each request. Further, we define two different tree structures to capture the inclusion relationships between websites and scripts.

(2) When a third-party script expires in the cache, the browser needs to check whether the script on the server is updated or not. If updated, the browser fetches the new version from the server, otherwise it continues using the old script. Therefore, rather than just evaluating the expiration time of the script, it is necessary to measure its life time, which is defined as the time duration that a script remains unmodified on the server. However, it's difficult to exactly predict when a script will be updated in the future. We propose a time bounding technique that can accurately bound the life time to solve this.

We study the third-party JavaScript inclusion and caching of Alexa top one million websites by collecting all associated scripts and caching configurations in HTTP headers[1]. We find that the Internet has largely overlooked the risk of caching third-party scripts and identify careless, insecure practices. Our major findings can be summarized as follows.

• **Complicated and insecure inclusion relationships:** We find that third-party JavaScript inclusions are quite complicated and intricate. Our study indicates that 56.84% websites have indirectly included third-party scripts, and the recursive inclusion achieves the maximum height of 18. Such intricate inclusion relationships substantially complicate the third-party script management and confuse website developers. In particular, we find that there are indeed a number of websites (89,723) that try to relocate included third-party scripts on local servers, such that browsers can load these scripts from their servers to reduce security threats. However, as many as 13,746 (15.32%) of them neglect to store at least one indirectly included third-party script on local servers.

• **Poor maintenance of third-party libraries:** A well-maintained third-party script should be updated in a timely manner. Nevertheless, we find that 43,665 third-party scripts have not been updated by third-party providers for at least two years. Because the browser reuses the stale cached script unless it is updated on the server, rarely updated third-party scripts can advance the adverse impact of the attack launched from a cached third-party script.

[1]Note that HTTPS is a version of secure HTTP communication and has the same header information. We use the HTTP header to indicate either HTTP or HTTPS header unless otherwise specified throughout this paper.

• **Non-HTTP-conforming and risk-incurring settings of expiration time:** HTTP 1.1 requires that the expiration time should be set no more than some small fraction of the life time (the recommended typical setting is 10% [12]). However, our study reveals that the expiration time of at least 10.65% third-party JavaScript files is even 10 times greater than the life time. This can significantly increase the adverse impact of third-party JavaScript inclusion and caching.

II. Preliminaries

A. Third-Party JavaScript and Security

Client-side JavaScript has been extensively used in modern web applications, since it allows the source code to be loaded and executed at the client's browser, thus alleviating web-server workloads and achieving fast, responsive interactions with users. In particular, developers can enrich the websites by using the `<script>` tag to include a third-party script.

Despite their popularity, third-party JavaScript may incur a range of security issues [2], [3], [5], [13]–[18]. Further, if an attacker is able to inject malicious codes into the cached script, the script can be executed whenever a user accesses any website including the script. This, in turn, enables the attacker to launch attacks of password interception, privacy tracking [19], and phishing [20].

B. Browser Caching Policy

Today's web browsers all come with the resource-caching mechanism [21], which caches web contents in local storage to reduce the page loading time for the next-time visit. The essence in cache management for a browser is to decide when it should update the cache to ensure that the displayed web content is up-to-date. This is generally determined by metrics in HTTP headers in the following two categories.

• *Client-side metrics:* Such metrics describe how a web resource (e.g., a webpage, script or image file) is cached in the local storage of a browser. They include `Expires` in HTTP 1.0 and `Cache-Control` in HTTP 1.1. `Expires` is a simple timestamp to indicate when the resource expires. `Cache-Control` includes a number of metrics, including `max-age`, `no-cache`, `no-store`, `must-revalidate`, and `immutable`. `max-age` is most frequently used to state the maximum amount of time in seconds that the resource stays fresh in the cache. Other metrics included in `Cache-Control` refine the behavior of JavaScript caching.

• *Server-side metrics:* `Etag` and `Last-Modified` are the signature and the last modified time of a web resource on the server, respectively. At the client, when a cached resource expires (based on client-side metrics), the client browser sends an HTTP request with the values of the previously stored `Last-Modified` and `ETag` to the server. The server then compares these values with current values of both metrics to check whether the resource is modified. If modified, the server returns the latest version; otherwise, it returns a message of "304 Not Modified".

III. Understanding the Adverse Impact

In this section, we first use an attack example to discuss the adverse impact of third-party JavaScript inclusion and caching, and then present the objectives of the research in this paper.

A. Attack Example

JavaScript caching indeed reduces webpage loading time; on the other hand, however, it incurs the security concern of attacks across multiple websites. As an example shown in Figure 1, `sample.js` at a third-party service provider is included in websites $1{\sim}n$. When a user visits website 1 for the first time, `sample.js` is retrieved from the third-party service provider and cached in the user's local storage. When the user subsequently visits any of the websites $1{\sim}n$, the browser will simply load `sample.js` from the local cache unless the cached script expires. If `sample.js` is manipulated by an attacker when the user visits website 1, the browser will run the cached, compromised `sample.js` during the subsequent visits to any of the websites $1{\sim}n$.

