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

Nick Nikiforakis[1], Luca Invernizzi[2], Alexandros Kapravelos[2], Steven Van Acker[1], Wouter Joosen[1], Christopher Kruegel[2], Frank Piessens[1], and Giovanni Vigna[2]

[1]IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium
`firstname.lastname@cs.kuleuven.be`
[2]University of California, Santa Barbara, CA, USA
`{invernizzi,kapravel,chris,vigna}@cs.ucsb.edu`

ABSTRACT

JavaScript is used by web developers to enhance the interactivity of their sites, offload work to the users' browsers and improve their sites' responsiveness and user-friendliness, making web pages feel and behave like traditional desktop applications. An important feature of JavaScript, is the ability to combine multiple libraries from local and remote sources into the same page, under the same namespace. While this enables the creation of more advanced web applications, it also allows for a malicious JavaScript provider to steal data from other scripts and from the page itself. Today, when developers include remote JavaScript libraries, they trust that the remote providers will not abuse the power bestowed upon them.

In this paper, we report on a large-scale crawl of more than three million pages of the top 10,000 Alexa sites, and identify the trust relationships of these sites with their library providers. We show the evolution of JavaScript inclusions over time and develop a set of metrics in order to assess the maintenance-quality of each JavaScript provider, showing that in some cases, top Internet sites trust remote providers that could be successfully compromised by determined attackers and subsequently serve malicious JavaScript. In this process, we identify four, previously unknown, types of vulnerabilities that attackers could use to attack popular web sites. Lastly, we review some proposed ways of protecting a web application from malicious remote scripts and show that some of them may not be as effective as previously thought.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access; H.3.5 [**Online Information Services**]: Web-based services; K.4.4 [**Electronic Commerce**]: Security

1. INTRODUCTION

The web has evolved from static web pages to web applications that dynamically render interactive content tailored to their users. The vast majority of these web applications, such as Facebook and Reddit, also rely on client-side languages to deliver this interactivity. JavaScript has emerged as the de facto standard client-side language, and it is supported by every modern browser.

Modern web applications use JavaScript to extend functionality and enrich user experience. These improvements include tracking statistics (e.g., Google Analytics), interface enhancements (e.g., jQuery), and social integration (e.g., Facebook Connect). Developers can include these external libraries in their web applications in two ways: either (1) by downloading a copy of the library from a third-party vendor and uploading it to their own web server, or (2) by instructing the users' browsers to fetch the code directly from a server operated by a third party (usually the vendor). The safest choice is the former, because the developer has complete control over the code that is served to the users' browsers and can inspect it to verify its proper functionality. However, this choice comes with a higher maintenance cost, as the library must be updated manually. Another downside is that by not including remote code from popular Content Distribution Networks, the developer forces the users' browsers to download scripts from his own servers even if they are identical with scripts that are already available in the browsers' cache. Moreover, this method is ineffective when the library loads additional, remotely-hosted, code at run time (e.g., like Google Analytics does). A developer might avoid these drawbacks by choosing the second option, but this comes at the cost of trusting the provider of the code. In particular, the provider has complete control over the content that is served to the user of the web application. For example, a malicious or compromised provider might deface the site or steal the user's credentials through DOM manipulation or by accessing the application's cookies. This makes the provider of the library an interesting target for cyber-criminals: after compromising the provider, attackers can exploit the trust that the web application is granting to the provider's code to obtain some control over the web application, which might be harder to attack directly. For example, on the 8th of December 2011 the domain distributing qTip2, a popular jQuery plugin, was compromised [2] through a WordPress vulnerability. The qTip2 library was modified, and the malicious version was distributed for 33 days.

It is generally known that developers should include JavaScript only from trustworthy vendors, though it is frightening to imagine the damage attackers could do when compromising a JavaScript vendor such as Google or Facebook. However, there has been no large-scale, in-depth study of how well the most popular web applications implement this policy. In this paper, we study this problem for the 10,000 most popular web sites and web applications (according to Alexa), outlining the trust relationships between these domains and their JavaScript code providers. We assess the maintenance-quality of each provider, i.e., how easy it would be for a determined attacker to compromise the trusted remote host due to its poor security-related maintenance, and we identify weak links that might be targeted to compromise these top domains. We also identify new types of vulnerabilities. The most notable is called "Typosquatting Cross-site Scripting" (TXSS), which occurs when a developer mistypes the address of a library inclusion, allowing an attacker to register the mistyped domain and easily compromise the script-including site. We found several popular domains that are vulnerable to this attack. To demonstrate the impact of this attack, we registered some domain names on which popular sites incorrectly bestowed trust, and recorded the number of users that were exposed to this attack.

The main contributions of this paper are the following:

- We present a detailed analysis of the trust relationships of the top 10,000 Internet domains and their remote JavaScript code providers
- We evaluate the security perimeter of top Internet domains that include code from third-party providers.
- We identify four new attack vectors to which several high traffic web sites are currently vulnerable.
- We study how the top domains have changed their inclusions over the last decade.

The rest of this paper is structured as follows. Section 2 presents the setup and results of our large-scale crawling experiment for the discovery of remote JavaScript inclusions. Section 3 presents the evolution of JavaScript inclusions of popular web sites and our metric for assessing the quality of maintenance of a given JavaScript provider. In Section 4 we introduce four new types of vulnerabilities discovered during our crawl. Section 5 reviews some techniques that web applications can utilize to protect themselves against malicious third-party JavaScript libraries. Section 6 explores the related work and Section 7 concludes.

2. DATA COLLECTION

In this section, we describe the setup and results of our large-scale crawling experiment of the Alexa top 10,000 web sites.

2.1 Discovering remote JavaScript inclusions

We performed a large web crawl in order to gather a large data set of web sites and the remote scripts that they include. Starting with Alexa's list of the top 10,000 Internet web sites [5], we requested and analyzed up to 500 pages from each site. Each set of pages was obtained by querying the Bing search engine for popular pages within each domain. For instance, the search for "site:google.com" will return pages hosted on Google's main domain as well as subdomains. In total, our crawler visited over 3,300,000 pages of top web sites in search for remote JavaScript inclusions. The set of visited pages was smaller than five million since a portion of sites had less than 500 different crawlable pages.

