Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2016, Article ID 7107042, 14 pages
http://dx.doi.org/10.1155/2016/7107042

Research Article

A Protection Mechanism against Malicious HTML and JavaScript Code in Vulnerable Web Applications

Shukai Liu, Xuexiong Yan, Qingxian Wang, Xu Zhao, Chuansen Chai, and Yajing Sun

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China

Correspondence should be addressed to Shukai Liu; wasd268@126.com

Received 7 November 2015; Revised 28 February 2016; Accepted 15 March 2016

Academic Editor: Ricardo Aguilar-López

The high-profile attacks of malicious HTML and JavaScript code have seen a dramatic increase in both awareness and exploitation in recent years. Unfortunately, exiting security mechanisms provide no enough protection. We propose a new protection mechanism named PMHJ based on the support of both web applications and web browsers against malicious HTML and JavaScript code in vulnerable web applications. PMHJ prevents the injection attack of HTML elements with a random attribute value and the node-split attack by an attribute with the hash value of the HTML element. PMHJ ensures the content security in web pages by verifying HTML elements, confining the insecure HTML usages which can be exploited by attackers, and disabling the JavaScript APIs which may incur injection vulnerabilities. PMHJ provides a flexible way to rein the high-risk JavaScript APIs with powerful ability according to the principle of least authority. The PMHJ policy is easy to be deployed into real-world web applications. The test results show that PMHJ has little influence on the run time and code size of web pages.

1. Introduction

Web applications are widely applied to all walks of life. Users are relying on web applications for all kinds of daily activities such as shopping, social activity, and banking. Web applications have become an important part of living and working. The new generation of web technologies represented by HTML5 is developing rapidly and growing popularity as well as the constantly updated web browsers such as Chrome, Firefox, and Edge. It indicates that a new era of web applications is coming.

Meanwhile, however, the new security risks of web applications have emerged [1]. It can be known from the published security reports. For example, WhiteHat 2013 demonstrated that 86% of the tested websites had 1 to 56 serious vulnerabilities and only 61% of the vulnerabilities could be resolved, but doing so required an average of 193 days from first customer notification. The attacks of malicious HTML or JavaScript code are one of the most significant and pervasive threats to the web application security such as cross-site scripting (XSS) attacks, embedding malicious third-party JavaScript code, and vulnerable third-party browser extensions or plug-ins.

XSS [2] summarizes a set of attacks on web applications that allows an attacker to inject HTML, typically including JavaScript code, into a vulnerable web page. It was ranked the third web application security risk by the OWASP (Open Web Application Security Project) in 2013 and the first software error by the SANS Institute in 2010. XSS attacks range in severity, which empower the adversary to conduct a wide range of potential attacks including redirection to phishing sites, password logging, session hijacking, clickjacking, stealing of sensitive data, self-propagating JavaScript worms, malware distribution, browser automation, and the ability to pivot onto an internal network to launch additional attacks.

Much of the power of modern web applications comes from the ability of a web page to combine HTML and JavaScript code from disparate servers. This is done to consume third-party services such as web analytics, enhance the web application with additional functionality such as integrating external mapping services, or include advertisements for monetary reasons. The survey of Nikiforakis et al. [3] showed that 88.45% of the Alexa top 10000 websites included at least one remote JavaScript. Unfortunately, these third-party servers are usually uncontrolled and untrusted. If an attacker is able to control the script's content, which is provided by the external provider, he/she is able to execute JavaScript in the context of the targeted web application.

In addition, the web browser using third-party extensions or plug-ins, which are phenomenally popular but usually buggy and vulnerable [4], may incur injection attacks of HTML and JavaScript code with the web browser's full privileges. That is to say, even if the web application has no vulnerability, it may also be attacked by malicious HTML and JavaScript code. In a word, the web application is running with a serious risk of malicious HTML and JavaScript code.

To combat these vulnerabilities, the industrial standard first-line of defense is sanitization, the process of applying encoding or filtering primitives called sanitizers to untrusted data. Although the sanitization technology has a very wide range of research [5], in practice it is still error-prone. A main reason is that the sanitization needs to know the details of vulnerabilities. It is very difficult to find all vulnerabilities in web applications even if we ignore unknown vulnerabilities. In addition, building reliable and usable sanitization in today's web application is not an easy job. For example, due to involving a variety of different coding contexts, it is very difficult to design correct sanitizers [6] or use them in a right way [7] for XSS.

Due to the fact that the quantitative occurrence of these vulnerabilities remains high according to the recent studies [8, 9] even after several years of increased attention, the safeguards enforced by web browsers as an important second-line of defense have been proposed such as SOMA [10], DCS [11], and CSP [12]. They are designed to mitigate vulnerabilities caused by missing or incorrect sanitization, which have aroused widespread attention. Different from other mechanisms with many problems such as great changes of the existing web platform or low efficiency of implementation, CSP (Content-Security-Policy) stands out in security, usability, and compatibility, which eventually became a W3C recommendation.