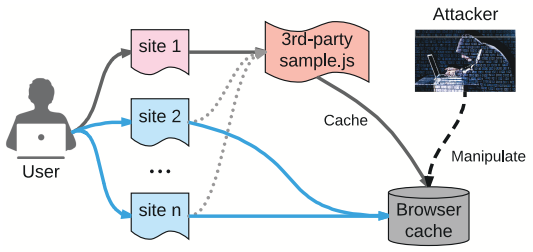

Fig. 1: Adverse impact of 3rd-party JS inclusion and caching.

B. Research Objectives

The example demonstrates the potential attack impact of third-party JavaScript inclusion and caching, which consists of three key components: (i) the initial attack that injects a malicious version of a script; (ii) the inclusion relationships between websites and scripts; (iii) the caching settings of scripts in local storage.

1) The Initial Attack: The strategies of the initial attack has been widely investigated in the security community [7], [9], [10]. In particular, such attacks include client-side attacks that utilize vulnerabilities existing in a browser to access cached files and modify their contents [10], server-side attacks that try to compromise the script hosted in the third-party server, such as third-party domain re-registration [7] and direct host compromising [9], and in-transit attacks that inject malicious scripts by exploring the communication vulnerabilities between a browser and a server [6], such as man-in-the-middle attack (MITM) using rogue WiFi APs [8]. An attacker can force the refresh of a cached script and deliver a malicious version of the script (e.g., using `<new Image.src="URL/to/malicious.js">`). Recent research [22]–[25] also reveals that when HTTPS is used, although modern browsers always warn users of invalid certificates under in-transit attacks, more than 50% users still click through the warning to visit the corresponding websites. However, as shown in Figure 1, a compromised script can be cached and come to bite later.

2) Inclusion Relationships and Caching Settings: The website-JavaScript inclusion relationships and caching settings are largely determined by the current Internet practices. The website-JavaScript inclusion relationships have a scale consequence of an attack. e.g., a large number of website can be vulnerable when they include the same malicious script (e.g., a large n in Figure 1). Caching settings cause a time consequence of an attack. e.g., a very long caching duration of a malicious script makes an attack long-lasting.

Although such negative consequences depend on the success of an initial attack, we can never focus solely on defending against the initial attack and ignore the fact that inclusion relationships and caching settings may further advance the attack. To this end, we aim to propose strategies to enable an Internet-wide study. Our objectives are twofold: (i) to quantitatively and systematically evaluate current practices of third-party JavaScript inclusion and caching from scale and time perspectives; and (ii) uncovering insecure practices that inadvertently increase the scale or time, thereby escalating a successful initial attack.

IV. Methodology

In this section, we present two tree-based structures to systematically characterize the relationships between websites and JavaScript files.

A. JavaScript Classification

The inclusion relationships between websites and third-party scripts are complicated on today's Internet. For example, a webpage can include a script, which also includes multiple scripts that further include more. We classify scripts included in a website based on two properties: (i) local or third-party scripts, depending on where scripts are hosted; (ii) directly or indirectly included scripts. In what follows, we describe how to track all directly and indirectly included scripts according to their inclusion relationships.

B. Inclusion Relationships

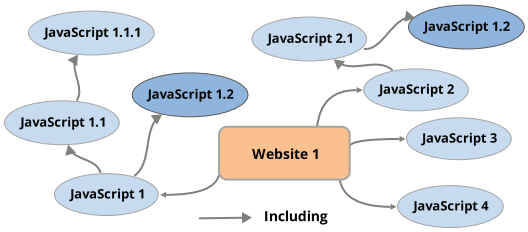

Fig. 2: Example of JavaScript Inclusion Tree.

As discussed, a JavaScript file may be included directly or indirectly by a website. Because of indirect inclusion, the website may contain multiple levels of recursive inclusions among scripts. As long as a third-party script is included in any level, the website may be exposed to the security risk

of caching a compromised version of the script. We propose two tree structures to characterize the inclusion relationships between scripts and websites.

1) JavaScript Inclusion Tree: For each website, we use the depth-first search algorithm to construct the JavaScript Inclusion Tree, which fully characterizes how scripts are included in a given website. In particular, we specify a website as the root; all scripts included directly by the website are added as children of the root; then for any script pair (J_1, J_2) with J_1 including J_2, we add J_2 as a child of J_1. Figure 2 shows an example of the JavaScript Inclusion Tree for the Website 1.

2) JavaScript Backtracking Tree: For each third-party script, we use a backtracking algorithm to build the JavaScript Backtracking Tree, which traces how a third-party script has been included in different websites. In particular, we specify a given script as the root. All the websites and scripts directly including it are added as children of the root. For any child as a script in the tree, we further add all websites and scripts directly including it as its children, and so on. The JavaScript Backtracking Tree is constructed recursively and is complete when all the leaves of the tree are websites. Figure 3 gives an example of the JavaScript Backtracking Tree of a given JavaScript 1.

