Characterizing Insecure JavaScript Practices on the Web

Chuan Yue
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23187, USA
cyue@cs.wm.edu

Haining Wang
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23187, USA
hnw@cs.wm.edu

ABSTRACT

JavaScript is an interpreted programming language most often used for enhancing webpage interactivity and functionality. It has powerful capabilities to interact with webpage documents and browser windows, however, it has also opened the door for many browser-based security attacks. Insecure engineering practices of using JavaScript may not directly lead to security breaches, but they can create new attack vectors and greatly increase the risks of browser-based attacks. In this paper, we present the first measurement study on insecure practices of using JavaScript on the Web. Our focus is on the insecure practices of JavaScript inclusion and dynamic generation, and we examine their severity and nature on 6,805 unique websites. Our measurement results reveal that insecure JavaScript practices are common at various websites: (1) at least 66.4% of the measured websites manifest the insecure practices of including JavaScript files from external domains into the top-level documents of their webpages; (2) over 44.4% of the measured websites use the dangerous eval() function to dynamically generate and execute JavaScript code on their webpages; and (3) in JavaScript dynamic generation, using the document.write() method and the innerHTML property is much more popular than using the relatively secure technique of creating script elements via DOM methods. Our analysis indicates that safe alternatives to these insecure practices exist in common cases and ought to be adopted by website developers and administrators for reducing potential security risks.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; I.7.2 [**Document and Text Processing**]: Document Preparation—*Scripting languages*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*

General Terms

Experimentation, Languages, Measurement, Security

Keywords

JavaScript, execution-based measurement, security, same origin policy, AST tree matching, Web engineering

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

1. INTRODUCTION

Security is an important aspect of Web engineering, and it should be taken into serious consideration in the development of high quality Web-based systems [5, 15, 19, 21, 23]. In many cases, however, security does not receive sufficient attention due to the complexity of Web-based systems, the ad hoc processes of system development, and even the fact that many designers or developers lack security knowledge on Web development techniques. It is not a surprise therefore, that website security breaches are common [7] and Web applications are more susceptible to malicious attacks than traditional computer applications [27].

Browser-based attacks have posed serious threats to the Web in recent years. Exploiting the vulnerabilities in Web browsers [6, 25] or Web applications [11, 14], attackers may directly harm a Web browser's host machine and user through various attacks such as drive-by download [20, 24, 28], cross-site scripting [10, 37], cross-site request forgery [2, 46], and Web privacy attacks [4, 12]. Attackers may even use browsers to indirectly launch large-scale distributed attacks against Web servers [17] or propagate Internet worms [18].

Most of these browser-based attacks are closely tied with JavaScript, which is an interpreted programming language most often used for client-side scripting. JavaScript code embedded or included in HTML pages runs locally in a user's Web browser, and it is mainly used by websites to enhance the interactivity and functionality of their webpages. However, because JavaScript is equipped with a powerful and diverse set of capabilities in Web browsers [9], it has also become the weapon of choice for attackers.

Modern Web browsers impose two restrictions to enforce JavaScript security: the *sandbox* mechanism and the *same-origin* policy. The former limits JavaScript to execute only in a certain environment without risking damage to the rest of the system, while the latter prevents JavaScript in a document of one origin from interacting with another document of a different origin [9, 45]. Unfortunately, most JavaScript-related security vulnerabilities are still the breaches of either of these two restrictions [40]. Some of these vulnerabilities are due to Web browser flaws, but the majority of them have been attributed to the flaws and insecure practices of websites [46, 48].

A great deal of attention has been paid to the JavaScript-related security vulnerabilities such as cross-site scripting [10, 29, 38, 46, 48] that could directly lead to security breaches. However, little attention has been given to websites' insecure practices of using JavaScript on their webpages. Similar to websites' other insecure practices such as using the

customers' social security numbers as their login IDs [8], insecure JavaScript practices may not necessarily result in direct security breaches, but they could definitely cultivate the creation of new attack vectors.

In this paper, we present the first measurement study on insecure practices of using JavaScript at different websites. We mainly focus on two types of insecure practices: *insecure JavaScript inclusion* and *insecure JavaScript dynamic generation*. We define the former as the practices of using the `src` attribute of a <script> tag to directly or indirectly include a JavaScript file from an external domain into the *top-level document* of a webpage. A top-level document is the document loaded from the URL displayed in a Web browser's address bar. By "directly", we mean that the <script> tag belongs to the top-level document, and by "indirectly", we mean that the <script> tag belongs to a sub-level frame or iframe document whose origin is the same as that of the top-level document. We define the latter as the practices of using dangerous techniques such as the eval() function to dynamically generate new scripts. Both types of insecure practices create new vectors for attackers to inject malicious JavaScript code into webpages and launch attacks such as cross-site scripting and cross-site request forgery.

The primary objective of our work is to examine the severity and nature of these two types of insecure JavaScript practices on the Web. To achieve this goal, we devised an execution-based measurement approach. More specifically, we instrumented the Mozilla Firefox 2 Web browser and visited the homepages of 6,805 popular websites in 15 different categories. The instrumented Firefox non-intrusively monitors the JavaScript inclusion and dynamic generation activities on those webpages, and it precisely records important information for offline analysis.

Our measurement results reveal that insecure JavaScript inclusion and dynamic generation practices are widely prevalent among websites. At least 66.4% of the measured websites have the insecure practices of including scripts from external domains into the top-level documents of their homepages. Over 74.9% of the measured websites use one or more types of JavaScript dynamic generation techniques, and insecure practices are quite common. For example, eval() function calls exist at 44.4% of the measured websites. Using the document.write() method and the innerHTML property is much more popular than using the relatively secure method of creating JavaScript elements via DOM (Document Object Model) methods. Our results also show that around 94.9% of the measured websites register various event handlers on their homepages, implying that the captured insecure JavaScript practices in inclusion and dynamic generation are likely conservative estimates.

The main contribution of our paper is threefold. First, we introduce a browser instrumentation framework that enables us to capture essential JavaScript execution behavior on webpages. Not only can this framework measure the insecure JavaScript practices, it can also examine other JavaScript execution characteristics such as function call patterns and code (de)obfuscation activities. Second, we present a classification method to analyze and classify different types of dynamically generated JavaScript code. By extracting the AST (abstract syntax tree) trees of scripts and performing AST signature creation and matching, our classification method can effectively assist us in understanding the structural information of the hundreds of thousands of dynamically generated scripts. Third, our measurement study sheds light on the insecure JavaScript practices and especially reveals the severity of insecure JavaScript inclusion and dynamic generation practices on the Web. Our in-depth analysis further indicates that safe alternatives to these insecure practices do exist in common cases. We therefore suggest website developers and administrators pay serious attention to these insecure engineering practices and use safe alternatives to avoid them.