From our preliminary experiments, we realized that simply requesting each page with a simple command-line tool that performs an HTTP request was not sufficient, since in-line JavaScript code can be used to create new, possibly remote, script inclusions. For example, in the following piece of code, the inline JavaScript will create, upon execution, a new remote script inclusion for the popular Google-Analytics JavaScript file:

```
var ishttps = "https:" == document.location.protocol;
var gaJsHost = (ishttps)?
     "https://ssl." : "http://www.");
var rscript = "";
rscript += "\%3Cscript src='" + gaJsHost;
rscript += "google-analytics.com/ga.js' type=";
rscript += "'text/javascript'\%3E\%3C/script\%3E";

document.write(unescape(rscript));
```

To account for dynamically generated scripts, we crawled each page utilizing HtmlUnit, a headless browser [1], which in our experiments pretended to be Mozilla Firefox 3.6. This approach allowed us to fully execute the inline JavaScript code of each page, and thus accurately process all remote script inclusion requests, exactly as they would be processed by a normal Web browser. At the same time, if any of the visited pages, included more remote scripts based on specific non-Firefox user-agents, these inclusions would be missed by our crawler. While in our experiments we did not account for such behaviour, such a crawler could be implemented either by fetching and executing each page with multiple user-agents and JavaScript environments, or using a system like Rozzle [14] which explores multiple execution paths within a single execution in order to uncover environment-specific malware.

2.2 Crawling Results

2.2.1 *Number of remote inclusions*

The results of our large-scale crawling of the top 10,000 Internet web sites are the following: From 3,300,000 pages, we extracted 8,439,799 inclusions. These inclusions map to 301,968 unique URLs of remote JavaScript files. This number does not include requests for external JavaScript files located on the same domain as the page requesting them. 88.45% of the Alexa top 10,000 web sites included at least one remote JavaScript library. The inclusions were requesting JavaScript from a total of 20,225 uniquely-addressed remote hosts (fully qualified domain names and IP addresses), with an average of 417 inclusions per remote host. Figure 1 shows the number of unique remote hosts that the top Internet sites trust for remote script inclusions. While the majority of sites trusts only a small number of remote hosts, the long-tailed graph shows that there are sites in the top

[1]HtmlUnit-`http://htmlunit.sourceforge.net`

Offered service	**JavaScript file**	**% Top Alexa**
Web analytics	`www.google-analytics.com/ga.js`	68.37%
Dynamic Ads	`pagead2.googlesyndication.com/pagead/show_ads.js`	23.87%
Web analytics	`www.google-analytics.com/urchin.js`	17.32%
Social Networking	`connect.facebook.net/en_us/all.js`	16.82%
Social Networking	`platform.twitter.com/widgets.js`	13.87%
Social Networking & Web analytics	`s7.addthis.com/js/250/addthis_widget.js`	12.68%
Web analytics & Tracking	`edge.quantserve.com/quant.js`	11.98%
Market Research	`b.scorecardresearch.com/beacon.js`	10.45%
Google Helper Functions	`www.google.com/jsapi`	10.14%
Web analytics	`ssl.google-analytics.com/ga.js`	10.12%

Table 1: The ten most popular remotely-included files by the Alexa top 10,000 Internet web-sites

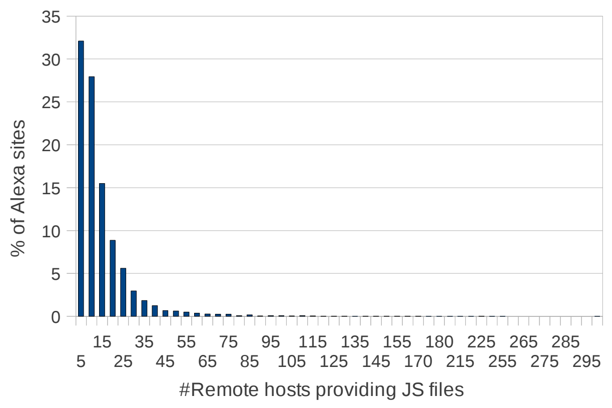

Figure 1: Relative frequency distribution of the percentage of top Alexa sites and the number of unique remote hosts from which they request JavaScript code

Alexa list that trust up to 295 remote hosts. Since a single compromised remote host is sufficient for the injection of malicious JavaScript code, the fact that some popular sites trust hundreds of different remote servers for JavaScript is worrisome.

2.2.2 *Remote IP address Inclusions*

From the total of 8,439,799 inclusions, we discovered that 23,063 (0.27%) were requests for a JavaScript script, where the URL did not contain a domain name but directly a remote IP address. These requests were addressing a total of 324 unique IP addresses. The number of requesting domains was 299 (2.99% percent of the Alexa top 10,000) revealing that the practice of addressing a remote host by its IP address is not widespread among popular Internet sites.

By geolocating the set of unique IP addresses, we discovered that they were located in 35 different countries. The country with most of these IP addresses is China (35.18%). In addition, by geolocating each domain that included JavaScript from a remote IP address, we recorded only 65 unique cases of cross-country inclusions, where the JavaScript provider and the web application were situated on different countries. This shows that if a web application requests a script directly from a remote host through its IP address, the remote host will most likely be in the same country as itself.

In general, IP-address-based script inclusion can be problematic if the IP addresses of the remote hosts are not statically allocated, forcing the script-including pages to keep track of the remote servers and constantly update their links instead of relying on the DNS protocol.

2.2.3 *Popular JavaScript libraries*

Table 1 presents the ten most included remote JavaScript files along with the services offered by each script and the percentage of the top 10,000 Alexa sites that utilize them. There are several observations that can be made based on this data. First, by grouping JavaScript inclusions by the party that benefits from them, one can observe that 60% of the top JavaScript inclusions do not directly benefit the user. These are JavaScript libraries that offer Web analytics, Market Research, User tracking and Dynamic Ads, none of which has any observable effect in a page's useful content. Inclusions that obviously benefit the user are the ones incorporating social-networking functionality.

At the same time, it is evident that a single company, Google, is responsible for half of the top remotely-included JavaScript files of the Internet. While a complete compromise of this company is improbable, history has shown that it is not impossible [31].

3. CHARACTERIZATION OF JAVASCRIPT PROVIDERS AND INCLUDERS

In this section, we show how the problem of remote JavaScript library inclusion is widespread and underplayed, even by the most popular web applications. First, we observe how the remote inclusions of top Internet sites change over time, seeking to understand whether these sites become more or less exposed to a potential attack that leverages this problem. Then, we study how well library providers are maintaining their hosts, inquiring whether the developers of popular web applications prefer to include JavaScript libraries from well-maintained providers, which should have a lower chance of being compromised, or whether they are not concerned about this issue.

3.1 Evolution of remote JavaScript Inclusions

In the previous section, we examined how popular web sites depend on remote JavaScript resources to enrich their functionality. In this section, we examine the remote JavaScript inclusions from the same web sites in another dimension: time. We have crawled *archive.org* [4] to study how

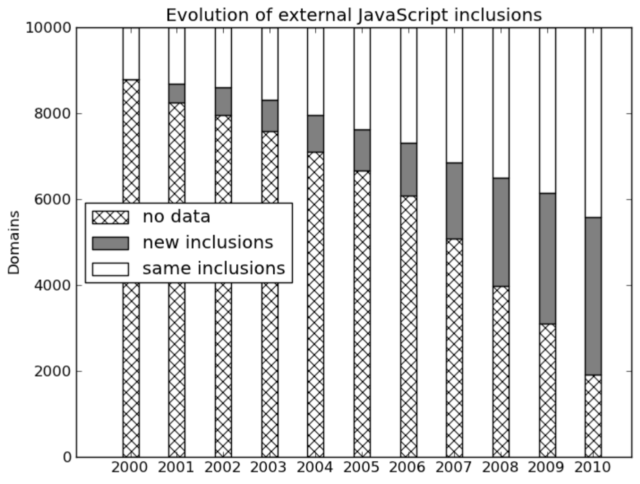

Figure 2: Evolution of remote JavaScript inclusions for domains ranked in the top 10,000 from Alexa.