CSP is developing rapidly. It has been supported and promoted by almost all popular web browsers. The number of web applications adopting CSP is increasing rapidly [13, 14]. W3C has successively proposed three recommendations: CSP 1.0 in 2012, CSP 1.1 in 2014, and CSP 2 [15] in 2015. Now CSP is one of several basic security mechanisms actually employed by modern web browsers and has developed into a comprehensive security framework for web applications combining with many previous techniques.

CSP is a typical functionality-oriented access control [16] on web pages in web browsers. It defines which functionalities of the web page are allowed or forbidden by web browsers. That is, it protects some crucial functionality of HTML and JavaScript code in web applications so that they cannot be exploited by attackers, instead of eliminating vulnerabilities like sanitization. In particular it mainly provides a site's administrators control over the permissible sources such as scripts and images on an origin level (i.e., protocol, host, and port) with a rule set.

However, the approach of CSP which relies on functionality-level or origin-level granularity for simplicity has essentially brought a lot of unsolved problems in protection against malicious HTML and JavaScript code. In order to resolve these problems, a new protection mechanism named PMHJ for short is proposed in this paper.

To summarize, our paper makes the following contributions:

(i) We give the first systemic analysis of CSP's problems in defense against malicious attacks of HTML and JavaScript code.

(ii) We give a method to provide enough protection against the injection attack and node-split attack of HTML elements only via HTML attributes.

(iii) We provide a fine-grained control over the insecure HTML and JavaScript usages which may incur injection attacks.

(iv) We present the first solution to rein the high-risk JavaScript APIs according to the least authority principle.

This paper is organized as follows. We give an illustration of problems in Section 2. PMHJ is overviewed in Section 3. Section 4 describes the policy of PMHJ and Section 5 presents the security properties. The implementation method is given in Section 6. The security and usability are evaluated in Section 7. Section 8 covers the related work. We discuss future work in Section 9. Section 10 concludes this paper.

2. Problem Definition

The security weaknesses of CSP in protection against malicious HTML or JavaScript code are systemically defined below.

2.1. Problem 1: Malicious Injection of HTML Elements. The defense ability of CSP against malicious injection of HTML elements is very limited. CSP can only identify the HTML elements whose attribute values are not consistent with the whitelist of CSP. For example, the origin of JavaScript code is limited to load only from *a.com* so that the HTML element *<script src="b.com">* is invalid, but an attacker can inject arbitrary code from the origin *a.com* such as *<script src="a.com/eval.js">*.

A simple exploit is to undermine the application logic by injecting further HTML elements pointing to the whitelist origins of CSP. All scripts in a web page run in the same execution context. And JavaScript provides no native isolation or scoping such as via library specific namespaces. Hence, all side effects that a script causes on the global state directly affect all scripts that are executed subsequently.

It is possible to combine with JSONP (JSON with padding) to enable more complex attacks. JSONP is a popular method for building JavaScript APIs, such as integrating with third-party services (e.g., to implement search or mapping capabilities) and retrieving private first-party data (e.g., an additional referer check or an XSRF token). The attacker is able to invoke any functions of his choice, often with at least partly controllable parameters, by specifying them as the callback parameter on the JavaScript API call. For example, a simple injected JSONP call is *<script src='login.php?user=a&callback=set_share'>* where *set_share* is any function which has been defined in the script context of the web page.

Due to the fact that in current web applications a lot of content such as advertisements needs to be loaded from untrusted third-party servers which must be whitelisted by CSP, an attacker can embed malicious or vulnerable code from these untrusted servers into the web page such as malicious [17] or vulnerable [18] advertisements via HTML element injection.

Even without untrusted origins, since scripts from origins in the whitelist of CSP can be combined freely, more complex attacks can be realized by a combination of HTML element injection attacks. That is, CSP allows the attacker to load arbitrary scripts and other resources found anywhere on the whitelisted origins in an unexpected context, an incorrect order, or an unplanned number of times. An attacker is able to control the URLs of included scripts and the order in which they are loaded in. Given the growing quantity and complexity of script code hosted by websites, a nontrivial site might provide an attacker with a well-equipped toolbox. Nikiforakis et al. [3] gave some subtle attacks which enabled the same class of script inclusion attacks and showed their practical applicability as alternatives to a full compromise of the script provider.

Furthermore, only a small part of HTML elements and their attribute values mainly about the network request are protected by CSP. However, there are still many other unprotected HTML elements and attributes with powerful ability which can be exploited by attackers to result in serious damage.