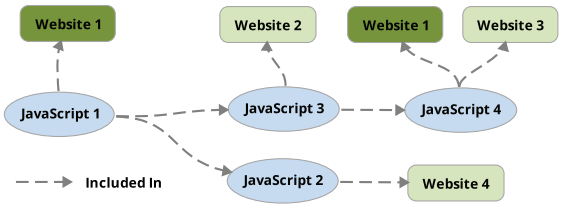

Fig. 3: Example of JavaScript Backtracking Tree.

C. Obtaining Scale Metrics from Trees

According to the two tree structures, we define two major scale metrics to understand the adverse impact of third-party JavaScript inclusion and caching.

• *Number of scripts included in a given website:* This metric is defined as the number of third-party JavaScript files included either directly or indirectly in a given website. A website may be exposed to the cached third-party JavaScript vulnerability, as long as it includes the third-party script at any level in its JavaScript Inclusion Tree. The more the number of scripts included in a website is, the more vulnerable it can be. The metric is measured via the JavaScript Inclusion Tree. In particular, for a given website, we can build its corresponding JavaScript Inclusion Tree to track all the third-party scripts directly or indirectly included. The metric is computed by counting the number of non-duplicate nodes in the tree.

• *Number of websites including a given script:* This metric is defined as the number of websites that either directly or indirectly include a given JavaScript file. As discussed previously, when a cached third-party script has been compromised, visits to all the websites including it become vulnerable. Thus, given a third-party script, we use this metric to understand the scale of the potential vulnerable websites on the Internet when it is compromised. The metric is measured from the JavaScript Backtracking Tree. Specifically, for a given script, its JavaScript Backtracking Tree traces back all the websites that include it. The value of the metric equals to the number of non-duplicate leaves of the tree.

D. Obtaining Time Metrics

Time metrics present another dimension of the security impact associated with third-party JavaScript inclusion and caching. When a compromised JavaScript file is cached, it remains harmful unless the cache is refreshed. As a result, we also aim to investigate the caching duration for a script to understand the current practices on the Internet.

As discussed in Section II-B, the caching time of a script is determined by both client-side (e.g., `Expires` and `max-age`) and server-side metrics (e.g., `ETag` and `Last-Modified`). Client-side metrics indicate the maximum time a cached script remains fresh in the local storage and server-side metrics indicate when and whether a script has been modified.

Intuitively, a compromised script can survive in the cache until it `Expires` or reaches the `max-age`, because the browser only revalidates the cached script when it expires (i.e., to send a request to check if it is modified on the server). It seems that the caching duration can be evaluated based on client-side metrics `Expires` and `max-age` only.

However, this is not always the case: (1) An expired script may still be used after revalidation with the third-party server. In particular, when a cached script expires, the browser sends an HTTP request with the previously stored `Last-Modified` and `ETag` of the script to the server. The server then checks current `Last-Modified` and `ETag`. Many times, the script is still not changed and the server returns a "304 Not Modified" response. Upon receiving the response, the browser continues to keep the cached script in local storage and resets its `Expires` or `max-age`; (2) If a powerful MITM attacker is able to temporarily control the entire connection between a client and a server [6], it can modify `Expires` or `max-age` in the HTTP headers to any arbitrary value to maximize the caching duration. Nevertheless, a large `Expires` or `max-age` does not always indicate a long caching duration. A user may manually make a hard refresh to enforce the revalidation, and a web developer may also configure the script to force the revalidation.

Therefore, both client-side metrics (`Expires` and `max-age`) and server-side metrics (`Last-Modified` and `ETag`) are vital to indicate the caching duration of the script. We define two time metrics to evaluate the temporal impact.

• *Local caching time of a script:* This metric, which is also referred to expiration time, indicates the time duration a script is cached in a browser's local storage until the browser sends a request to check if it is modified. The value of local caching time equals `max-age` in HTTP 1.1 or the time interval from the measurement time to `Expires` in HTTP 1.0.

• *Life time of a script:* This metric indicates the time duration that a third-party script remains unmodified in the server. Intuitively, if a script is not modified frequently on the server, it can be cached in the local storage of a browser for quite a while even with a small value of local caching time, because the server always sends "304 Not Modified" responses. We measure the life time as the time interval between two consecutive changes of `Last-Modified/ETag` of a script.

1) Bounding Life Time: Local caching time of a script can be easily measured from the metrics `max-age` and `Expires`. Nevertheless, as a script may remain valid on a server for hours, days, years, or even never be modified, it is challenging to capture two exactly consecutive changes of `Last-Modified/ETag` to obtain the life time. We propose a time bounding technique to accurately bound the life time.

As a solution, we repeatedly crawling and estimate upper and lower bounds of the life time of each script. Particularly, we record the `Last-Modified/ETag` for each third-party script during our first measurement, and try to find the change of the `Last-Modified/ETag` in our subsequent measurements. Suppose that for a third-party script, we first capture the change of its `Last-Modified/ETag` in the k-th measurement. Then, the lower bound of the life time of the script is the time interval from its `Last-Modified` in the first measurement to the $(k-1)$-th measurement time. As shown in the timeline of JavaScript 1 in Figure 4, we record `Last-Modified` as $JS1.LM1$ during the first measurement, and find it changes to $JS1.LM2$ during the second measurement. Then, the lower bound of its life time is the first measurement time minus $JS1.LM1$. Because we can be sure that it is not modified from $JS1.LM1$ to the first measurement time.