The rest of the paper is structured as follows. Section 2 explains why the two types of JavaScript practices are insecure. Section 3 introduces our measurement and analysis methodologies. Section 4 describes the data set of this study. Section 5 presents and analyzes our measurement results. Section 6 reviews related work, and finally, Section 7 concludes the paper.

2. MOTIVATION

In the same-origin policy, the origin of a document is defined using the protocol, domain name, and port of the URL from which the document is loaded. It is important to realize that this policy does not limit the origin of a script itself. Although JavaScript code cannot access another document loaded from a different origin, it can fully access the document in which it is embedded or included even when the code has a different origin than the document [9]. Including scripts from an external domain into the top-level document of a webpage is very dangerous because it grants the scripts the maximum permissions allowed to control the webpage and the browser window. Therefore, if the author of a script file or the administrator of a script hosting site is insincere or irresponsible, insecure JavaScript inclusion practices could lead to serious security and privacy breaches. Moreover, script hosting sites could become attractive targets of attacks, especially when their JavaScript files are included by multiple websites. To lower the potential risks, websites should avoid external JavaScript inclusion by using internal JavaScript files from the same sites when possible. Otherwise if external inclusion is really inevitable, for example some advertising sites or traffic analysis sites may necessitate it [22], external included scripts should be retrieved using HTTPS connections and should be restricted within a sub-level HTML frame or iframe document whose origin is different from that of the top-level document.

The eval() function takes a string parameter and evaluates it as JavaScript code. This function is dangerous because it executes the passed script code with the privileges of the function's caller [39]. Therefore, attackers may endeavor to inject malicious code into the evaluated string in order to take advantage of this capability. Meanwhile, since scripts are dynamically generated and evaluated, it is very challenging to effectively filter out maliciously injected code [13, 25, 32]. Eval() should be avoided [1] if at all possible, and its safe alternatives should be used [35, 39]. Other JavaScript dynamic generation techniques such as using the document.write() function and the innerHTML property also pose similar security risks, as discussed in Section 5.

Once attackers have successfully exploited these insecure practices and injected their malicious JavaScript code, they can easily launch severe attacks such as cross-site scripting

[1]Searching "eval is evil" on the Web for many discussions.

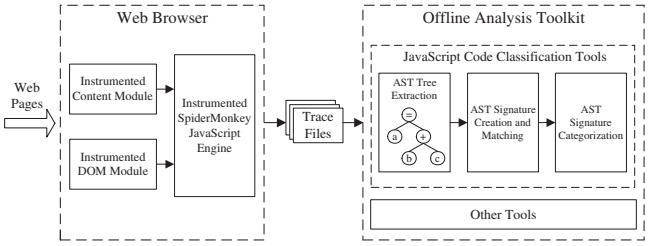

Figure 1: Overview of the Instrumentation Framework and Analysis Toolkit.

and cross-site request forgery. These attacks can be used to conduct many malicious activities such as account hijacking, user behavior tracking, denial of service attacking, and website defacing. Therefore, insecure engineering practices of using JavaScript should be thoroughly investigated, their risks should be highlighted to Web developers, and safe alternatives should be used to avoid them.

3. METHODOLOGY

We devised an execution-based measurement approach to study the insecure JavaScript practices on the Web. Our strategy is to first use an instrumented Web browser to obtain actual JavaScript execution trace information on different webpages, and then use offline analysis to characterize and understand various JavaScript practices. An alternative approach is to simply perform static analysis on webpages. However, this approach suffers from the problem of undecidability and is unable to precisely determine which scripts will be generated and executed. In contrast, our approach allows us to effectively capture the dynamics of webpages and JavaScript code in their real runtime environments. Figure 1 gives an overview of our instrumentation framework and analysis toolkit.

3.1 Instrumentation Framework

To achieve an accurate and efficient measurement, we employed the source code instrumentation technique and instrumented the most popular open source Web browser—Mozilla Firefox. Our instrumentation method is similar to program tracing, which is a well-known approach for monitoring program behavior and measuring program performance. We followed a few rules suggested in [1] to minimize instrumentation overhead. More specifically, we attempted to insert less instrumentation code and place the code only at necessary points with low execution frequency.

We mainly instrumented three modules of Firefox 2 source code: the JavaScript engine, the content module, and the DOM module. Firefox uses SpiderMonkey as its JavaScript engine [47]. SpiderMonkey JavaScript engine is written in C programming language and is a relatively independent module in Firefox. The major interface between SpiderMonkey and other modules in Firefox is the SpiderMonkey JSAPI [41]. JSAPI facilitates other modules in Firefox to use the core JavaScript data types and functions of SpiderMonkey, and it also allows other modules to expose some of their objects and functions to JavaScript code.

Inside the SpiderMonkey, our instrumented code written in C consists of three parts. First, eight trace logging functions were integrated into the JSAPI interface. These functions facilitate the trace collection in a consistent manner, recording various information such as script text, function calls, and event handler registrations. Second, we added code to the byte-code interpreter of SpiderMonkey so that we can record the execution information of any global scripts and function scripts. Third, we instrumented the object system implementation of SpiderMonkey to monitor the calls to the eval() function and collect both the calling context information and the evaluated content information.

The trace files generated in the above instrumentation points enable us to analyze the practices of JavaScript inclusion and the practices of JavaScript dynamic generation using eval(). We also needed to monitor the practices of other JavaScript dynamic generation techniques. Originally we attempted to fulfill this task by still instrumenting inside the SpiderMonkey and monitoring the engine's native callbacks to the content and DOM modules. However, we found that this approach induces high overhead and could only record partial information. Therefore, we decided to directly instrument the content module and the DOM module of Firefox.

In the content module of Firefox, we integrated C++ code to measure the other three types of JavaScript dynamic generation techniques. We instrumented the document.write() method [2] and the method for setting the innerHTML property of an HTML element to track their invocations. Both techniques can be used to add new content to an HTML document, and the added content may contain new JavaScript code. We also added code to monitor the method for replacing, inserting or appending a new DOM element, which could be created by using DOM methods such as document.createElement() and document.createTextNode(). Our instrumentation code can identify the script type of elements and record their source and text information. Other techniques such as the insertAdjacentHTML() method or the outerHTML property are supported in the Internet Explorer Web browser only, and we cannot measure them in Firefox.

In the DOM module and the content module, we added C++ code to measure various event handler registration techniques supported in Firefox. Event handlers can be triggered by user interaction or timer events. We collected event handler registration information to show that further JavaScript inclusion and execution could happen and our captured insecure practices are likely conservative estimates. Event handler registration and other aspects of information described above are written into a set of six different trace files to assist our offline analysis.

Since many internal user interface components of Firefox also heavily use JavaScript, special care is needed to ensure that the above instrumentation code only records the JavaScript execution activities of a visited webpage. Our code checks the JSPrincipals [43] information of an object or script to guarantee this requirement. We also ensured that our instrumentation code only monitors and records essential information and does not change the execution logic of Firefox and SpiderMonkey.