2.2. Problem 2: The Integrity Protection of HTML Elements. The integrity protection of HTML elements is an open problem. Most XSS attacks can be classified as a break of document structure integrity according to the work by Nadji et al. [19]. A straight forward scenario for such an attack is the node-split attack [20]. The attacker destroys the integrity of HTML elements by injecting a string to split the enclosing HTML element and then inject a new HTML element. A simple example is *<div>elliptically legitimate code</div><script>malicious script</script><div>elliptically legitimate code</div>* where the underlined part is arbitrary JavaScript code injected by attackers.

Other attacks that break the structure integrity of HTML elements include the attribute injection attack, the dynamic code injection attack, and the dynamic active HTML update attack. Some attack examples are seen in [19]. However, CSP provides no integrity protection of HTML elements.

2.3. Problem 3: Unsafe HTML Usages. The unsafe usages of HTML code in web applications are easy to result in security weaknesses. An obvious case is the event handler attribute of HTML elements and *javascript:protocol* URLs causing an insecure mix of HTML and JavaScript code which are disabled by CSP. However, there are a large number of other unsafe usages which are actually supported by web browsers for different purposes such as the fault-tolerant mechanism or quirks mode. It may lead to serious damage if they are exploited by attackers.

The most serious case may be that several DOM (Document Object Model) elements with the same *id* attribute value are allowed by web browsers, but only the first value which may be controlled by the attacker will be returned on *getElementById* lookups. What makes it worse is that, for several special elements such as *img*, *iframe*, or *embed*, the *id* attribute is additionally inserted into the document object which has a higher priority than built-in and script-created variables. It can be exploited for the namespace attack, which makes the user unwittingly interact with UI controls, shadow built-ins such as *document.domain*, in order to interfere with certain security decisions, disrupt the operation of methods such as *document.createElement*, or fabricate the availability of the security JavaScript APIs such as *postMessage*. These attacks are application-specific. A simple example is *<input id='set_share' value='liu'><input id='set_share' value=' '> <script> share = document.getElementById('set_share').value; if (set_share) mode = 'public';</script>* where the underlined part is injected by attackers and the variable values of *share* and *mode* are '*liu*' and '*public*', respectively.

Many other insecure HTML usages are dangerous. The injection of the *base* or *meta* element outside the standards-mandated *head* element can be exploited to hijack relative URLs using <*base href* = ' '/>, refresh or redirect the web page using <*meta http-equiv*='*Refresh*'>, or even inject the CSP policy possibly in Google Chrome using <*meta http-equiv*='*Content-Security-Policy*'>.

Since the top-level occurrence of the <*form*> tags always takes precedence over subsequent appearances, the nested *form* elements are able to lead to the form parameter injection to reroute existing forms, intercept browser-managed passwords, or infiltrate the application logic. A simple example is *<form action='http://evil.com'> elliptically legitimate code <form action='update.php'> <input value='liu'> <input type='submit'/> </form>* where the underlined part is injected by attackers.

Another common risky HTML usage is caused by the fault-tolerant mechanism for incorrect HTML elements including the unclosed tags and the attribute values with nonterminated quotation marks. They can be exploited for different malicious purposes such as altering the appearance of the targeted website or content exfiltration. A typical example is *<img src='http://evil.com? elliptically legitimate code <input name="xsrf_token" value="123"> elliptically legitimate code ' elliptically legitimate code <div>* where the underlined part is injected by attackers and the request for the image will carry the sensitive information.

2.4. Problem 4: Insecure JavaScript Usages. The XSS vulnerability in JavaScript code is very common in web applications such as the DOM based XSS. These vulnerabilities are usually introduced by some unsafe JavaScript usages. The functions that can convert a string to JavaScript code dynamically such as *eval* which are disabled by CSP are the most serious.

There is another class of JavaScript APIs converting a string to HTML code unsafely (namely, interfacing the

HTML parser) such as *document.write* and *innerHTML*, which is ignored by CSP. The XSS attack occurs once the data controlled or tainted by the attacker is inputted into these JavaScript APIs. These insecure JavaScript APIs with powerful ability to inject arbitrary HTML and JavaScript code are easy to incur vulnerabilities and make the sanitization of untrusted input a very challenging task [7]. For example, *function foo(taint_data)*{*document.write(*"<*input onclick ='foo(*"+*taint_data+*") '>"*);*} makes the server side sanitization infeasible since *taint_data* is repeatedly pumped through the JavaScript and HTML parsers.

On the other hand, these insecure usages are not necessary in web application development and are recommended to be replaced with the safe HTML5 DOM methods to explicitly create HTML elements such as *createElement* and *appendChild* [21]. However, they are still prevalent in contemporary web applications due to the simplicity or compatibility and are proved to be easy to incur vulnerabilities in practice [22].