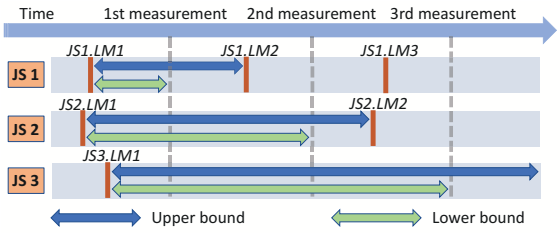

Fig. 4: Examples: bounding the life time.

The upper bound of the life time of a script is computed as the time interval from its `Last-Modified` in the first measurement to the `Last-Modified` that is observed to be changed for the first time in the subsequent measurements. For example, JavaScript 1 in Figure 4 has the first observed change of `Last-Modified` from $JS1.LM1$ to $JS1.LM2$ in the second measurement; then the upper bound of its life time is $JS1.LM2 - JS1.LM1$.

V. Measurement and Analysis

In what follows, we first describe the toolkit for our web data crawling. Then we present overall JavaScript usage statistics, and reveal the insecure practices.

A. Web Data Crawling

We target crawling Alexa top one million websites, which include most popular websites on the Internet. We developed an automatic testing toolkit that works as a headless browser to simulate the visiting of websites. This toolkit is written in JavaScript and developed on Chromium utilizing the Chrome DevTools Protocol [26]. Using this toolkit, we can simulate user visits to each website and collect essential HTTP header information, including cache-related parameters (e.g., Expires, Cache-Control, Last-Modified and ETag), which are important for our study. We used a high performance computing workstation for web data crawling. We repeated our crawling weekly for 12 months to have a continuous and sequential measurements on JavaScript inclusion and caching settings.

B. Third-Party JavaScript Identification

We design a method to identify whether a script is hosted on a local or third-party server. Intuitively, we may compare the domain names of a website and a script. If their domains match each other, the script is considered as local. Otherwise, the script is considered as third-party. This method, however, may not always return the correct results. For example, it may falsely identify a local script as third-party when a website maintains its local resources (e.g. scripts, and images) in an independent server, the domain name of which is different from the website. For instance, we find that website `www.tianya.cn` includes a script `static.tianyaui.com/global/ty/TY.js`, which has a different domain name `static.tianyaui.com`, and can be falsely identified as third-party. Nevertheless, the resources on `static.tianyaui.com` are actually created and used by the website `www.tianya.cn` only.

To provide a more reliable mechanism to identify third-party scripts, we build a host list to identify domains of third-party JavaScript providers. Each entry in the list is the domain of a third-party JavaScript provider. If the domain name of a script matches no entry in the list, it will be identified as local; otherwise, it will be identified as third-party.

We rank the domains based on their popularity, i.e., the number of websites that include any script from the domain. We refer to this number as *frequency of references*. A domain is added to the host list as long as its frequency of references is no less than 2. Table I shows the host list with top 5 frequencies of references from our crawling results.

TABLE I: The 5 most popular 3rd-party JavaScript hosts.

Rank	Third-party JavaScript Host	Frequency
1	www.google-analytics.com	601,717
2	connect.facebook.net	263,048
3	ajax.googleapis.com	190,621
4	pagead2.googlesyndication.com	171,373
5	www.googletagmanager.com	125,597

C. Overall Usage of JavaScript

We successfully crawled 970,698 websites from Alexa top one million websites. From them, we collected 10,684,818

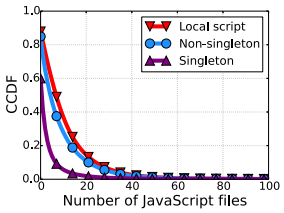

Fig. 5: Num. of 3rd-party scripts per web.

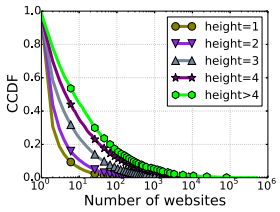

Fig. 6: Num. of webs in backtracking trees.

different scripts in total, in which 3,107,929 were identified as third-party ones. These third-party scripts have been included for 11,149,205 times. Nevertheless, we find that there are a large number, i.e., 2,778,633 (89.40% of 3,107,929), of third-party scripts are included only once. We call them *singletons*, and call the rest of third-party scripts *non-singletons*.

Observing a large number of singletons seems to be inconsistent with the intuition that third-party scripts should be shared across multiple websites. We find that these singletons are mainly due to two reasons: (i) upon the inclusion request from a website, a third-party provider may return the client a script with unique directory or filename dedicated to this website (e.g., `pagead2.googlesyndication.com/pub-config/r20160913/ca-pub-7278577707988162.js`); and (ii) a website developer or third-party provider may also append a unique query string to the URL of a script (e.g., `cse.google.com/cse/cse.js?cx=017837193470827524476:s65nylv7hj0`). Because the caching policy of a web browser identifies a script by its entire URL (including the query string), scripts with the same content but different domains, paths, file names, or query strings are indeed treated as different caching entities.