3.2 Analysis Toolkit

We took an offline analysis approach so that we can sufficiently analyze the trace information without interfering with the actual measurement process. We developed an offline analysis toolkit that consists of a set of tools written in approximately 5,000 lines of Java code, 200 lines of C code,

[2]In this paper, it includes the document.writeln() method.

```
SigCreateMatch (XMLfiles, N)
1.  Initialize an empty AST signature set S;
2.  for  each AST tree in the XML files do
3.     thisSig=the top N level structure of the AST tree;
4.     if thisSig matches an existing signature in S then
5.        Record the information of this matching;
6.     else
7.        S = S ∪ {thisSig};
8.     endif
9.  endfor
10. return the result set S;
```

Figure 2: High-level AST Signature Creation and Matching Procedure.

500 lines of Linux shell script code, and 300 lines of Matlab script code. About half of the tools are used for classifying dynamically generated JavaScript code, and the others are used for processing trace records and calculating statistical information. The detailed description of the JavaScript code classification tools is as follows.

The motivation for developing these classification tools is to automate the challenging task of understanding a large number of dynamically generated JavaScript code. To achieve this goal, we explored the concepts in software engineering and developed an AST (abstract syntax tree)-based classification method. As illustrated in Figure 1, the key idea is to first extract the AST trees of scripts, then create and match AST signatures, and finally merge signatures into different categories. We devised such an AST-based approach in that ASTs have been demonstrated effective in program understanding [3, 30].

The AST tree extraction tool is a standalone C program that embeds the SpiderMonkey 1.7 [47]. This is the same version of the SpiderMonkey as used in our instrumented Firefox 2 Web browser. Therefore, our extraction tool can create a token stream and parse the stream into a syntax tree for a script in the same manner as in the instrumented Firefox. The tool finally constructs the essential structure of a syntax tree as an AST tree and writes the tree into an XML file to facilitate further comparison.

We applied *top-down tree matching* techniques to perform AST signature creation and matching, and the high-level procedure is illustrated in Figure 2. First, an empty AST signature set S is initialized. Next, for each AST tree in the XML files, its top N level structure is used to generate an AST signature, denoted as *thisSig*. Then, top-down tree comparisons are made to seek a match between the *thisSig* and an existing signature in the set S. If a match exists, this procedure keeps a record of the related information, otherwise, the *thisSig* is added to the set S as a new AST signature. Finally, this procedure returns the signature set S as its output.

To be accurate and representative, an AST signature keeps the name and type information of an operator node, but it only keeps the type information of an operand. Top-down tree matching techniques can capture the key structural differences between trees, and they have been used in several Web-related projects [26, 33, 34]. The comparison algorithm used in line 4 of this procedure is adapted from the STM (simple tree matching) algorithm presented in [31]. STM is an efficient top-down tree distance comparison algorithm, and our adaptation is to only compare the top N levels of trees. As shown in Section 5, such an adaptation is effective in striking a good balance between retaining the accuracy and reducing the total number of signatures.

Table 1: Category Breakdown by Top-Level Domain.

Category	com	org	gov	net	edu	cc	other	Total
arts	417	16	0	27	1	39	0	500
business	430	7	10	4	0	49	0	500
computers	432	29	1	21	1	15	1	500
games	428	13	0	43	0	14	2	500
health	277	107	41	8	33	30	4	500
home	415	28	22	14	2	18	1	500
news	412	24	6	12	3	43	0	500
recreation	409	19	12	19	0	40	1	500
reference	116	17	11	4	192	158	2	500
regional	292	23	21	6	3	152	3	500
science	209	96	68	8	47	64	8	500
shopping	479	2	0	2	0	17	0	500
society	302	84	34	11	3	58	8	500
sports	403	13	0	21	0	62	1	500
world	199	15	1	23	0	262	0	500
Total	5220	493	227	223	285	1021	31	7500
Uniq-Total	4727	445	170	212	276	950	25	6805

The AST signature categorization tool was developed to further merge AST signatures into different categories. We defined categories according to different types of JavaScript expressions and statements such as arithmetic expressions and assignment statements. Such a categorization can help us to understand the use purposes of JavaScript code from a programming language perspective. This tool is especially useful for analyzing dynamically generated scripts, most of which have specific use purposes in terms of programming language functionality as revealed in our analysis.

4. DATA SET

To obtain a representative data set, we followed a similar method as used in [16] and selected top websites listed by Alexa.com [36]. We chose 15 categories and then top 500 sites from each of these categories. Table 1 gives the breakdown of 15 categories by DNS top-level domain (TLD). Since some sites appear in multiple categories, the total number of unique sites is 6,805 in our study. This number is over five times larger than that in [16], and we also only visited the homepages of those sites so that we can have a consistent measurement. Meanwhile, measuring the insecure JavaScript practices on homepages is sufficient to illustrate the severity of the problem. Table 1 shows that the majority of the 6,805 sites come from the .com TLD and the country code (denoted as the cc) TLD. The former contributes 4,727 unique sites and the latter contributes 950 unique sites.

The execution of JavaScript on a webpage can be roughly divided into two phases: the document loading and parsing phase and the event-driven phase [9]. When the document loading and parsing phase ends, the event-driven phase starts and event handlers can be asynchronously executed in response to various user interaction and timer events. In our study, we developed a browser extension to automatically visit each of the 6,805 webpages using our instrumented Firefox Web browser. On each page, our browser extension waits for the end of the document loading and parsing phase and then stays in the event-driven phase for 10 seconds. Our browser extension has no intention to trigger the execution of any specific event-handlers on a page. This is because the event handlers registered on different webpages are very diverse, and it is difficult to trigger their executions

Table 2: JavaScript Presence by Category and Top Level Domain.

Category/ TLD	Pages with any JS			Pages with DJS
	embedded JS	included JS	Total	
arts	484(96.8%)	483(96.6%)	491(98.2%)	437(87.4%)
business	482(96.4%)	473(94.6%)	492(98.4%)	380(76.0%)
computers	471(94.2%)	465(93.0%)	484(96.8%)	374(74.8%)
games	471(94.2%)	473(94.6%)	488(97.6%)	375(75.0%)
health	467(93.4%)	451(90.2%)	481(96.2%)	330(66.0%)
home	479(95.8%)	471(94.2%)	487(97.4%)	389(77.8%)
news	477(95.4%)	475(95.0%)	483(96.6%)	430(86.0%)
recreation	477(95.4%)	467(93.4%)	487(97.4%)	389(77.8%)
reference	455(91.0%)	443(88.6%)	476(95.2%)	286(57.2%)
regional	479(95.8%)	457(91.4%)	492(98.4%)	401(80.2%)
science	421(84.2%)	405(81.0%)	449(89.8%)	274(54.8%)
shopping	487(97.4%)	486(97.2%)	493(98.6%)	393(78.6%)
society	441(88.2%)	435(87.0%)	466(93.2%)	329(65.8%)
sports	492(98.4%)	482(96.4%)	496(99.2%)	456(91.2%)
world	481(96.2%)	438(87.6%)	489(97.8%)	377(75.4%)
com	4551(96.3%)	4504(95.3%)	4629(97.9%)	3838(81.2%)
org	401(90.1%)	378(84.9%)	422(94.8%)	247(55.5%)
gov	150(88.2%)	137(80.6%)	160(94.1%)	75(44.1%)
net	194(91.5%)	189(89.2%)	204(96.2%)	153(72.2%)
edu	239(86.6%)	223(80.8%)	250(90.6%)	122(44.2%)
cc	863(90.8%)	817(86.0%)	902(94.9%)	654(68.8%)
other	23(92.0%)	22(88.0%)	24(96.0%)	9(36.0%)
All	6421(94.4%)	6270(92.1%)	6591(96.9%)	5098(74.9%)

in a consistent manner. Therefore, the JavaScript execution data set collected in our measurement study covers the whole document loading and parsing phase and 10 seconds of the event-driven phase for each of the 6,805 homepages. The data set was collected in the second week of July 2008.