Year	No data	Same inclusions	New inclusions	% New inclusions
2001	8,256	1,317	427	24.48%
2002	7,952	1,397	651	31.79%
2003	7,576	1,687	737	30.40%
2004	7,100	2,037	863	29.76%
2005	6,672	2,367	961	28.88%
2006	6,073	2,679	1,248	31.78%
2007	5,074	3,136	1,790	36.34%
2008	3,977	3,491	2,532	42.04%
2009	3,111	3,855	3,034	44.04%
2010	1,920	4,407	3,673	45.46%

Table 2: Evolution of the number of domains with same and new remote JavaScript inclusions for the Alexa top 10,000

Year	Unique domains	Total remote inclusions	Average # of new domains
2001	428	1,447	1.41
2002	680	2,392	1.57
2003	759	2,732	1.67
2004	894	3,258	1.67
2005	941	3,576	1.64
2006	974	3,943	1.61
2007	1,168	5,765	1.67
2008	1,513	8,816	1.75
2009	1,728	11,439	1.86
2010	2,249	16,901	2.10

Table 3: Number of new domains that are introduced every year in remote inclusions.

JavaScript inclusions have evolved through time in terms of new remote dependencies and if these increase or decrease over time.

To better understand how JavaScript is included and how the inclusions change over time, we examine each page from different snapshots that span across several years. For the same pages that we crawled in Section 2, we have queried *archive.org* to obtain their versions for past years (if available). For each domain, we choose one representative page that has the most remote inclusions and the highest availability since 2000. For every chosen page we downloaded one snapshot per year from 2000 to 2010. Every snapshot was compared with the previous one in order to compute the inclusion changes.

In Figure 2, one can see the evolution of remote JavaScript inclusions for domains ranked in the top 10,000 from Alexa. For every year, we show how the inclusions from the previous available snapshot changed with the addition of new inclusions or if they remained the same. A *new inclusion* means that the examined domain introduced at least one new remote script inclusion since the last year. If the page's inclusions were the same as the previous year, we consider those as *same inclusion*. Unfortunately, *archive.org* does not cover all the pages we examined completely, and thus we have cases where *no data* could be retrieved for a specific domain for all of the requested years. Also, many popular web sites did not exist 10 years ago. There were 892 domains for which we did not find a single URL that we previously crawled in *archive.org*. A domain might not be found on *archive.org* because of one of the following reasons: the website restricts crawling from its robots.txt file (182 domains), the domain was never chosen to be crawled (320 domains) or the domain was crawled, but not the specific pages that we chose during our first crawl (390 domains). In Table 2, we show how many domains introduced new inclusions in absolute numbers. In our experiment, we find (not surprisingly) that as we get closer in time to the present, *archive.org* has available versions for more of the URLs that we query for and thus we can examine more inclusions. We discovered that every year, a significant amount of inclusions change. Every year there are additional URLs involved in the inclusions of a website compared to the previous years and there is a clear trend of including even more. Back in 2001, 24.48% of the studied domains had at least one new remote inclusion. But as the web evolves and becomes more dynamic, more web sites extend their functionality by including more JavaScript code. In 2010, 45.46% of the examined web sites introduced a new JavaScript inclusion since the last year. This means that almost half of the top 10,000 Alexa domains had at least one new remote JavaScript inclusion in 2010, when compared to 2009.

But introducing a new JavaScript inclusion does not automatically translate to a new dependency from a remote provider. In Table 3, we examine whether more inclusions translate to more top-level remote domains. We calculate the unique domains involved in the inclusions and the total number of remote inclusions. For every page examined, we keep the unique domains involved in its new inclusions, and we provide the average of that number for all available pages per year. There is a clear trend in Table 3 that more inclusions result into more external dependencies from new domains. In fact in 2010 we observed that on average each page expanded their inclusions by including JavaScript from 2.1 new domains on average compared to 2009. This trend shows that the circle of trust for each page is expanding every year and that the surface of attack against them increases.

3.2 Quality of Maintenance Metric

Whenever developers of a web application decide to include a library from a third-party provider, they allow the latter to execute code with the same level of privilege as their own code. Effectively, they are adding the third-party host to the *security perimeter* of the web application, that is the set of the hosts whose exploitation leads to controlling the code running on that web application. Attacking the third-party, and then using that foothold to compromise the web application, might be easier than a direct attack of the latter. The aforementioned incident of the malicious modification of the qTip2 plugin [2], shows that cybercriminals are aware of this and have already used indirect exploitation to infect more hosts and hosts with more secure perimeters.

To better understand how many web applications are exposed to this kind of indirect attack, we aim to identify third-party providers that could be a weak link in the security of popular web applications. To do so, we design a metric that evaluates how well a website is being maintained, and apply it to the web applications running on the hosts of library providers (that is co-located with the JavaScript library that is being remotely included). We indicate the low-scoring as potential weak links, on the assumption that unkempt websites seem easier targets to attackers, and therefore are attacked more often.

Note that this metric aims at characterizing how well websites are maintained, and how security-conscious are their developers and administrators. It is not meant to investigate if a URL could lead to malicious content (a la Google Safebrowsing, for example). Also, we designed this metric to look for the signs of low maintenance that an attacker, scouting for the weakest host to attack, might look for. We recognize that a white-box approach, where we have access to the host under scrutiny, would provide a much more precise metric, but this would require a level of access that attackers usually do not have. We identified the closest prior work in establishing such a metric in SSL Labs's SSL/TLS survey [3] and have included their findings in our metric.

Our *Quality of Maintenance* (QoM) metric is based on the following features:

- **Availability:** If the host has a DNS record associated with it, we check that its registration is not expired. Also, we resolve the host's IP address, and we verify that it is not in the ranges reserved for private networks (e.g., `192.168.0.0/16`). Both of these features are critical, because an attacker could impersonate a domain by either registering the domain name or claiming its IP address. By impersonating a domain, an attacker gains the trust of any web application that includes code hosted on the domain.

- **Cookies:** We check the presence of at least one cookie set as `HttpOnly` and, if SSL/TLS is available, at least one cookie set as `Secure`. Also, we check that at least one cookie has its `Path` and `Expiration` attributes set. All these attributes improve the privacy of session cookies, so they are a good indication that the domain administrators are concerned about security.