In addition, the strict mode of JavaScript is introduced from ECMAScript 5 [23] which has been supported by all major web browsers. It is used to limit the unsafe grammars of JavaScript in order to avoid some potential bugs or errors. However, the strict mode is not popular in current web applications mainly for the sake of compatibility.

2.5. Problem 5: Abuse of JavaScript APIs. A prominent problem which web applications are now facing is that each web page owns the whole ability of HTML and JavaScript code. Once the web page is loaded, arbitrary HTML and JavaScript code is allowed to be executed in the web browser. In other words, once it is suffering from vulnerabilities such as XSS, an attacker can inject and execute arbitrary HTML or JavaScript code in the web page. It means that a vulnerability in the web page will damage all the functionalities or ability without limitation. The most serious case is that thousands of JavaScript APIs with powerful ability are provided by web browsers, but only a very few of them are actually needed in real-world web pages. This obviously violates the principle of least authority that if an application does not need a capability, it should not have it.

JavaScript is one of the most important components on the web platform which enables a new generation of dynamic and interactive web applications. If modern web browsers are considered as a simple operating system, the web application can be defined as web pages using JavaScript APIs to obtain high-level service (e.g., network, storage, and hardware) provided by the web browser. It may lead to a serious security problem if one or several of these JavaScript APIs with powerful ability are exploited by attackers. For example, the JavaScript API *ActiveXObject* is not used in most web applications but can be exploited to connect other native applications in IE by attackers.

Unfortunately, at present, there is no security mechanism to give a solution to the problem that each web page can invoke all the JavaScript APIs provided by web browsers. As JavaScript APIs such as HTML5 APIs are becoming more and more powerful than before, the problem must occupy more attention of security practitioners.

3. System Overview

3.1. Approach Overview. All the problems discussed above indicate that the protection of CSP against malicious HTML and JavaScript code is not enough. It is time to approach these problems from a different angle instead of whitelisting the origins of resource. The goal of PMHJ is to provide not only a systemic strengthening of CSP, but also an independent and complete security mechanism against malicious HTML and JavaScript code in vulnerable web applications.

Firstly PMHJ provides a content protection mechanism for web applications which enables the web browser to identify and prevent the code injected by attackers. The content of web pages is loaded directly or indirectly by HTML elements, and different kinds of HTML elements usually correspond to different content types. Thus, HTML elements are treated as the basic unit of content protection in PMHJ. The HTML5 recommendation and web browsers define a number of HTML elements for web applications, but not all of them are used to load content such as the *br* element. PMHJ only protects the HTML elements which are faced with serious security risks in web applications since a complex protection mechanism for all HTML elements is unnecessary.

Inspired by the *nonce* attribute of the inline *script* or *style* element used in CSP 1.1 and CSP 2 to solve the efficiency problem [24] caused by disabling inline scripts in CSP 1.0, a new *nonce* attribute is defined for the protected HTML elements which is used to specify a random value in PMHJ. The web browser distinguishes the developer's HTML elements from the attacker's HTML elements via the *nonce* attribute value.

Another method to identify the HTML elements injected by attackers is computing the hash values of HTML elements, which has been proposed to identify the *script* element in BEEP [20] and the inline *script* or *style* element in CSP 1.1 and CSP 2. Different from them, we make use of the hash values of HTML elements to protect the integrity of HTML elements which cannot be destroyed by malicious attacks. PMHJ defines a *hash* attribute for the protected HTML elements, which is used to record the hash value of the HTML element. The web browser checks the *hash* attribute value to confirm the integrity.

Different from CSP which only disables the functions converting a string to JavaScript code, PMHJ systemically strengthens the limits on the insecure usages of JavaScript which may incur bugs or vulnerabilities. PMHJ also provides a strict restriction on unsafe HTML usages supported by web browsers. By these ways, PMHJ ensures the content security of web pages even if without limiting the content servers via CSP.

At last PMHJ allows the developer to get a fine-grained control over the JavaScript APIs which can be invoked in each web page. It provides a strong protection for the unused JavaScript APIs against malicious attacks. Due to the fact that a large number of JavaScript APIs are provided by web browsers, the method of the whitelist to manage all JavaScript APIs is impossible. PMJA defines a set of JavaScript APIs faced with serious security risks and employs the blacklist to rein these JavaScript APIs. PMHJ utilizes JavaScript objects