Singletons are included uniquely by their associated websites. In contrast, the rest 329,296 non-singletons are included 8,370,582 times. Each of these non-singletons is included in 25.42 websites on average, and 824,290 (84.92% of 970,698) websites include at least one non-singleton. We focus on non-singleton third-party scripts in our analysis, since singletons are not included across websites. In the following, when we mention third-party scripts, we mean non-singletons as they are widely included, unless otherwise specified.

D. Scale Metrics based on Tree Structures

We present measurement results on scale metrics based on proposed tree structures to demonstrate inclusion relationships between scripts and websites.

1) Third-party JavaScript Deployment: We first investigate the third-party JavaScript deployment among websites.

• *Number of scripts included in a given website:* This metric is derived from the JavaScript Inclusion Tree and measures the number of third-party scripts included in a given website. Intuitively, a website including more third-party scripts exposes a higher security risk to potential attackers. According to our measurement, each website includes 19.29 scripts on average, and 8.62 of them are non-singletons. Figure 5 plots the complementary cumulative distribution function (CCDF) of the number of local scripts, singletons, and non-singletons included in each website. As in Figure 5, local scripts and non-singletons have similar distributions and are widely included, while singletons are less included.

• *Number of websites including a given script:* This metric is calculated from the JavaScript Backtracking Tree and describes the number of websites directly or indirectly including a given third-party script. This metric can indicate the maximum number of potential vulnerable websites if a particular script is manipulated. According to our measurement, a third-party script is included in 25.42 websites on average. Table II demonstrates the top 5 third-party scripts ranked by corresponding numbers of websites including them. As shown, a popular third-party script can be included in around 6% $\sim$ 50% of Alexa top one million websites. (e.g. Google's `analytics.js` is included in 504,692 websites out of the one million websites).

TABLE II: Top 5 most popular third-party scripts.

Rank	Third-party script	Number
1	www.google-analytics.com/analytics.js	504,692
2	pagead2.googlesyndication.com/..._impl.js	127,787
3	connect.facebook.net/en_US/fbevents.js	122,445
4	pagead2.googlesyndication.com/...google.js	120,620
5	pagead2.googlesyndication.com/.../osd.js	115,188

2) Multi-level JavaScript Inclusion: Our measurement results show that indirectly included third-party scripts are common on the Internet. Specifically, according to our measurement, 551,689 websites contain indirectly included third-party scripts. In addition, all third-party scripts have been included for 8,370,582 times in total, and 3,068,041 inclusions among them are indirect. Our study further reveals that 56.84% websites have heights of JavaScript Inclusion Trees larger than 1, and the maximum height of the trees is surprisingly as large as 18. The JavaScript Backtracking Tree tracks all the websites that directly or indirectly include a third-party script. In Figure 6, we plot the CCDF of the number of websites including a script given different heights of JavaScript Backtracking Tree. We can observe that third-party scripts tend to be included in more websites as the height increases.

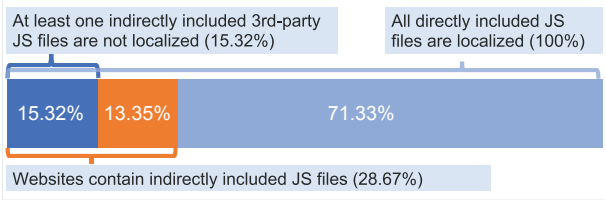

Fig. 7: Statistics about the localization of 3-party JS files.

• **Insecure practice: careless localization of third-party JavaScript files.** Web developers may try to store all included third-party scripts on their local servers to reduce the security risk raised by third-party JavaScript inclusion and caching. During this practice, web developers must track all third-party scripts included in their websites. Nevertheless, we

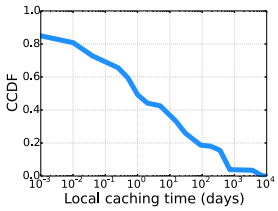

Fig. 8: Local caching time for 3rd-party scripts.

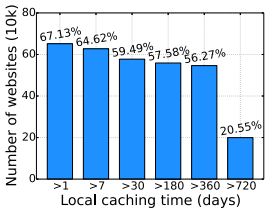

Fig. 9: Num. of webs given local caching time.

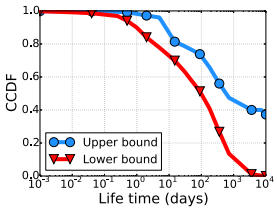

Fig. 10: Upper and lower bounds of life time.

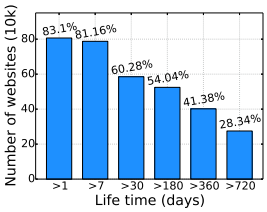

Fig. 11: Num of webs (1 script with life time $>$ a threshold).

observe that a portion of web developers only pay attention to directly included third-party scripts but neglect indirectly included ones. In particular, as shown in Figure 7, we find that 89,723 websites store all their directly included third-party scripts on their local servers, but 13,746 (15.32% of 89,723) of them neglect to store indirectly included third-party scripts locally. This measurement results indicate that multi-level JavaScript inclusion makes the website security management complicated.