- **Anti-XSS and Anti-Clickjacking protocols:** We check for the presence of the `X-XSS-Protection` protocol, which was introduced with Internet Explorer 8 [24] to prevent some categories of Cross-site Scripting (XSS) attacks [18]. Also, we check for the presence of Mozilla's Content Security Policy protocol, which prevents some XSS and Clickjacking attacks [6] in Firefox. Finally, we check for the presence of the `X-Frame-Options` protocol, which aims at preventing ClickJacking attacks and is supported by all major browsers.

- **Cache control:** If SSL/TLS is present, we check if some content is served with the headers `Cache-Control: private` and `Pragma:no-cache`. These headers indicate that the content is sensitive and should not be cached by the browser, so that local attacks are prevented.

- **SSL/TLS implementation:** For a thorough evaluation of the SSL/TLS implementation, we rely on the study conducted by SSL Labs in April 2011. In particular, we check that the domain's certificate is valid (unrevoked, current, unexpired, and matches the domain name) and that it is trusted by all major browsers. Also, we verify that current protocols (e.g, TLS 1.2, SSL 3.0) are implemented, that older ones (e.g., SSL 2.0) are not used, and if the protocols allow weak ciphers. In addition, we check if the implementation is PCI-DSS compliant [12], which is a security standard to which organizations that handle credit card information must comply, and adherence to it is certified yearly by the Payment Card Industry. Also, we check if the domain is vulnerable to the SSL insecure-renegotiation attack. We check if the key is weak due to a small key size, or the Debian OpenSSL flaw. Finally, we check if the site offers Strict Transport Security, which forces a browser to use secure connections only, like HTTPS.

  SSL Labs collected the features described in the previous paragraph nine months before we collected all the remaining features. We believe that this is acceptable, as certificates usually have a lifespan of a few years, and the Payment Card Industry checks PCI-DSS compliance yearly. Also, since these features have been collected in the same period for all the hosts, they do not give unfair advantages to some of them.

- **Outdated web servers:** Attackers can exploit known vulnerabilities in web servers to execute arbitrary code or access sensitive configuration files. For this reason, an obsolete web server is a weak link in the security of a domain. To establish which server versions (in the HTTP `Server` header) should be considered obsolete, we collected these HTTP Server header strings during our crawl and, after clustering them, we selected the most popular web servers and their versions. Consulting change-logs and CVE reports, we compiled a list of stable and up-to-date versions, which is shown in Table 4. While it is technically possible for a web server to report an arbitrary version number, we assume that if the version is modified it will be modified to pretend that the web server is more up-to-date rather than less, since the latter would attract more attacks. This feature is not consulted in the cases where a web server does not send a Server header or specifies it in a generic way (e.g., "Apache").

The next step in building our QoM metric is to weigh these features. We cannot approach this problem from a su-

Web server	Up-to-date version(s)
Apache	1.3.42, 2.0.65, 2.2.22
NGINX	1.1.10, 1.0.9, 0.8.55, 0.7.69, 0.6.39, 0.5.38
IIS	7.5, 7.0
Lighttpd	1.5 , 1.4.29
Zeus	4.3
Cherokee	1.2
CWS	3.0
LiteSpeed	4.1.3
0w	0.8d

Table 4: Up-to-date versions of popular web servers, at the time of our experiment

pervised learning angle because we have no training set: We are not aware of any study that quantifies the QoM of domains on a large scale. Thus, while an automated approach through supervised learning would have been more precise, we had to assign the weights manually. Even so, we can verify that our QoM metric is realistic. To do so, we evaluated with our metric the websites in the following four datasets of domains in the Alexa Top 10,000:

- **XSSed domains:** This dataset contains 1,702 domains that have been exploited through cross-site scripting in the past. That is, an attacker injected malicious JavaScript on at least one page of each domain. Using an XSS exploit, an attacker can steal the cookies or password as it is typed into a login form [18]. Recently, the Apache Foundation disclosed that their servers were attacked via an XSS vulnerability, and the attacker obtained administrative access to several servers [1]. To build this dataset, we used XSSed [29], a publicly available database of over 45,000 reported XSS attacks.

- **Defaced domains:** This dataset contains 888 domains that have been defaced in the past. That is, an attacker changed the content of one or more pages on the domain. To build this dataset, we employed the Zone-H database [32]. This database contains more than six million reports of defacements, however, only 888 out of the 10,000 top Alexa domains have suffered a defacement.

- **Bank domains:** This dataset contains 141 domains belonging to banking institutions (online and brick and mortar) in the US.

- **Random domains:** This dataset contains 4,500 domains, randomly picked, that do not belong to the previous categories.

The cumulative distribution function of the metric on these datasets is shown in Figure 3. At score 60, we have 506 defaced domains, 698 XSSed domains, 765 domains belonging to the random set, and only 5 banks. At score 120, we have all the defaced and XSSed domains, 4,409 domains from the random set, and all but 5 of the banking sites. The maximum score recorded is 160, held by `paypal.com`. According to the metric, sites that have been defaced or XSSed in the past appear to be maintained less than our dataset of random domains. On the other hand, the majority of banking institutions are very concerned with the maintenance of their domains. These findings are reasonable, and empirically demonstrate that our metric is a good indicator of the quality of maintenance of a particular host. This is especially valid also because we will use this metric to classify hosts into three wide categories: high maintenance (metric greater than 150), medium, and low maintenance (metric lower than 70).

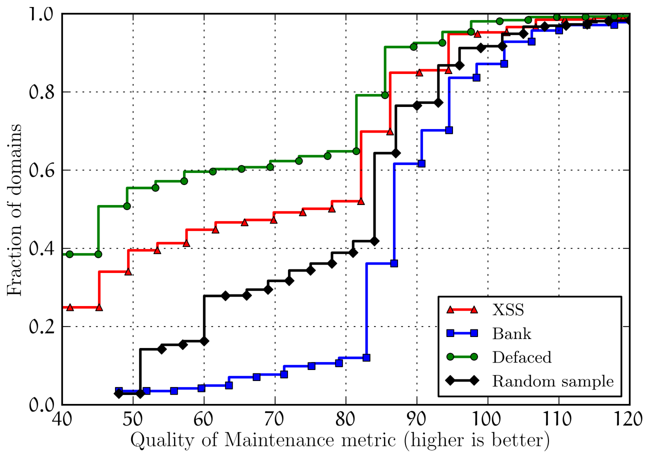

Figure 3: Cumulative distribution function of the maintenance metric, for different datasets

3.3 Risk of Including Third-Party Providers

We applied our QoM metric to the top 10,000 domains in Alexa and the domains providing their JavaScript inclusions. The top-ranking domain is `paypal.com`, which has also always been very concerned with security (e.g., it was one of the proposers of HTTP Strict Transport Security). The worst score goes to `cafemom.com`, because its SSL certificate is not valid for that domain (its `CommonName` is set to `mom.com`), and it is setting cookies non-`HTTPOnly`, and not `Secure`. Interestingly, it is possible to login to the site both in HTTPS, and in plain-text HTTP.