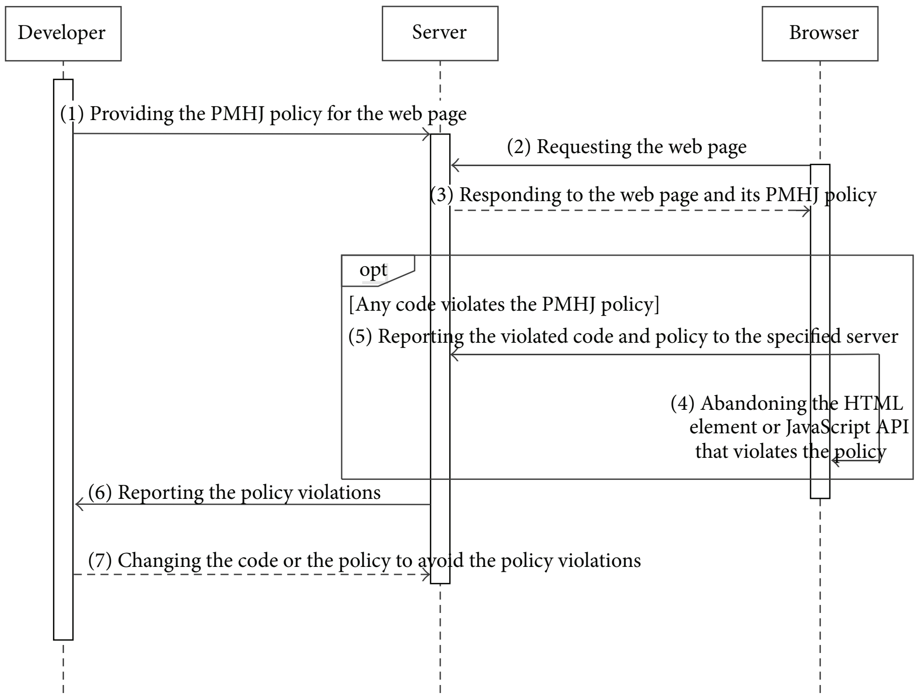
Developer
Server
Browser
(1) Providing the PMHJ policy for the web page
(2) Requesting the web page
(3) Responding to the web page and its PMHJ policy
opt
[Any code violates the PMHJ policy]
(5) Reporting the violated code and policy to the specified server
(4) Abandoning the HTML element or JavaScript API that violates the policy
(6) Reporting the policy violations
(7) Changing the code or the policy to avoid the policy violations

FIGURE 1: The system architecture of PMHJ.

[23] to represent JavaScript APIs, which include the host objects supplied by the hosting environment, the objects defined by the ECMAScript standard, and the objects created in JavaScript code, respectively.

3.2. System Architecture. The methodology of PMHJ policies is given above. Another important problem is how to deploy these policies into the web platform considering the security, usability, and compatibility at the same time. We give the system architecture to deploy PMHJ into the web platform below, as shown in Figure 1.

On the one hand, the security mechanism on the pure browser side is not easy to balance the functionality requirements and the security. For example, a strict policy may prevent the normal functionality and a relax policy can greatly reduce the security. On the other hand, the security mechanism on the pure server side is not easy to guarantee whether the policy is strictly enforced by web browsers, because the code of HTML or JavaScript is finally run in the browsers not the servers. In a word, the developer of web application knows exactly what HTML or JavaScript code (namely, the HTML element or the JavaScript API) should be executed or not compared with the web browser, and the web browser knows exactly whether the code is run or not.

Therefore, PMHJ adopts a working mechanism that the policy is designed by the developer, transported by the HTTP, and eventually enforced by the web browser. That is, PMHJ needs to be supported by both web applications and web browsers. PMHJ designs and enforces policies for each single web page independently, because each web page displayed in the browser is running with an independent running environment.

Specifically, the PMHJ policy is designed and provided by the developer for each web page according to the security requirement (sometimes via the semiautomatic or automatic policy tool). When the web page is requested by a web browser, the server responds to the web page and its PMHJ policy to the browser. When receiving the response, the browser reads the PMHJ policy before parsing and running any content of the web page. In the end, the browser strictly enforces the PMHJ policy when parsing and running the web page.

Moreover, if a PMHJ policy is violated in the web browser, it means that the policy is improper for the web page or that the web page is suffering from malicious attacks. Inspired by CSP, the mechanism reporting the information of policy violation to the developer or security administrator is introduced by PMHJ. The web browser reports the violation information to the server specified by the policy via the HTTP request, respectively. It is able to report the policy violation to the security administrator when the web page is running, so that it can detect the policy violation caused by malicious attacks. That is, it can be used as an early-warning system to notify the administrator when an injection attack may be occurring. It can also help to remind the developer to change the code or the policy to avoid the policy violation when debugging the web page.

4. PMHJ Policies

4.1. Policy Design. PMHJ defines a set of HTML elements with serious security risks, namely, E = {*script, form, style, base, link, iframe, frame, meta, object*, *embed*}. Each HTML element in the set E is used to load a type of content, such

TABLE 1: The JavaScript objects with serious security risks.