5. RESULTS AND ANALYSIS

We present and analyze our measurement results in this section. We first briefly present the results on JavaScript presence. Then, we detail the results on the insecure practices of JavaScript inclusion and dynamic generation. Finally, we give a short summary of the results on event handler registrations.

5.1 Overall JavaScript Presence

Table 2 lists the results of overall JavaScript presence for the 6,805 measured homepages. We use *JS* to represent any JavaScript code, and we use *DJS* to represent the JavaScript code that is dynamically generated by using one of the four dynamic generation techniques measured in our instrumented Firefox Web browser. The *embedded JS* indicates that the executed JavaScript code is embedded within an HTML document, and the *included JS* indicates that the executed JavaScript code is included from a separate file.

Overall, JavaScript execution has been widely observed on 6,591(96.9%) homepages. Both the JS embedding and JS inclusion are very common, and they are practiced on 6,421 and 6,270 pages, respectively. The percentage of webpages containing JavaScript execution within a category ranges from 89.8% for science to 99.2% for sports, and the percentage of webpages containing JavaScript execution within a TLD ranges from 90.6% for .edu to 97.9% for .com. JavaScript dynamic generation is also very popular, and there are 5,098 (74.9%) sites containing DJS on their homepages. For the DJS presence within a category, the lowest percentage is 54.8% for science, and the highest percentage is 91.2% for sports. For the DJS presence within a TLD, the highest percentage is 81.2% for .com, and the lowest percentage is 36.0% for `other` domains such as .mil and .info.

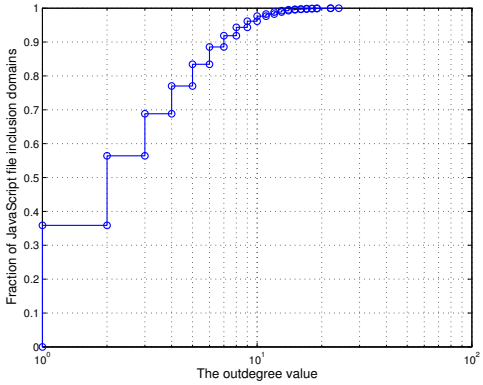

Figure 3: Cumulative distribution of the 4,517 JavaScript file inclusion domains in terms of their outdegree values.

5.2 Insecure JavaScript Inclusion

Among all the 6,270 webpages with the included JS, we identify and analyze insecure practices of JavaScript inclusion. Note that we defined the insecure JavaScript inclusion as the practices of using the `src` attribute of a <script> tag to directly or indirectly include a JavaScript file from an external domain into the top-level document of a webpage. Keeping JavaScript code separate from HTML markups is actually a good engineering practice, advocated especially in the unobtrusive JavaScript programming paradigm [9, 49]. Therefore, there is no need to analyze the good practices of including JavaScript files from the same host or domain, and we only focus on the insecure inclusion practices.

5.2.1 Results and analysis

To our surprise, insecure JavaScript inclusion is very prevalent. Around 66.4% (4,517 out of 6,805) of websites directly or indirectly include JavaScript files from external domains into the top-level documents of their homepages. Note that our analysis tool applies a conservative standard to compare the domain name of a JavaScript file and that of its including homepage. Two domain names are regarded as different only if, after discarding their top-level domain names (e.g., .com) and the leading name "www" (if existing), they do not have any common sub-domain name[3]. Therefore, this 66.4% result is basically an objective estimate of the severity of insecure JavaScript inclusion practices.

After further analyzing the domain name relationship between JavaScript file inclusion sites and JavaScript file hosting sites, we found that those 4,517 sites include JavaScript files from a diverse set of 1,985 external domains. We can use a directed graph to characterize the domain name relationship between these sites. Different vertices represent different domain names, and a direct edge from vertex A to vertex B means that the homepage in domain A includes at least one JavaScript file from domain B. Therefore, 4,517 vertices have a greater than zero outdegree value, and 1,985 vertices have a greater than zero indegree value.

Figure 3 illustrates the CDF (cumulative distribution function) of the 4,517 JavaScript file inclusion domains in terms of their outdegree values. We can see that approximately

[3]For example, two domain names www.d1sub2.d1sub1.d1tld and d2sub3.d2sub2.d2sub1.d2tld are regarded as different only if the intersection of the two sets {d1sub2, d1sub1} and {d2sub3, d2sub2, d2sub1} is empty.

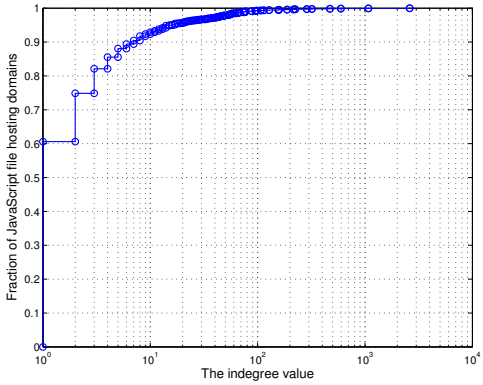

Figure 4: Cumulative distribution of the 1,985 JavaScript file hosting domains in terms of their in-degree values.

43.6% of the 4,517 sites include JavaScript files from at least three external domains. While the mean value of outdegree is 3.1, the maximum value of outdegree reaches 24. These results indicate that not only 66.4% of measured sites are at the risk of having their homepages under the control of the included JavaScript code, but many of them also face higher risks from multiple sources.