In Figure 4, we show the cumulative distribution function for the inclusions we recorded. We can see that low-maintenance domains often include JavaScript libraries from low-maintenance providers. High-maintenance domains, instead, tend to prefer high-maintenance providers, showing that they are indeed concerned about the providers they include. For instance, we can see that the JavaScript libraries provided by sites with the worst maintenance scores, are included by over 60% of the population of low-maintenance sites, versus less than 12% of the population of sites with high-maintenance scores. While this percentage is five times smaller than the one of low-maintenance sites, still, about one out of four of their inclusions come from providers with a low maintenance score, which are potential "'weak spots"' in their security perimeter. For example, `criteo.com` is an advertising platform that is remotely included in 117 of the top 10,000 Alexa domains, including `ebay.de` and `sisal.it`, the society that holds the state monopoly on bets and lottery in Italy. `criteo.com` has an implementation of SSL that supports weak ciphers, and a weak Diffie-Hellman ephemeral key exchange of 512 bits. Another example is `levexis.com`, a marketing platform, which is included in 15 of the top

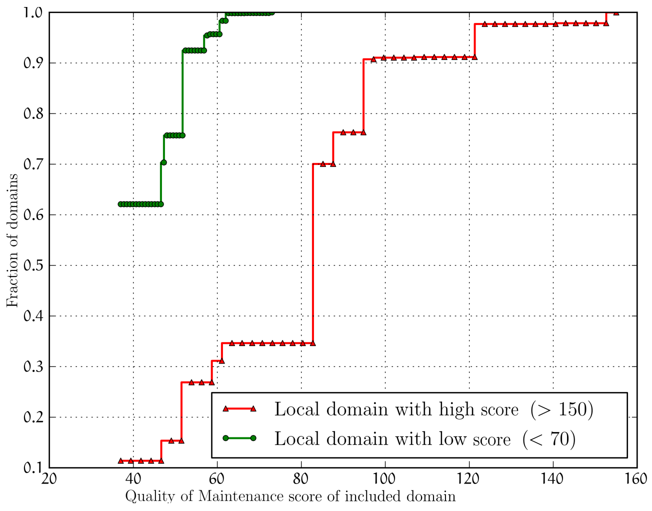

Figure 4: Risk of including third-party providers, included in high and low maintenance web applications.

10,000 Alexa websites, including `lastminute.com`, and has an invalid SSL certificate.

4. ATTACKS

In this section, we describe four types of vulnerabilities that are related to unsafe third-party inclusion practices, which we encountered in the analysis of the top 10,000 Alexa sites. Given the right conditions, these vulnerabilities enable an attacker to take over popular web sites and web applications.

4.1 Cross-user and Cross-network Scripting

In the set of remote script inclusions resulting from our large-scale crawling experiment, we discovered 133 script inclusions where the "src" attribute of the script tag was requesting a JavaScript file from `localhost` or from the `127.0.0.1` IP address. Since JavaScript is a client-side language, when a user's browser encounters such a script tag, it will request the JavaScript file from the user's machine. Interestingly, 131 out of the 133 localhost inclusions specified a port (e.g., `localhost:12345`), which was always greater than 1024 (i.e., a non-privileged port number). This means that, in a multiuser environment, a malicious user can set up a web server, let it listen to high port numbers, and serve malicious JavaScript whenever a script is requested from `localhost`. The high port number is important because it allows a user to attack other users without requiring administrator-level privileges.

In addition to connections to `localhost`, we found several instances where the source of a script tag was pointing to a private IP address (e.g., `192.168.2.2`). If a user visits a site with such a script inclusion, then her browser will search for the JavaScript file on the user's local network. If an attacker manages to get the referenced IP address assigned to his machine, he will be able to serve malicious JavaScript to the victim user.

We believe that both vulnerabilities result from a developer's erroneous understanding of the way in which JavaScript is fetched and executed. The error introduced is not immediately apparent because, often times, these scripts are developed and tested on the developer's local machine (or network), which also hosts the web server.

The set of domains hosting pages vulnerable to cross-user and cross-network scripting, included popular domains such as `virginmobileusa.com`, `akamai.com`, `callofduty.com` and `gc.ca`.

4.2 Stale Domain-name-based Inclusions

Whenever a domain name expires, its owner may choose not to renew it without necessarily broadcasting this decision to the site's user-base. This becomes problematic when such a site is providing remote JavaScript scripts to sites registered under different domains. If the administrators of the including sites do not routinely check their sites for errors, they will not realize that the script-providing site stopped responding. We call these inclusions "stale inclusions". Stale inclusions are a security vulnerability for a site, since an attacker can register the newly-available domain and start providing all stale JavaScript inclusion requests with malicious JavaScript. Since the vulnerable pages already contain the stale script inclusions, an attacker does not need to interact with the victims or convince them to visit a specific page, making the attack equivalent to a stored XSS.

To quantify the existence of stale JavaScript inclusions, we first compiled a list of all JavaScript-providing domains that were discovered through our large-scale crawling experiment. From that list, we first excluded all domains that were part of Alexa's top one million web sites list. The remaining 4,225 domains were queried for their IP address and the ones that did not resolve to an address were recorded. The recorded ones were then queried in an online WHOIS database. When results for a domain were not available, we attempted to register it on a popular domain-name registrar.

The final result of this process was the identification of 56 domain names, used for inclusion in 47 of the top 10,000 Internet web sites, that were, at the time of our experiments, available for registration. By manually reviewing these 56 domain names, we realized that in 6 cases, the developers mistyped the JavaScript-providing domain. These form an interesting security issue, which we consider separately in Section 4.4.

Attackers could register these domains to steal credentials or to serve malware to a large number of users, exploiting the trust that the target web application puts in the hijacked domain. To demonstrate how easy and effective this attack is, we registered two domains that appear as stale inclusions in popular web sites, and make them resolve to our server. We recorded the `Referer`, source IP address, and requested URL for every HTTP request received for 15 days. We minimized the inconvenience that our study might have caused by always replying to HTTP requests with a HTML-only `404 Not Found` error page, with a brief explanation of our experiment and how to contact us. Since our interaction with the users is limited to logging the three aforementioned pieces of data, we believe there are no ethical implications in this experiment. In particular, we registered `blogtools.us`, a domain included on `goldprice.org`, which is a web application that monitors the price of gold and that ranks 4,779th in the US (according to Alexa). Previously, `blogtools.us` was part of a platform to create RSS feeds. We also registered `hbotapadmin.com`, included in a low-traffic page on `hbo.com`, which is an American cable tele-

	blogtools.us	hbotapadmin.com
Visits	80,466	4,615
Including domains	24	4
Including pages	84	41

Table 5: Results from our experiment on expired remotely-included domains

Intended domain	Actual domain
googlesyndication.com	googlesyndicatio.com
purdue.edu	purude.edu
worldofwarcraft.com	worldofwaircraft.com
lesechos.fr	lessechos.fr
onegrp.com	onegrp.nl

Table 6: Examples of mistyped domains found in remote JavaScript inclusion tags

vision network, ranking 1,411th in the US. `hbotapadmin.com` was once owned by the same company, and its registration expired in July 2010. The results of our experiment are shown in Table 5. While `hbotapadmin.com` is being included exclusively by HBO-owned domains, it is interesting to notice that `blogtools.us` is still included by several lower-ranking domains, such as `happysurfer.com`, even though the service is not available anymore.