Category	Functionality	Object
Communication	Network communication, Cross-Document Messaging, Web Real-Time Communication	*WebSocket*, *EventSource*, *XMLHttpRequest*, *postMessage*, *RTCPeerConnection*, *RTCDataChannel*
Storage	Storage, File System API	*localStorage*, *sessionStorage*, *indexedDB*, *openDatabase*, *requestFileSystem*
Device	Hardware Device Access	*getUserMedia, geolocation, vibrate, battery, connection*
Interaction	Interaction with users (such as popups, desktop notifications, and full screen)	*open (window.open), showModalDialog, showModelessDialog, alert, prompt, print, createPopup, Notification, requestFullscreen*
Privilege	Special privileges (e.g., connecting to other applications) and High-Risk HTML5 APIs (e.g., executing the script in the background)	*clipboardData, addFavorite, addPanel, setHomePage, AddSearchProvider, ActiveXObject, Worker, history, webkitCreateShadowRoot*

as the *script* element loading JavaScript code. These HTML elements and their corresponding content types are faced with a serious security risk which is exploited by attackers leading to a serious consequence.

PMHJ defines a new attribute named *nonce* for all HTML elements in E. The *nonce* attribute value is a string which should be randomly generated on the server for each request before inserting it into an HTTP response.

It needs to be stressed that if the HTML element $e \in E$ needs to be protected by the *nonce* attribute, every HTML element e in the web page must use the *nonce* attribute at the same time. The length of the *nonce* attribute value can be arbitrarily assigned by the server, but the string of 16 bytes is recommended in order to ensure safety and availability. In a web page different *nonce* attributes can use different random values. However, we recommend that all the *nonce* attribute values in a web page might share the same random string for ease of use.

A global attribute named *hash* for all HTML elements is defined in PMHJ. Its attribute value is the hash value of the HTML element, which is computed by the SHA-2 algorithms including SHA-256, SHA-384, and SHA-384. However, only one kind of these hash algorithms can be used in a web page. For the sake of simplicity and efficiency that a large number of computing *hash* values increase the rendering time of the web page, we recommend only using the *hash* attribute for the *script* element.

All JavaScript objects that convert a string to JavaScript code can be summarized as a set $J = \{eval, Function, setTimeout, setInterval\}$.

All JavaScript objects that parse a string to the HTML parser unsafely are denoted as a set $H = \{write, writeln, innerHTML, outerHTML, pasteHTML, insertAdjacentHTML\}$.

The JavaScript objects with serious security risks are defined as a set U, which is listed in Table 1. These JavaScript objects with powerful capacity are usually the goals or means of attacks.

Define $M = \{communication, storage, device, interaction, privilege\}$, where each $m \in M$ denotes a category of JavaScript objects in Table 1.

4.1.1. Policy Language. The policy of PMHJ is a rule set which needs to be enforced by web browsers. A simple policy language of PMHJ is given as follows.

A policy is composed of directives and the directives are split on spaces. All the directives of a web page are denoted as a set P and the set P is the PMHJ policy of the web page.

An element directive with the same name of the element in E is used to declare that all the HTML elements with the name in the web page should use the *nonce* attribute. In particular, the element directive *allelement* satisfies that $allelement \in P$ is equivalent to $E \subseteq P$.

A nonce directive *nonce-$random* is used to declare a string *$random* which can be used as a valid *nonce* attribute value in the web page.

A hash directive in $S = \{sha\text{-}256, sha\text{-}384, sha\text{-}512\}$ is used to specify the hash algorithm used in the web page.

The directive $string2js$ is used to proclaim that the set of JavaScript objects J should be disabled to convert a string to JavaScript code.

The directive *string2html* is used to declare that the set of JavaScript objects H is disabled.

The directive *htmlstrict* and the directive *scriptstrict* are used to state a kind of strict mode of HTML and JavaScript, respectively.

A multiple-object directive with the same name of the element in M is used to declare that the category of JavaScript objects defined by the homonymic element in M is disabled. In particular, the multiple-object directive *allobject* satisfies that $allobject \in P$ is equivalent to $M \subseteq P$.

An object directive with the same name of the element in U is used to state that the homonymic JavaScript object is still able to be usable even if it is disabled by multiple-object directives.

A report directive *report-$URL* is used to specify a server by the URL address *$URL* where the web browser reports the policy violation information (including the line and column number of the code and the violated directive at least) if a PMHJ policy violation occurs.

To sum up, all the directives of PMHJ can be denoted as a set $I = E \cup S \cup M \cup U \cup \{allelement, allobject, nonce\text{-}\$random, string2js, string2html, htmlstrict, scriptstrict, report\text{-}\$URL\}$. A PMHJ policy P satisfies $P \subseteq I$.

4.2. Policy Delivery. PMHJ delivers policies from the server to the web browser via the HTTP response header. A new field named *PMHJPolicy* is defined in the HTTP response header. It is used to specify a set of directives, namely, the PMHJ policy, which should be enforced by the web browser.