From a different perspective, Figure 4 depicts the CDF of the 1,985 JavaScript file hosting domains in terms of their indegree values. We can observe two interesting phenomena. On the one hand, JavaScript files in approximately 60.6% of the hosting domains are only included by one of our visited homepages. On the other hand, JavaScript files in approximately 7.7% of the hosting domains are included by at least 10 of our visited homepages, and JavaScript files in 14 sites are even included by at least 100 of our visited homepages. The mean value of indegree is 7.2, but the maximum value of indegree reaches a very high value of 2,606. After inspecting those 14 high-profile JavaScript file hosting domains and many other low-profile domains, we found that a few of them are popular traffic analysis service sites and advertising servers. However, most of them are the kind of "hidden" sites that provide nothing on their root URLs but just point to some stored JavaScript files using URL paths. Understanding the properties of those sites is beyond the scope of this paper, but what we need to emphasize is that external JavaScript file hosting sites, especially those high-profile ones, create new vectors for large-scale browser-based attacks. Even a single compromised JavaScript file could directly cause security breaches on thousands of websites.

Among the 4,517 sites that include JavaScript files from external domains, we also observed that 125 sites only use the HTTPS protocol to retrieve JavaScript files and 138 sites use both the HTTP protocol and the HTTPS protocol to retrieve different JavaScript files. In total, there are 263 sites using HTTPS to include scripts from 72 JavaScript file hosting sites. These observations imply that some JavaScript file hosting sites do provide the secure transmission service for accessing their hosted JavaScript files, and some of our measured sites do use this service. However, this secure JavaScript transmission service is not popular. Only 3.6% (72 out of 1,985) of the JavaScript file hosting sites provide the service, and only 5.8% (263 out of 4,517) of the JavaScript file inclusion sites use the service. Also note that HTTPS protects data in transit, but it does not guarantee that a JavaScript file is uncompromised in a hosting site.

In contrast to these 4,517 sites, we did find that there are 324 other sites, in which an external included JavaScript file is always restricted within a sub-level HTML frame or iframe document whose origin is different from that of the top-level document. This observation implies that some sites do limit the control of external included JavaScript code within sub-level documents and provide a protection to the top-level documents of their homepages. However, such a relatively secure practice is exclusively followed by only 324 measured sites, and those 4,517 sites still use a very insecure way to include external JavaScript files.

5.2.2 *Safe alternatives to insecure inclusion*

Our results show that insecure JavaScript inclusion is widely practiced by the majority (66.4%) of our measured sites. Our in-depth analysis on the domain name relationship between JavaScript file inclusion sites and hosting sites further reveals the severity and nature of those insecure practices. Although HTTPS and sub-level documents are used by a small portion of sites to enhance the security of external JavaScript file inclusion, we believe that the majority of measured JavaScript file inclusion sites and hosting sites have not paid sufficient attention to the potential risks of insecure JavaScript inclusion. For JavaScript file inclusion sites, we suggest them (1) avoid external JavaScript inclusion by using internal JavaScript files from the same sites, if at all possible; (2) restrict the permission of external included scripts by placing them within a sub-level HTML frame or iframe document whose origin is different from that of the top-level document, if external inclusion is really inevitable; and (3) retrieve external JavaScript files using HTTPS connections, if the HTTPS service is available. The third suggestion needs a hosting site to provide the HTTPS service for accessing its JavaScript files, but the first two suggestions can be easily adopted by JavaScript file inclusion sites.

5.3 Insecure JavaScript Dynamic Generation

Since 74.9% of measured sites (5,098 out of 6,805) contain DJS scripts on their homepages, we now characterize all the DJS scripts based on their generation techniques and analyze insecure practices.

5.3.1 *DJS presence by category and TLD*

Table 3 lists the overall DJS presence by category and TLD for the four different DJS generation techniques. We can see that the eval() function and the document.write() method are widely used on 44.4% and 64.6% of webpages, respectively. In contrast, the innerHTML property and the DOM methods (i.e., replacing, inserting or appending a new created script element) are only used on 13.7% and 11.7% of webpages, respectively. It is also interesting to notice that the categories with the highest DJS presence values are news and sports for all the four generation techniques. The TLDs with the highest DJS presence values are .com, .net, and country code domains. These results indicate that JavaScript dynamic generation is more likely to be used on those sites that have more dynamic contents.

5.3.2 *DJS instance summary*

We now examine the generated DJS instances on each webpage. A DJS instance is identified in different ways for different generation techniques. For the eval() function, the

Table 3: DJS Presence by Category and Top-Level Domain.

Category/ TLD	eval-generated	write-generated	innerHTML-generated	DOM-generated
arts	258(51.6%)	403(80.6%)	76(15.2%)	83(16.6%)
business	253(50.6%)	295(59.0%)	73(14.6%)	56(11.2%)
computers	205(41.0%)	307(61.4%)	55(11.0%)	55(11.0%)
games	203(40.6%)	327(65.4%)	58(11.6%)	57(11.4%)
health	190(38.0%)	276(55.2%)	35(7.0%)	34(6.8%)
home	240(48.0%)	357(71.4%)	57(11.4%)	73(14.6%)
news	314(62.8%)	412(82.4%)	161(32.2%)	110(22.0%)
recreation	229(45.8%)	310(62.0%)	67(13.4%)	57(11.4%)
reference	144(28.8%)	214(42.8%)	44(8.8%)	22(4.4%)
regional	258(51.6%)	337(67.4%)	97(19.4%)	58(11.6%)
science	137(27.4%)	234(46.8%)	39(7.8%)	35(7.0%)
shopping	245(49.0%)	307(61.4%)	37(7.4%)	38(7.6%)
society	163(32.6%)	283(56.6%)	42(8.4%)	42(8.4%)
sports	322(64.4%)	424(84.8%)	114(22.8%)	95(19.0%)
world	212(42.4%)	341(68.2%)	92(18.4%)	55(11.0%)
com	2359(49.9%)	3359(71.1%)	724(15.3%)	656(13.9%)
org	109(24.5%)	195(43.8%)	25(5.6%)	22(4.9%)
gov	32(18.8%)	50(29.4%)	9(5.3%)	9(5.3%)
net	77(36.3%)	135(63.7%)	26(12.3%)	21(9.9%)
edu	50(18.1%)	92(33.3%)	17(6.2%)	7(2.5%)
cc	393(41.4%)	558(58.7%)	130(13.7%)	79(8.3%)
other	4(16.0%)	7(28.0%)	3(12.0%)	0(0.0%)
All	3024(44.4%)	4396(64.6%)	934(13.7%)	794(11.7%)

whole evaluated string content is regarded as a DJS instance. Within the written content of the document.write() method and the value of the innerHTML property, a DJS instance can be identified from three sources: (1) between a pair of <script> and </script> tags; (2) in an event handler specified as the value of an HTML attribute such as onclick or onmouseover; and (3) in a URL that uses the special `javascript:protocol` specifier [9]. For the DOM methods, each new script element is identified as a DJS instance.

Table 4 gives a summary of DJS instances for both the document loading and parsing phase, denoted as the pre-onload phase, and the event-driven phase, denoted as the post-onload phase. The two numbers in each table cell represent the data for the pre-onload and post-onload phases, respectively. The data in the second row of the table gives the total number of DJS instances identified in the two execution phases for the four different techniques. The data in the third row of the table gives the total number of webpages on which those DJS instances are identified. The IPP in the last three rows of the table stands for the "Instance Per Page".