4.3 Stale IP-address-based Inclusions

As described in Section 2, some administrators choose to include remote scripts by addressing the remote hosts, not through a domain name but directly through an IP address. While at first this decision seems suboptimal, it is as safe as a domain-name-based inclusion, as long as the IP address of the remote machine is static or the including page is automatically updated whenever the IP address of the remote server changes.

To assess whether one of these two conditions hold, we manually visited all 299 pages performing an IP address-based inclusion, three months after our initial crawl. In the majority of cases, we recorded one of the following three scenarios: a) the same scripts were included, but the host was now addressed through a domain name, b) the IP addresses had changed or the inclusions were removed or c) the IP addresses remained static. Unfortunately, in the last category, we found a total of 39 IP addresses (13.04%) that had not changed since our original crawl but at the same time, were not providing any JavaScript files to the requests. Even worse, for 35 of them (89.74%) we recorded a "Connection Timeout," attesting to the fact that there was not even a Web server available on the remote hosts. This fact reveals that the remote host providing the original scripts either became unavailable or changed its IP address, without an equivalent change in the including pages.

As in domain-name-based stale inclusions, these inclusions can be exploited by an attacker who manages to obtain the appropriate IP address. While this is definitely harder than registering a domain-name, it is still a vulnerability that could be exploited given an appropriate network configuration and possibly the use of the address as part of a DHCP address pool.

4.4 Typosquatting Cross-site Scripting (TXSS)

Typosquatting [17, 28] is the practice of registering domain names that are slight variations of the domains associated with popular web sites. For instance, an individual could register `wikiepdia.org` with the intent of capturing a part of the traffic originally meant to go toward the popular Wikipedia website. The user that mistypes Wikipedia, instead of getting a "Server not found" error, will now get a page that is under the control of the owner of the mistyped domain. The resulting page could be used for advertising, brand wars, phishing credentials, or triggering a drive-by download exploit against a vulnerable browser.

Traditionally, typosquatting always refers to a user mistyping a URL in her browser's address bar. However, web developers are also humans and can thus mistype a URL when typing it into their HTML pages or JavaScript code. Unfortunately, the damage of these mistakes is much greater than in the previous case, since every user visiting the page containing the typo will be exposed to data originating from the mistyped domain. In Table 6, we provide five examples of mistyped URLs found during our experiment for which we could identify the intended domain.

As in the case of stale domain-names, an attacker can simply register these sites and provide malicious JavaScript to all unintended requests. We observed this attack in the wild: according to Google's Safe Browsing, `worldofwaircraft.com` has spread malware in January 2012. To prove the efficacy of this attack, we registered `googlesyndicatio.com` (mistyped `googlesyndication.com`), and logged the incoming traffic. We found this domain because it is included in `leonardo.it`, an Italian online newspaper (Alexa global rank: 1,883, Italian rank: 56). Over the course of 15 days, we recorded 163,188 unique visitors. Interestingly, we discovered that this misspelling is widespread: we had visitors incoming from 1,185 different domains, for a total of 21,830 pages including this domain. 552 of the domains that include ours belong to blogs hosted on `*.blogspot.com.br`, and come from the same snippet of code: It seems that bloggers copied that code from one another. This mistype is also long living: We located a page containing the error, `http://www.oocities.org/br/dicas.html/`, that is a mirror of a Brazilian Geocities site made in October 2009.

5. COUNTERMEASURES

In this section, we review two techniques that a web application can utilize to protect itself from malicious remotely-included scripts. Specifically, we examine the effectiveness of using a coarse-grained JavaScript sandboxing system and the option of creating local copies of remote JavaScript libraries.

5.1 Sandboxing remote scripts

Recognizing the danger of including a remote script, researchers have proposed a plethora of client-side and server-side systems that aim to limit the functionality of remotely-included JavaScript libraries (see Section 6). The majority of these countermeasures apply the principle of least privilege to remotely-included JavaScript code. More precisely, these systems attempt to limit the actions that can be performed by a remotely-included script to the bare minimum.

The least-privilege technique requires, for each remotely-

JS Action	**# of Top scripts**
Reading Cookies	41
`document.write()`	36
Writing Cookies	30
`eval()`	28
`XHR`	14
Accessing LocalStorage	3
Accessing SessionStorage	0
Geolocation	0

Table 7: JavaScript functionality used by the 100 most popularly included remote JavaScript files

included JavaScript file, a profile describing which functionality is needed when the script is executed. This profile can be generated either through manual code inspection or by first allowing the included script to execute and then recording all functions and properties of the Document Object Model (DOM) and Browser Object Model (BOM) that the script accessed. Depending on the sandboxing mechanism, these profiles can be either coarse-grained or fine-grained.

In a coarse-grained sandboxing system, the profile-writer instructs the sandbox to either forbid or give full access to any given resource, such as forbidding a script to use `eval`. Constrastingly, in a fine-grained sandboxing system, the profile-writer is able to instruct the sandbox to give access to only parts of resources to a remotely included script. For instance, using ConScript [16], a profile-writer can allow the dynamic creation of all types of elements except iframes, or allow the use of `eval` but only for the unpacking of JSON data. While this approach provides significantly more control over each script than a coarse-grained profile, it also requires more effort to describe correct and exact profiles. Moreover, each profile would need to be updated, every time that a remote script *legitimately* changes in a way that affects its current profile.

Static and dynamic analysis have been proposed as ways of automatically constructing profiles for sandboxing systems, however, they both have limitations in the coverage and correctness of the profiles that they can create. Static analysis cannot account for dynamically-loaded content, and dynamic analysis cannot account for code paths that were not followed in the training phase of the analysis. Moreover, even assuming a perfect code-coverage during training, it is non-trivial to automatically identify the particular use of each requested resource in order to transit from coarse-grained sandboxing to fine-grained.