E. Time Metrics Measurement

In what follows, we evaluate time metrics based on the HTTP header information of third-party scripts.

1) Local Caching Time: As noted in Section IV-D, local caching time can be measured from the client-side metrics via `max-age` or `Expires`. Figure 8 illustrates the distribution of local caching time of third-party scripts. We find that over 115,517 third-party scripts are set to be cached in a browser for more than 10 days, and over 58,988 are set for more than 100 days. As a result, a malicious third-party script may survive in the cache for quite a long time. Figure 9 plots the number of websites containing at least one third-party script with local caching time in a given interval, and shows that as many as 57.58% websites include at least one script with local caching time larger than 180 days.

• **Insecure practice: inconsistent configurations of Expires and max-age.** Although `max-age` can override `Expires` when both metrics are presented in HTTP headers, they should be set consistent to avoid ambiguity, i.e., local caching time calculated from `Expires` and `max-age` should be equal to each other [12]. According to our results, 168,317 third-party scripts set both `Expires` and `max-age`. Nevertheless, we find that 54,771 scripts have inconsistent `Expires` and `max-age`. Moreover, 40,568 exhibit this difference larger than 50% of the value of `max-age`. These observations imply that web developers may be less aware of the security risk of caching and careless when setting these metrics.

2) Life Time: Next, we proceed to measure the life time of a third-party script, which is defined as the duration a third-party script remains unchanged in the server. We use the estimation method proposed in Section IV-D to bound the life time of a third-party script. Figure 10 plots the CCDF of upper and lower bounds of the life time for different third-party scripts. We see from Figure 10 that at least 241,769 third-party scripts have a life time larger than 10 days, and at least 168,072 have a life time larger than 100 days. We are also interested in the number of websites that include third-party scripts with long life time. Figure 11 shows the number of websites that include at least one third-party script with life time longer than a given threshold. We observe that over half of websites include at least one third-party script, whose life time is longer than half a year. This indicates that a substantial number of websites include third-party scripts with long life time.

• **Insecure practice: poor maintenance of third-party libraries.** A well-maintained third-party script should be updated in a timely manner. Nevertheless, we find that 43,665 third-party scripts have not been updated by third-party providers for at least two years. The possible reason can be that these third-party scripts are developed by unreliable providers or have been rarely maintained.

3) Local Caching Time Versus Life Time: HTTP 1.1 requires that local caching time should be no more than some fraction (the recommended typical setting is 10% [12]) of the life time to ensure a browser can timely retrieve the updated content from a server. We further evaluate the local caching time settings using the ratio of local caching time to the upper bound of life time.

• **Insecure practice: risk-incurring settings of local caching time.** Figure 12 illustrates the distribution of the ratios of all third-party scripts. We observe that 29.23% scripts have ratios larger than 10%, and 10.65% scripts even have ratios larger than 1000%, indicating that their local caching time is improperly set and does not conform the HTTP 1.1 requirement. This further worsens the long-lasting impact of an attack.

F. Joint Analysis of Scale and Time

Previous studies evaluate the adverse impact of third-party JavaScript inclusion and caching from scale and time aspects separately. We wonder whether a widely-included script is also associated with a very long local caching or life time?

Figure 13 box-plots the distribution of the local caching time of different third-party scripts as a function of the number of websites including them (measured from the JavaScript Backtracking Tree). We can see that the local caching time of a third-party script increases when it is included in more websites. For example, for third-party scripts included in more than 10,000 websites, their median local caching time is 365 days (most famous third-party service providers set 1 year as the local caching time, e.g. Google Hosted Libraries [27]).

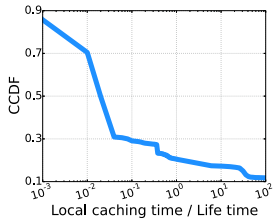

Fig. 12: CCDF of local caching time/life time.

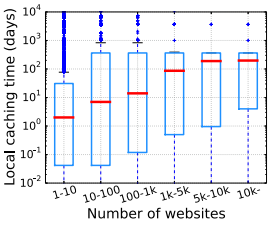

Fig. 13: Local caching time given including websites.

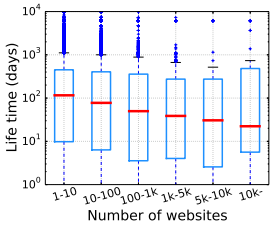

Fig. 14: Lower bounds given including websites.

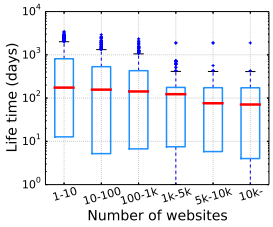

Fig. 15: Upper bounds given including websites.

Figures 14 and 15 box-plot distributions of lower and upper bounds of the life time as a function of the number of websites including them. We can see that lower and upper bounds of the life time of a third-party script both exhibit decreasing trends as it is included in more websites. This shows that popular third-party providers usually update their scripts in a frequent way. Nevertheless, as shown in Figures 14, most third-party scripts still have long life time for more than 50 days.