For example, a response might include the following PMHJ header field to deliver the PMHJ policy of the web page:

PMHJPolicy: *nonce-z0h3sdfaEDNnf03n nonce-dDNnf03-nceIOfn39 allelement htmlstrict string2js allobject string2html XMLHttpRequest report-http://www.exa.com/report.php*

This header field delivers the PMHJ policy of the web page to inform the web browser that when running the web page, all HTML elements defined in the set E must own a *nonce* attribute with the attribute value *z0h3sdfaEDNnf03n* or *dDNnf03nceIOfn39*, all JavaScript objects defined in the set $U \cup J \cup H$ except *XMLHttpRequest* must be disabled, and any code violates the policy must report the detailed information to the server *http://www.exa.com/report.php*.

4.3. Policy Enforcement. After receiving the HTTP response and before parsing and executing the content of the web page, the web browser obtains all the PMHJ directives denoted by a set P by reading and parsing the values of the *PMHJPolicy* field in the HTTP response header.

To help to describe how the web browser internally enforces the PMHJ policy, we give the formalized definition of the needed sets at first.

Let $E_P = P \cap (E \cup \{allelement\})$ be the set of HTML elements specified by all the element directives in P, where *allelement* $\in P$ means $E_P = E$. It is obvious that $E_P \subseteq E$. E_P is the set of all the HTML elements which need to be verified.

Let N_P be the set of values specified by all the nonce directives in P, and let $S_P = P \cap S$ be the set of hash algorithms specified by all the hash directives in P. N_P is the set of all the legitimate nonce attribute values of the HTML elements in E_P, and S_P is the set of all the legitimate hash algorithms for the hash attribute values.

Let $M_P = P \cap (M \cup \{allobject\})$ be the set of JavaScript objects specified by all the multiple-object directives in P, where *allobject* $\in P$ means $M_P = U$. Let $U_P = P \cap U$ be the set of JavaScript objects specified by all the object directives in P. Then, the difference set of M_P and U_P, namely, $D_P = M_P - U_P = \{a \mid a \in M_P, a \notin U_P\}$, is actually the set of all the JavaScript objects which need to be disabled according to all the multiple-object directives and the object directives in the policy.

Let R_P be the set of URLs specified by all the report directives in P. In fact, R_P gives all the servers that the policy violation information should be reported to by the web browser.

If $P \neq \emptyset$, the web browser strictly enforces the following policies in the whole process of parsing and executing the web page:

(i) For each $e \in E_P$, if the HTML element e does not have the *nonce* attribute or the *nonce* attribute value n_e is not correct, namely, $n_e \notin N$, then the browser abandons the HTML element e. Otherwise, the element e is parsed as usual.

(ii) The *nonce* attributes of HTML elements in E_P cannot be accessed by JavaScript code.

(iii) If *script* $\in P$, the event handler attributes of HTML elements and *javascript:protocol* URLs that cause JavaScript code to execute are disabled.

(iv) If *style* $\in P$, the *style* attributes of HTML elements are disabled.

(v) If a HTML element e has a *hash* attribute with the attribute value h_e and there is only one hash algorithm in P, namely, $|S_P| = 1$, the browser computes the hash value h of the element e using the hash algorithm specified by S_P. If and only if $h = h_e$, the HTML element e is parsed; otherwise, it is abandoned.

(vi) If *htmlstrict* $\in P$, then

  (a) the incorrect HTML elements are rejected by the browser including that the tags cannot be closed normally and that the attribute values cannot be terminated with quotation marks,

  (b) the nested *form* elements, the HTML elements with the same *id* attribute value, and the *base* or *meta* element used outside the *head* element are all invalid.

(vii) If *string2html* $\in P$, all the JavaScript objects in the set H are disabled.

(viii) If *string2js* $\in P$, all the JavaScript objects in the set J are disabled to convert a string to JavaScript code.

(ix) If $D_P \neq \emptyset$, all the JavaScript objects in D_P are disabled.

(x) If *scripstrict* $\in P$, then the strict mode of ECMAScript is applied to all the JavaScript code in the web page.

(xi) If $R_P \neq \emptyset$ and a policy above (the policy from i to x) is violated, the violation information including the reason and code of the policy violation is reported to the specified servers of R_P via the HTTP request.

4.4. Examples. Often, the easiest way to explain usages of a policy language is through examples. Here we provide simple samples that are illustrative.

Example 1. PMHJPolicy: *device geolocation report-http://www.exa.com/report.php*.

The JavaScript objects in the set D_P = *device* − {*geolocation*} = {*getUserMedia*, *vibrate*, *battery*, *connection*} are disabled according to the policy in Example 1. And if any JavaScript object in D_P is invoked, then a violation report will be sent to the server specified by the URL *http://www.exa.com/report.php*.