Table 4: DJS Instance Summary for Pre-onload/Post-onload Phases.

Summary	eval-generated	write-generated	innerHTML-generated	DOM-generated
total number of DJS instances	194676/ 22632	67446/ 519	28717/ 6626	1370/ 557
total number of pages	2986/ 363	4385/ 63	844/ 187	680/ 260
mean value of IPP	65.2/ 62.3	15.4/ 8.2	34.0/ 35.4	2.0/ 2.1
maximum value of IPP	2543/ 6350	1053/ 160	5001/ 1403	13/ 25
standard deviation of IPP	174.4/ 367.2	41.6/ 20.9	184.3/ 134.1	1.7/ 2.7

It is evident that the eval() function generates the largest number of DJS instances in both phases (194,676 in the pre-onload phase and 22,632 in the post-onload phase). The mean value of IPP for eval-generated DJS instances is 65.2 in the pre-onload phase and 62.3 in the post-onload phase. The maximum value of IPP for eval-generated DJS instances reaches 2,543 in the pre-onload phase and 6,350 in the post-onload phase. These numbers indicate that eval() may be misused or abused. The document.write() method also generates a large number of DJS instances in the pre-onload phase, but it only generates 519 DJS instances on 63 pages in the post-onload phase. Calling document.write() in post-onload phase is usually not desirable because it will overwrite the current document with the written content. In both phases, the innerHTML property also generates a large number of DJS instances, while DOM methods generate much fewer DJS instances.

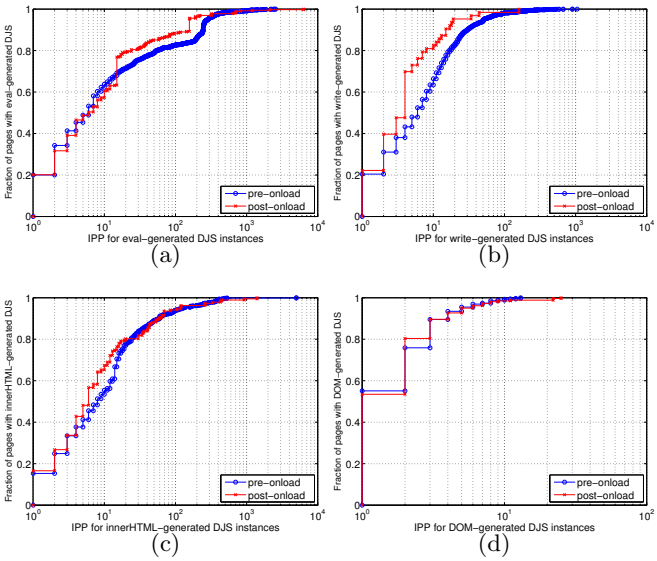

Figure 5: Cumulative distribution of the webpages in terms of IPP (Instance Per Page) for (a) eval-generated, (b) write-generated, (c) innerHTML-generated, and (d) DOM-generated DJS instances.

For the four JavaScript dynamic generation techniques, Figures 5(a) to 5(d) further illustrate the cumulative distribution of the webpages in terms of IPP. In each of these four figures, the "o" curve is for the pre-onload phase and the "∗" curve is for the post-onload phase. Note that the total number of pages is different for the two phases (as shown in the third row of Table 4), and we present the two curves together for ease of comparison. We can see that the indication of misuse or abuse is especially evident for the eval() function. While the majority (about 60%) of webpages have 10 or less eval-generated DJS instances, nearly 17% and 11% of webpages have 100 or more eval-generated DJS instances for the pre-onload phase and the post-onload phase, respectively.

5.3.3 *Structural analysis of eval-generated DJS*

The prevalence of DJS on various categories of webpages and the high IPP values motivate us to further understand the use purposes of the large number of DJS instances. Using our JavaScript code classification tools, we now uncover the use purposes of eval-generated DJS instances in terms of programming language functionality.

From the total 217,308 (both the pre-onload phase and the post-onload phase) eval-generated DJS instances, 217,308

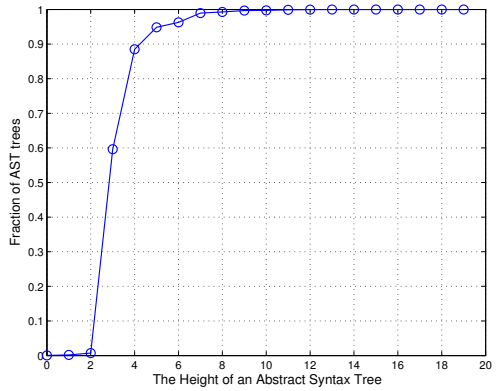

Figure 6: Cumulative distribution of the AST trees in terms of the height of an AST tree.

AST trees are extracted by our AST tree extraction tool. The maximum height of these AST trees is 19. Figure 6 shows the cumulative distribution of the AST trees in terms of the height of an AST tree. Nearly 90% of AST trees have a height less than or equal to 4. Therefore, we selected $N = 4$ as the input parameter (Figure 2) and used the top-four level structure of AST trees to create and match AST signatures. A total number of 647 AST signatures are created and matched from the 217,308 AST trees. These 647 AST signatures capture the essential structural information of the 217,308 AST trees, and they greatly facilitate our further analysis. Using more levels of AST tree structure is unnecessary because lower-level AST tree nodes only contain less important structural information.

Finally, AST signatures with the same programming language functionality are merged into the same category by using our AST signature categorization tool. For example, two AST signatures representing two types of function calls with different number or type of parameters are merged into the same *function calls* category. Table 5 lists the final 17 categories of DJS instances classified from the 647 AST signatures, and in turn from the 217,308 DJS instances.

Table 5: The 17 categories of eval-generated DJS instances.

Category	Presence in Pages	Number of DJS instances	Average DJS length
parse error	20(0.7%)	111(0.05%)	1175.5
empty content	124(4.1%)	291(0.13%)	0.0
simple expression	1209(40.0%)	134251(61.8%)	13.9
arithmetic expression	12(0.4%)	158(0.1%)	67.9
relational expression	79(2.6%)	3246(1.5%)	31.7
logical expression	159(5.3%)	3249(1.5%)	75.8
object/array literal	265(8.8%)	4798(2.2%)	623.0
other expression	126(4.2%)	5789(2.7%)	18.5
variable declarations	98(3.2%)	411(0.2%)	586.0
function declarations	1157(38.3%)	1929(0.9%)	13380.7
assignment statements	1289(42.6%)	42015(19.3%)	51.9
function calls	527(17.4%)	2733(1.3%)	368.9
method calls	561(18.6%)	2062(0.9%)	75.9
object/array creations	29(1.0%)	212(0.1%)	41.3
conditional statements	181(6.0%)	9371(4.3%)	519.1
try-catch statements	1075(35.5%)	4127(1.9%)	51.7
mixed statements	910(30.1%)	2555(1.2%)	6884.1

We can see that 0.05% of the DJS instances have *parse error* when AST trees are extracted, and 0.13% of the DJS instances have *empty content*. The majority (around 98.6%) of the eval-generated DJS instances are classified into the 14 categories from *simple expression* to *try-catch statements*. The DJS instances in these 14 categories all have specific use purposes in terms of programming language functionality. Only 1.2% of the DJS instances have mixed programming language functionalities, and they are classified into the last category of *mixed statements*. The generated DJS instances in the last 15 categories are either various expressions (from *simple expression* to *other expression*) or various statements (from *variable declarations* to *mixed statements*). In general, a JavaScript expression is used only to produce a value, while a JavaScript statement normally has side effects and is often used to accomplish some tasks.