Given this complex, error-prone and time-consuming nature of constructing fine-grained profiles, we wanted to assess whether coarse-grained profiles would sufficiently constrain popular scripts. To this end, we automatically generated profiles for the 100 most included JavaScript files, discovered through our crawl. If the privileges/resources required by legitimate scripts include everything that an attacker needs to launch an attack, then a coarse-grained sandboxing mechanism would not be an effective solution.

The actions performed by an included JavaScript file were discovered using the following setup: A proxy was placed in between a browser and the Internet. All traffic from the web browser was routed through the web proxy [11], which we modified to intercept HTTP traffic and inject instrumentation code into the passing HTML pages. This instrumentation code uses JavaScript's `setters` and `getters` to add wrappers to certain sensitive JavaScript functions and DOM/BOM properties, allowing us to monitor their use. The browser-provided on-demand stack-tracing functionality, allowed us to determine, at the time of execution of our wrappers, the chain of function calls that resulted in a specific access of a monitored resource. If a function, executed by a remote script, was part of this chain, then we safely deduce that the script was responsible for the activity, either by directly accessing our monitored resources or by assisting the access of other scripts.

For instance, suppose that a web page loads `a.js` and `b.js` as follows:

```
/* a.js */
function myalert(msg) {
    window.alert(msg);
}
```

```
/* b.js */
myalert("hello");
```

```
/* stack trace */
b.js:1:myalert(...)
a.js:2:window.alert(...)
```

In `a.js`, a function `myalert` is defined, which passes its arguments to the `window.alert()` function. Suppose `b.js` then calls `myalert()`. At the time this function is executed, the *wrapped* `window.alert()` function is executed. At this point, the stack trace contains both `a.js` and `b.js`, indicating that both are involved in the call to `window.alert()` (a potentially-sensitive function) and thus both can be held responsible. These accesses can be straightforwardly transformed into profiles, which can then be utilized by coarse-grained sandboxing systems.

Using the aforementioned setup, we visited web pages that included the top 100 most-included JavaScript files and monitored the access to sensitive JavaScript methods, DOM/BOM properties. The results of this experiment, presented in Table 7, indicate that the bulk of the most included JavaScript files read and write cookies, make calls to `document.write()`, and dynamically evaluate code from strings. Newer APIs on the other hand, like `localStorage`, `sessionStorage` and `Geolocation`, are hardly ever used, most likely due to their relatively recent implementation in modern web browsers.

The results show that, for a large part of the included scripts, it would be impossible for a coarse-grained sandboxing system to differentiate between benign and malicious scripts solely on their usage of cookie functionality. For instance, a remotely-included benign script that needs to access cookies to read and write identifiers for user-tracking can be substituted for a malicious script that leaks the including site's session identifiers. Both of these scripts access the same set of resources, yet the second one has the possibility of fully compromising the script-including site. It is also important to note that, due to the use of dynamic analysis and the fact that some code-paths of the executed scripts may not have been followed, our results are lower bounds of the scripts' access to resources, i.e., the tracked scripts may need access to more resources to fully execute.

Overall, our results highlight the fact that even in the presence of a coarse-grained sandboxing system that forbids unexpected accesses to JavaScript and browser resources, an attacker could still abuse the access already white-listed in the attacked script's profile. This means that regardless of their complexity, fine-grained profiles would be required in the majority of cases. We believe that this result motivates further research in fine-grained sandboxing and specifically in the automatic generation of correct script profiles.

5.2 Using local copies

Another way that web sites can avoid the risk of malicious script inclusions is by simply not including any remote scripts. To this end, a site needs to create local copies of remote JavaScript resources and then use these copies in their script inclusions. The creation of a local copy separates the security of the remote site from the script-including one, allowing the latter to be unaffected by a future compromise of the former. At the same time, however, this shifts the burden of updates to the developer of the script-including site who must verify and create a new local copy of the remote JavaScript library whenever a new version is made available.

To quantify the overhead of this manual procedure on the developer of a script-including web application, we decided to track the updates of the top 1,000 most-included scripts over the period of one week. This experiment was conducted four months after our large-scale crawling experiment, thus some URLs were no longer pointing to valid scripts. More precisely, from the top 1,000 scripts we were able to successfully download 803. We started by downloading this set three consecutive times within the same hour and comparing the three versions of each script. If a downloaded script was different all three times then we assume that the changes are not due to actual updates of the library, such as bug fixes or the addition of new functionality, but due to the embedding of constantly-changing data, such as random tokens, dates, and execution times. From this experiment, we found that 3.99% of our set of JavaScript scripts, seem to embed such data and thus appear to be constantly modified. For the rest of the experiment, we stopped tracking these scripts and focused on the ones that were identical all three times.

Over a period of one week, 10.21% of the monitored scripts were modified. From the modified scripts, 6.97% were modified once, 1.86% were modified twice, and 1.36% were modified three or more times. This shows that while some scripts undergo modifications more than once a week, 96.76% are modified at most once. We believe that the weekly manual inspection of a script's modified code is an acceptable tradeoff between increased maintenance time and increased security of the script-including web application. At the same time, a developer who currently utilizes frequently-modified remote JavaScript libraries, might consider substituting these libraries for others of comparable functionality and less frequent modifications.

6. RELATED WORK

Measurement Studies.

To the best of our knowledge, there has been no study of remote JavaScript inclusions and their implications that is of comparable breadth to our work. Yue and Wang conducted the first measurement study of insecure JavaScript practices on the web [30]. Using a set of 6,805 homepages of popular sites, they counted the sites that include remote JavaScript files, use the `eval` function, and add more information to the DOM of a page using `document.write`. Contrastingly, in our study, we crawled more than 3 million pages of the top 10,000 popular web sites, allowing us to capture five hundred times more inclusions and record behavior that is not necessarily present on a site's homepage. Moreover, instead of treating all remote inclusions as uniformly dangerous, we attempt to characterize the quality of their providers so that more trustworthy JavaScript providers can be utilized when a remote inclusion is unavoidable.

Richards et al. [23] and Ratanaworabhan et al. [20] study the dynamic behavior of popular JavaScript libraries and compare their findings with common usage assumptions of the JavaScript language and the functionality tested by common benchmarks. However, this is done without particular focus on the security features of the language. Richarts et al. [22] have also separately studied the use of `eval` in popular web sites.

Ocariza et al. [13] performed an empirical study of JavaScript errors in the top 100 Alexa sites. Seeking to quantify the reliability of JavaScript code in popular web applications, they recorded errors falling into four categories: "Permission Denied," "Null Exception," "Undefined Symbol" and "Syntax Error." Additionally, the authors showed that in some cases the errors were non-deterministic and depended on factors such as the speed of a user's interaction with the web application. The authors did not encounter any of the new types of vulnerabilities we described in Section 4, probably due to the limited size of their study.

Limiting available JavaScript functionality.

Based on the characterization of used functionality, included JavaScript files could be executed in a restricted environment that only offers the required subset of functionality. As we showed in Section 5.1, a fine-grained sandboxing system is necessary because of the inability of coarse-grained sandboxes to differentiate between legitimate and malicious access to resources.