• **Insecure practice: setting local-caching time not in accordance to life time.** From the security perspective, if a script is updated more frequently (indicating a smaller life time), its local caching time should also be set smaller to mitigate the risk of caching compromised scripts. Unfortunately, the current Internet practice demonstrates the opposite. Comparing Figure 13 with Figures 14 and 15, we find that a more popular script tends to be more frequently updated, but at the same time tends to have a longer local caching time.

VI. Improving Security and Discussions

The measurement results have identified a number of insecure practices that overlook or even increase the security risk of third-party JavaScript inclusion and caching. We discuss potential methods to be used by third-party service providers, websites and browser developers to minimize the adverse impact of third-party JavaScript inclusion and caching.

• *Simplifying Inclusion Relationships and Proper Localization.* We find that third-party JavaScript inclusions are quite complicated and intricate, that substantially complicate the third-party script management and confuse website developers. Third-party JavaScript developers should simplify its JavaScript inclusion relationships as much as possible. It also becomes essential for website developers to regularly update the JavaScript Inclusion Tree and store all third-party scripts at local servers to minimize the security risk.

• *Adequate Configurations of Caching Settings.* Our results reveal that third-party service providers tend to set caching parameters `Expires` and `max-age` with large values. This is both non-HTTP-conforming and insecure. We recommend setting `Expires` and `max-age` according to the 10% recommendation in HTTP 1.1 or even smaller.

• *Isolated Caching.* As discussed, if a widely-included third-party script has been manipulated by an attacker, it may potentially infect all the websites including this malicious script. We suggest that caching scripts for different websites can be isolated from each other, thus containing a potentially malicious script in the scope of a single website. Specifically, developers may make each third-party script dedicated to a particular website. This can be achieved by (i) giving a unique path or file name of the script (by third-party providers) to a particular website, or (ii) appending a query string to the URL of the third-party script.

• *No Caching over Untrustworthy Connections.* A user may face MITM attacks when connecting to an untrustworthy public network. We suggest that a browser may never keep any cached files over the untrustworthy connections, especially when a user visits a website via clicking through an HTTPS certificate warning. The browser can minimize the security risk by disabling caching at the moment that it issues the HTTPS certificate warning. The browser can resume users' caching setting (or even clean all cached contents) when it restarts.

VII. Related Work

• *Malicious JavaScript Injection Strategies.* Our study is motivated by the security concern that when a cached JavaScript file is compromised, access to any website including the same script can be vulnerable. Our work is related to existing works [13]–[17], [28] that detail the attack strategies to compromise a JavaScript file and inject the malicious script into a web browser, such as MITM based manipulations [28], the Pretty-Bad-Proxy attack [13], and modified HTTPS certificate attack [14]. Our research does not focus on designing an attack strategy that has been well studied in the literature; rather, we aim to assess the scope and security indication of third-party JavaScript inclusion and caching on the Internet.

• *Large-scale Measurements for Security Analysis.* Network measurement studies have been conducted in a line of works [29]–[36] to perform the security analysis for the Internet from various perspectives. For example, ZMap [34] developed a fast Internet-wide scanning tool, which has enabled a wide range of security applications, such as botnet analysis [37], domain name system (DNS) manipulation analysis [37], and monitoring of global censorship [38]. The research in this paper focuses on analyzing current practices of third-party JavaScript inclusion and caching, and identifying the security risk. Our research is complementary to these works.

• *Large-scale JavaScript Measurements on the Internet.* Our work is also related to prior measurement studies of client-side JavaScript libraries on the Internet. For example, the work in [39] quantified the reliability of third-party JavaScript providers based on Alexa top ten thousand websites. In [40],

the authors measured the usage of outdated JavaScript libraries over 133,000 websites, and revealed that a large number of websites still use outdated JavaScript libraries. In contrast, the scale of our measurement is larger and we point out the insecure practices about JavaScript inclusion and caching in today's websites and third-party service providers.

VIII. Conclusions

In this paper, we present a comprehensive study to analyze how current practices of third-party JavaScript deployment and caching may exacerbate the adverse impact of malicious scripts. We evaluate the security risk caused by third-party JavaScript inclusion and caching from both scale and time perspectives, and identify insecure practices on the Internet. We also discuss potential solutions to minimize the risk. Our study is instrumental in helping website developers to improve the security perspectives of JavaScript management.

IX. Acknowledgement

The work at the New Mexico State University was supported in part by NSF under grant ECCS-2139028.

References

[1] Adam Barth. *Rfc 6454: The web origin concept*, 2011.

[2] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proc. of NDSS*, volume 2007, page 12, 2007.

[3] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *Proc. of ACM CCS*, pages 1193–1204, 2013.

[4] Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site request forgery. In *Proc. of ACM CCS*, pages 75–88, 2008.

[5] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The unexpected dangers of dynamic JavaScript. In *Proc. of USENIX Security*, pages 723–735, 2015.

[6] Yaoqi Jia, Yue Chen, Xinshu Dong, Prateek Saxena, Jian Mao, and Zhenkai Liang. Man-in-the-browser-cache: Persisting HTTPS attacks via browser cache poisoning. *Computers & Security*, 55:62–80, 2015.