5.3.4 Safe alternatives to eval()

To further understand whether using eval() is necessary in these different categories, we randomly sampled and inspected both the content and the calling context of 700 DJS instances. We sampled 200 DJS instances from the simple expression category and 200 DJS instances from the assignment statements category. These two categories have the largest numbers of DJS instances, accounting for 61.8% and 19.3%, respectively, of all the eval-generated DJS instances. The remaining 300 DJS instances are sampled from the other 15 categories, with each of them contributing 20 instances.

In at least 70% of the sampled cases, the eval() function is misused or abused while safe alternatives can be easily identified. Here we illustrate three representative sampled cases. The first one is: this.homePos = eval("0" + this.dirType + this.dim), in which a string simple expression "0-500" is generated. Indeed, such a kind of string concatenation directly generates a string value, and using eval() is redundant. The second one is: var ff_nav=eval("nav_"+tt[i][1]), in which a variable name "nav_20912" is dynamically accessed. A safe alternative is using the JavaScript *window* object to directly access the variable: var ff_nav=window["nav_"+tt[i][1]]. The third one is: var responses = eval(o.responseText), in which the response content of an XMLHttpRequest [50] is directly evaluated. This practice is used in many of our sampled cases to convert a *responseText* into a JSON object. However, since malicious JavaScript code could be injected into the *responseText*, it would be better to use a JSON parser rather than the eval() function to perform such a transformation [42]. The other 30% of the sampled cases usually have complex calling context, so we do not further identify their safe alternatives.

We suggest that eval() should be avoided if at all possible. In addition to the safe alternatives exemplified above, DOM methods can be generally used to generate and execute various JavaScript statements.

5.3.5 Structural analysis of other types of DJS

As mentioned before, the DJS instances generated by the document.write method() and the innerHTML property are identified from three different sources. We use *jscode* to present a DJS instance identified between a pair of <script> and </script> tags, use *eventhandler* to represent a DJS instance identified in an event handler, and use *jsprotocol* to represent a DJS instance identified in a `javascript:protocol` URL. The DJS instances generated by the DOM methods are specified in either the `src` attribute or the `text` attribute of a script element. Table 6 gives the structural analysis results of the DJS instances generated by these three dynamic generation techniques. The main usage of each type of DJS instance is summarized in the last column of the Table 6.

Table 6: Structural analysis of DJS instances generated by the document.write() method, innerHTML property, and DOM methods.

Technique and Type		Presence in Pages	Number of DJS instances	Avg. length	Main usage
write	jscode	4000	26125	77	JS inclusion
	eventhandler	1773	38650	39	function call
	jsprotocol	501	3190	45	function call
innerHTML	jscode	120	503	262	JS inclusion
	eventhandler	747	31267	60	function call
	jsprotocol	336	3573	33	function call
DOM	src	779	1866	-	JS inclusion
	text	33	61	623	assignment

5.3.6 *Safe alternatives to jscode generation via document.write() and innerHTML*

For the eventhandler and jsprotocol DJS instances generated by document.write() and innerHTML, their usages are relatively safe. When new content is added to a document, event handlers are directly specified on various elements of the newly-added content to respond to various events. The `javascript:protocol` scripts are often used on links to execute some statements without loading a new document.

What we emphasize is that generating jscode using document.write() and innerHTML is not desirable. For document.write(), the generated jscode is immediately executed. Multiple document.write() calls can be used to construct a jscode, and document.write() calls can be nested. All these factors make the filtering of write-generated malicious JavaScript code a very challenging task [32]. However, our results show that 26,125 instances of write-generated jscode are identified on a large number of 4,000 homepages. For innerHTML, the generated jscode is recognized by a browser, but it is not necessarily executed. For example, Firefox does not directly execute a jscode generated by innerHMTL. In Internet Explorer, the `defer` attribute and some tricks need to be used to execute an innerHTML-generated jscode, but this practice is also not recommended due to potential script-injection attacks [44]. Fortunately, only 503 instances of this practice are identified on 120 pages as shown in Table 6.

The best practice is to use DOM methods to dynamically generate JavaScript code. Using DOM methods (such as createElement() and createTextNode()) to create JavaScript elements explicitly declares that the new elements are scripts. This practice can enable potential Web content protection mechanisms such as those presented in [13, 25, 32] to accurately define security policies and weed out potential malicious JavaScript code. Unfortunately, only 1,927(1,866 plus 61) instances of this practice are identified.

Our results show that the main usage of jscode generated by document.write() and innerHTML is for including other JavaScript files (denoted as *JS inclusion* in Table 6). Indeed, by specifying the `src` attribute of a script element, DOM methods fit well for such a usage. By specifying the `text` attribute of a script element, DOM methods can also be used to generate and execute various statements such as assignment statements or function calls, thus safely replacing the relatively insecure practices of jscode generation via document.write() and innerHTML.

5.4 Event Handler Registration

Our measurement results show that event handler registrations occurred on 6,451(94.9%) pages in the pre-onload phase, with an average of 108.2 registrations per page and a maximum of 5,074 registrations per page. Event handler registration occurred on 1,767(26.0%) pages in the post-onload phase, with an average of 61.4 registrations per page and a maximum of 2,229 registrations per page. These results include majority event types (e.g., event attributes of HTML tags, timer events, and XMLHttpRequest events) and event registration techniques supported in Firefox. The execution of event handlers may trigger further JavaScript inclusion and dynamic generation, implying that our captured insecure JavaScript practices are likely conservative estimates.

6. RELATED WORK

To the best of our knowledge, there is no directly related work on characterizing the insecure practices of JavaScript inclusion and dynamic generation. Therefore, we only briefly review some JavaScript related measurement studies. Krishnamurthy and Wills [16] measured the homepages of 1,158 unique sites selected from Alexa.com [36] to study the content delivery tradeoffs in Web access. The focus of their study is on the performance impact of extraneous content, and their results show that JavaScript is often used on popular webpages to retrieve extraneous content such as images and advertisements.

In the investigation of malware, several execution-based measurement studies [20, 24, 28] have been conducted to identify malicious webpages that contain code (in many cases, JavaScript code) for exploiting Web browser vulnerabilities and installing malware. Instead of targeting at malicious sites, our focus in this work is on legitimate websites' insecure JavaScript practices.