BrowserShield [21] is a server-side rewriting technique that replaces certain JavaScript functions to use safe equivalents. These safe equivalents are implemented in the "bshield" object that is introduced through the BrowserShield JavaScript libraries and injected into each page. BrowserShield makes use of a proxy to inject its code into a web page. Self-protecting JavaScript [19, 15] is a client-side wrapping technique that applies wrappers around JavaScript functions, without requiring any browser modifications. The wrapping code and policies are provided by the server and are executed first, ensuring a clean environment to start from.

ConScript [16] allows the enforcement of fine-grained security policies for JavaScript in the browser. The approach is similar to self-protecting JavaScript, except that ConScript modifies the browser to ensure that an attacker cannot abuse the browser-specific DOM implementation to find an unprotected access path. WebJail [27] is a client-side security architecture that enforces secure composition policies specified by a web-mashup integrator on third-party web-mashup components. Inspired by ConScript, WebJail modifies the Mozilla Firefox browser and JavaScript engine, to enforce these secure composition policies inside the browser. The new "sandbox" attribute of the iframe element in HTML5 [10] provides a way to limit JavaScript functionality, but it is very coarse-grained. It only supports limited restrictions, and as far as JavaScript APIs are concerned, it only supports to completely enable or disable JavaScript.

ADJail [26] is geared toward securely isolating ads from a hosting page for confidentiality and integrity purposes, while maintaining usability. The ad is loaded on a shadow page that contains only those elements of the hosting page to which the web developer wishes the ad to have access. Changes to the shadow page are replicated to the hosting page if those changes conform to the specified policy. Likewise, user actions on the hosting page are mimicked to the shadow page if allowed by the policy.

FlowFox [7] uses the related technique of secure multi-execution [8] to execute arbitrary included scripts with strong guarantees that these scripts can not break a specified confidentiality policy.

Content Security Policy (CSP) [25] is a mechanism that allows web application developers to specify from which locations their web application is allowed to load additional resources. Using CSP, a web application could be limited to only load JavaScript files from a specific set of third-party locations. In the case of typos in the URL, a CSP policy not containing that same typo will prevent a JavaScript file from being loaded from that mistyped URL. Cases where a JavaScript-hosting site has been compromised and is serving malicious JavaScript however, will not be stopped by CSP.

AdSentry [9] is a confinement solution for JavaScript-based advertisement scripts. It consists of a shadow JavaScript engine which is used to execute untrusted JavaScript advertisements. Instead of having direct access to the DOM and sensitive functions, access from the shadow JavaScript engine is mediated by an access control policy enforcement subsystem.

7. CONCLUSION

Web sites that include JavaScript from remote sources in different administrative domains open themselves to attacks in which malicious JavaScript is sent to unsuspecting users, possibly with severe consequences. In this paper, we extensively evaluated the JavaScript remote inclusion phenomenon, analyzing it from different points of view. We first determined how common it is to include remote JavaScript code among the most popular web sites on the Internet. We then provided an empirical evaluation of the quality-of-maintenance of these "code providers," according to a number of indicators. The results of our experiments show that indeed there is a considerable number of high-profile web sites that include JavaScript code from external sources that are not taking all the necessary security-related precautions and thus could be compromised by a determined attacker. As a by-product of our experiments, we identified several attacks that can be carried out by exploiting failures in the configuration and provision of JavaScript code inclusions. Our findings shed some light into the JavaScript code provider infrastructure and the risks associated with trusting external parties in implementing web applications.

Acknowledgments: We want to thank our shepherd Ben Livshits and the anonymous reviewers for their valuable comments. For KU Leuven, this research was done with the financial support from the Prevention against Crime Programme of the European Union, the IBBT, the Research Fund KU Leuven, and the EU-funded FP7 projects NESSoS and WebSand. For UCSB, this work was supported by the Office of Naval Research (ONR) under Grant N000140911042, the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537.

8. REFERENCES

[1] Apache.org. `https://blogs.apache.org/infra/entry/apache_org_04_09_2010`.

[2] Qtip compromised. `https://github.com/Craga89/qTip2/issues/286`.

[3] SSL Labs Server Rating Guide. `https://www.ssllabs.com/downloads/SSL_Server_Rating_Guide_2009.pdf`.

[4] Wayback Machine. `http://archive.org`.

[5] Alexa - Top sites on the Web. `http://www.alexa.com/topsites`.

[6] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 135–144, 2010.

[7] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[8] D. Devriese and F. Piessens. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[9] X. Dong, M. Tran, Z. Liang, and X. Jiang. AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 297–306, New York, NY, USA, 2011. ACM.

[10] I. Hickson and D. Hyatt. HTML 5 Working Draft - The sandbox Attribute. `http://www.w3.org/TR/html5/the-iframe-element.html#attr-iframe-sandbox`, June 2010.

[11] S. Hisao. Tiny HTTP Proxy in Python. `http://www.okisoft.co.jp/esc/python/proxy/`.

[12] P. C. Industry. (Approved Scanning Vendor) Program Guide. `https://www.pcisecuritystandards.org/pdfs/asv_program_guide_v1.0.pdf`.

[13] F. O. Jr., K. Pattabiraman, and B. Zorn. Javascript errors in the wild: An empirical study. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 100 –109, 2011.

[14] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *IEEE Symposium on Security and Privacy*, May 2012.

[15] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In *The 15th Nordic Conf. in Secure IT Systems. Springer Verlag*, 2010.

[16] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[17] T. Moore and B. Edelman. Measuring the perpetrators and funders of typosquatting. In *Proceedings of the 14th international conference on Financial Cryptography and Data Security*, FC'10, pages 175–191, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] OWASP. "cross-site scripting (xss)". `https://www.owasp.org/index.php/XSS`.

[19] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[20] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[21] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *OSDI '06: Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[22] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[24] D. Ross. IE8 Security Part IV: The XSS Filter. `http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx`.

[25] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.

[26] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the 19th USENIX Security Symposium*, Aug. 2010.

[27] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. Webjail: least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 307–316, New York, NY, USA, 2011. ACM.

[28] Y.-M. Wang, D. Beck, J. Wang, C. Verbowski, and B. Daniels. Strider typo-patrol: discovery and analysis of systematic typo-squatting. In *Proceedings of the 2nd conference on Steps to Reducing Unwanted Traffic on the Internet - Volume 2*, SRUTI'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.

[29] XSSed | Cross Site Scripting (XSS) attacks information and archive.

[30] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 961–970, New York, NY, USA, 2009. ACM.

[31] K. Zetter. Google Hack Attack Was Ultra Sophisticated, New Details Show. `http://www.wired.com/threatlevel/2010/01/operation-aurora/`.

[32] Zone-H: Unrestricted information. `http://zone-h.org/`.