[7] Tobias Lauinger, Abdelberi Chaabane, Ahmet Salih Buyukkayhan, Kaan Onarlioglu, and William Robertson. Game of registrars: An empirical analysis of post-expiration domain name takeovers. In *Proc. of USENIX Security*, pages 865–880, 2017.

[8] Chen Wang, Xiuyuan Zheng, Yingying Jennifer Chen, and Jie Yang. Locating rogue access point using fine-grained channel information. *IEEE Trans. Mobile Computing*, 16(9):2560–2573, 2017.

[9] Amit Levy, Henry Corrigan-Gibbs, and Dan Boneh. Stickler: Defending against malicious content distribution networks in an unmodified browser. *IEEE Security & Privacy*, 14(2):22–28, Mar 2016.

[10] Sean-Philip Oriyano and Robert Shimonski. *Client-side Attacks and Defense*. Newnes, 2012.

[11] *The Chromium Projects*. https://www.chromium.org/chromium-projects.

[12] R Fielding, Mark Nottingham, and J Reschke. Hypertext transfer protocol (HTTP/1.1): Caching. Technical report, 2014.

[13] Shuo Chen, Ziqing Mao, Yi-Min Wang, and Ming Zhang. Pretty-bad-proxy: An overlooked adversary in browsers' HTTPS deployments. In *Proc. of IEEE S&P*, pages 347–359, 2009.

[14] Franco Callegati, Walter Cerroni, and Marco Ramilli. Man-in-the-Middle attack to the HTTPS protocol. *IEEE Security & Privacy*, 7(1):78–81, 2009.

[15] Karthikeyan Bhargavan, Antoine Delignat Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *Proc. of IEEE S&P*, pages 98–113, 2014.

[16] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan, and Marianne Winslett. VEX: Vetting browser extensions for security vulnerabilities. In *Proc. of USENIX Security*, volume 10, pages 339–354, 2010.

[17] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Proc. of NDSS*, 2010.

[18] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Transactions on Internet Technology*, 12, 2012.

[19] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proc. of ACM CCS*, pages 270–283, 2010.

[20] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing your enemy: understanding and detecting malicious web advertising. In *Proc. of ACM CCS*, pages 674–686, 2012.

[21] Kaimin Zhang, Lu Wang, Aimin Pan, and Bin Benjamin Zhu. Smart caching for web browsers. In *Proc. of WWW*, pages 491–500, 2010.

[22] Devdatta Akhawe and Adrienne Porter Felt. Alice in Warningland: A large-scale field study of browser security warning effectiveness. In *Proc. of USENIX Security*, volume 13, 2013.

[23] Rachna Dhamija, J Doug Tygar, and Marti Hearst. Why phishing works. In *Proc. of ACM CHI*, pages 581–590, 2006.

[24] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying wolf: An empirical study of SSL warning effectiveness. In *Proc. of USENIX Security*, pages 399–416, 2009.

[25] Jeremy Clark and Paul C van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *Proc. of IEEE S&P*, pages 511–525, 2013.

[26] *Chrome DevTools Protocol*. https://chromedevtools.github.io/devtools-protocol.

[27] *Google Hosted Libraries*. https://developers.google.com/speed/libraries/.

[28] Roi Saltzman and Adi Sharabani. Active man in the middle attacks. *OWASP AU*, 2009.

[29] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proc. of ACM IMC*, pages 475–488, 2014.

[30] Zakir Durumeric, James Kasten, Michael Bailey, and J Alex Halderman. Analysis of the HTTPS certificate ecosystem. In *Proc. of ACM IMC*, pages 291–304, 2013.

[31] Zain Shamsi, Ankur Nandwani, Derek Leonard, and Dmitri Loguinov. Hershel: single-packet OS fingerprinting. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 195–206, 2014.

[32] Matthew Luckie, Robert Beverly, Tiange Wu, Mark Allman, et al. Resilience of deployed TCP to blind attacks. In *Proc. of ACM IMC*, pages 13–26, 2015.

[33] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *Proc. of ACM IMC*, pages 15–27, 2009.

[34] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. ZMap: Fast internet-wide scanning and its security applications. In *Proc. of USENIX Security*, volume 8, pages 47–53, 2013.

[35] Alan Quach, Zhongjie Wang, and Zhiyun Qian. Investigation of the 2016 linux TCP stack vulnerability at scale. In *Proc. of ACM SIGMETRICS*, 2017.

[36] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai botnet. In *Proc. of USENIX Security*, 2017.

[37] Paul Pearce, Ben Jones, Frank Li, Roya Ensafi, Nick Feamster, Nick Weaver, and Vern Paxson. Global measurement of DNS manipulation. In *Proc. USENIX Security*, 2017.

[38] Paul Pearce, Roya Ensafi, Frank Li, Nick Feamster, and Vern Paxson. Augur: Internet-wide detection of connectivity disruptions. In *Proc. of IEEE S&P*, pages 427–443, 2017.

[39] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *Proc. of ACM CCS*, pages 736–747, 2012.

[40] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. In *Proc. of NDSS*, 2017.