7. CONCLUSION

In this paper, we presented the first measurement study on insecure practices of using JavaScript on the Web. We focused on investigating the severity and nature of insecure JavaScript inclusion and dynamic generation. Through an instrumented Mozilla Firefox 2 Web browser, we visited the homepages of 6,805 popular websites in 15 different categories. We found that at least 66.4% of the measured websites have the insecure practices of including JavaScript files from external domains into the top-level documents of their homepages. Our in-depth analysis on the domain name relationship between JavaScript file inclusion sites and hosting sites further reveals the severity and nature of those insecure practices. Our measurement results on JavaScript dynamic generation show that the "evil" function eval() was called on 44.4% of the measured homepages, and the document.write() method and the innerHTML property were also used to generate JavaScript code. Our AST-based structural analysis on various DJS instances further uncovers their usages with respect to programming language functionality. Our analysis indicates that in common cases, safe alternatives do exist for both the insecure JavaScript inclusion and insecure JavaScript dynamic generation. Since Web-based attacks have become more common and damaging in recent years, we suggest website developers and administrators pay serious attention to these insecure JavaScript practices and use safe alternatives to avoid them. In the future, we will measure insecure JavaScript practices on more specific types of websites and webpages. We will also investigate whether other insecure JavaScript practices exist on the Web.

8. ACKNOWLEDGMENTS

The authors thank anonymous reviewers for their valuable comments and suggestions. This work was partially supported by NSF grants CNS-0627339 and CNS-0627340.

9. REFERENCES

[1] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[2] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. of the CCS*, pages 75–88, 2008.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the ICSM*, pages 368–377, 1998.

[4] A. Bortz, D. Boneh, and P. Nandy. Exposing private information by timing web applications. In *Proc. of the WWW*, pages 621–628, 2007.

[5] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, ISBN 1-55860-843-5, 2002.

[6] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang. A systematic approach to uncover security flaws in gui logic. In *Proc. of the S&P*, pages 71–85, 2007.

[7] W. S. (Editor). *Web Engineering: Principles And Techniques*. IGI Publishing, ISBN 1-591-40433-9, 2005.

[8] L. Falk, A. Prakash, and K. Borders. Analyzing websites for user-visible security design flaws. In *Proceedings of SOUPS*, pages 117–126, 2008.

[9] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, ISBN 0-596-10199-6, 2006.

[10] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Exploits: Cross Site Scripting Attacks and Defense*. Syngress, ISBN 1-597-49154-3, 2007.

[11] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. of the WWW*, pages 40–52, 2004.

[12] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proc. of the WWW*, pages 737–744, 2006.

[13] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proc. of the WWW*, pages 601–610, 2007.

[14] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proc. of the WWW*, pages 247–256, 2006.

[15] G. Kappel, B. Proll, S. Reich, and W. R. (Eds.). *Web Engineering: The Discipline of Systematic Development of Web Applications*. John Wiley & Sons, ISBN 0-470-01554-3, 2006.

[16] B. Krishnamurthy and C. E. Wills. Cat and mouse: content delivery tradeoffs in web access. In *Proc. of the WWW*, pages 337–346, 2006.

[17] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *Proc. of the CCS*, pages 221–234, 2006.

[18] B. Livshits and W. Cui. Spectator: detection and containment of javascript worms. In *Proc. of the USENIX Annual Technical Conference*, pages 335–348, 2008.

[19] E. Mendes and N. M. (Eds.). *Web Engineering*. Springer, ISBN 3-540-28196-7, 2005.

[20] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware in the web. In *Proc. of the NDSS*, 2006.

[21] S. Murugesan and Y. D. (Eds.). *Web Engineering : Managing Diversity and Complexity of Web Application Development*. Springer, ISBN 3-540-42130-0, 2001.

[22] T. Oda, G. Wurster, P. V. Oorschot, and A. Somayaji. Soma: Mutual approval for included content in web pages. In *Proc. of the CCS*, pages 89–98, 2008.

[23] T. A. Powell, D. L. Jones, and D. C. Cutts. *Web Site Engineering: Beyond Web Page Design*. Prentice Hall, ISBN: 0-13650-920-7, 1998.

[24] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proc. of the USENIX Security Symposium*, 2008.

[25] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: vulnerability-driven filtering of dynamic html. In *Proc. of the OSDI*, pages 61–74, 2006.

[26] D. C. Reis, P. B. Golgher, A. S. Silva, and A. F. Laender. Automatic web news extraction using tree edit distance. In *Proc. of the WWW*, pages 502–511, 2004.

[27] G. Rossi, O. Pastor, D. Schwabe, and L. O. (Eds.). *Web Engineering: Modelling and Implementing Web Applications*. Springer, ISBN: 1-84628-922-X, 2007.

[28] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proc. of the NDSS*, 2006.

[29] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proc. of the ICSE*, pages 171–180, 2008.

[30] C. A. Welty. Augmenting abstract syntax trees for program understanding. In *Proc. of the ASE*, pages 126–133, 1997.

[31] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, 1991.

[32] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *Proc. of the POPL*, pages 237–249, 2007.

[33] C. Yue, M. Xie, and H. Wang. Automatic cookie usage setting with cookiepicker. In *Proc. of the DSN*, pages 460–470, 2007.

[34] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *Proc. of the WWW*, pages 76–85, 2005.

[35] 24 ways: Don't be eval(). http://24ways.org/2005/dont-be-eval.

[36] Alexa Top Sites. http://www.alexa.com/browse?CategoryID=1.

[37] CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. http://www.cert.org/advisories/CA-2000-02.html.

[38] Cross-site scripting. http://en.wikipedia.org/wiki/Cross-site_scripting.

[39] eval - MDC. http://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Functions/eval.

[40] JavaScript. http://en.wikipedia.org/wiki/JavaScript.

[41] JSAPI reference - MDC. http://developer.mozilla.org/en/JSAPI_Reference.

[42] JSON in JavaScript. http://www.json.org/js.html.

[43] JSPrincipals - MDC. http://developer.mozilla.org/en/JSPrincipals.

[44] MSDN: innerHTML property. http://msdn.microsoft .com /en-us/library/ms533897(VS.85).aspx.

[45] Same origin policy. http://en.wikipedia.org/wiki/Same_origin_policy.

[46] SANS Top-20 2007 Security Risks (2007 Annual Update). http://www.sans.org/top20/2007/.

[47] SpiderMonkey (JavaScript-C) Engine. http://www.mozilla.org/js/spidermonkey/.

[48] Symantec Internet security threat report volume XIII: April, 2008. http://www.symantec.com/business/theme.jsp?themeid=threatreport.

[49] Unobtrusive Javascript. http://www.onlinetools.org/articles/unobtrusivejavascript/.

[50] XMLHttpRequest. http://www.w3.org/TR/XMLHttpRequest/.