Example 2. PMHJPolicy: *script htmlstrict string2html string-2js nonce-z0h3sdfaEDNnf03n*.

Table 2: The codes violating the policy in Example 2.

The code violating the policy	Reason	Policy
<script>*malicious script*</script>	Missing the *nonce* attribute	(i)
<script nonce="dDNnf03nceIOfn39">*malicious script*</script>	An incorrect *nonce* attribute value	(i)
<script nonce="z0h3sdfaEDNnf03n"id="example"> alert(document.getElementById("example").getAttribute("nonce")); </script>	The *nonce* attribute cannot be obtained or set by JavaScript code	(ii)
<a href = "javascript:*malicious script*"> example</a> <a href = "#"onclick="*malicious script*">example</a>	The *javascript:protocol URL* and the attribute *onclick* are disabled	(iii)
<img src ='http://evil.com/log.cgi? <base href = 'http://evil.com/' > <form action = 'update_profile.php' > <input type = 'hidden' id='share_with' value='fred' > <input id = 'share_with' value='bogo' > </form>	An incorrect *src* attribute value The *base* element outside the *head* element The two *input* elements owning the same *id* attribute value are both invalid	(vi)
<script nonce="z0h3sdfaEDNnf03n"> document.write("*malicious html or script*") </script>	The call to the JavaScript object *document.write* is forbidden	(vii)
<script nonce="z0h3sdfaEDNnf03n"> setTimeout("*malicious script*",1000) </script>	The first parameter of *setTimeout* is a string	(viii)

The codes in Table 2 are all violating the policy in Example 2. That is, these code are invalid in the web page employing the policy.

5. Security Properties

The security properties which can or should be satisfied by PMHJ are discussed below.

Property 1. If $e \in P \cap E$, the e element injected by attackers is invalid.

Proof. It is a fact that the JavaScript code in web applications cannot access the value of the response header field including the *PMHJPolicy* header field. The attacker also cannot access the *nonce* attribute value in the web page according the policy (ii) via JavaScript code. The *nonce* attribute value of the e element is randomly generated when the server responds to the web page every time, so that the attacker cannot guess a valid *nonce* attribute value. Therefore, the attacker without a valid *nonce* attribute value cannot inject a valid e element which is able to be parsed by the web browser according to policy (i). □

PMHJ utilizes the *nonce* attribute with a random value to help the web browser to distinguish which HTML elements belong to the developer and which HTML elements are injected by attackers. That is, PMHJ is able to protect web applications against malicious injection attacks of HTML elements.

Property 2. If the HTML element e has a *hash* attribute and $|S_P| = 1$, the element e which is suffering from a node-split attack will not be parsed by the web browser.

Proof. According to the propertyof the SHA-2 algorithm [25], given a hash value, the attacker cannot construct a string with the same hash value within the existing computing power at least. If a node-split attack on the element e occurs, the integrity of e is broken so that the hash value of the element e is changed. According to policy (v), if the web browser detects that the hash value of e is different from the *hash* attribute value of e, it will discard the HTML element e. □

It is obvious that the *nonce* attribute cannot protect the integrity of HTML elements. The *hash* attribute provides a simple and effective method to prevent the node-split attack which CSP relying on an origin-level protection cannot prevent. Unfortunately, since some other attacks that break the HTML document integrity such as the attribute injection attack [19] need a value-level granularity protection such as sanitization, PMHJ providing a HTML element level protection cannot work. Other attempts against these attacks such as DSI [19] and blueprint [26] also fail in practice due to many problems especially in availability and compatibility with the exiting web platform, although they give valuable solutions in theory.

Each HTML element in E is used to load a type of content in the web page. However, different HTML elements can load the same type of content. For example, the *style* and *link* elements can both be used to load CSS (cascading style sheets). For the sake of describing convenience, PMHJ considers that the CSS loaded by the *style* element and the *link* element are the two different content types. In other words, the HTML elements in E are used to load different content types.

Property 3. If $e \in E_P$, the content type of e in the web page can only be loaded by one of the following two methods:

(1) loaded by e element which has a *nonce* attribute with the attribute value n_e satisfying $n_e \in N$; (2) created by a piece of JavaScript code satisfying the notion that if $script \in P$, the JavaScript code is loaded by a *script* element which has a *nonce* attribute with the attribute value $n_s \in N$.

Proof. PMHJ disables the insecure methods of loading JavaScript and CSS by policies (iii) and (iv). There are only two kinds of methods to load the content types in E into the web page using PMHJ, that is, loaded by HTML elements or created dynamically by JavaScript code. Then we can get Property 3 by combining Property 1 with policy (i). □

Property 3 shows that the content type protected by PMHJ in the web page can only be loaded by the trusted HTML element or generated by the JavaScript code which is loaded by a trusted *script* element. That is, PMHJ can ensure that the content types in E load only by secure HTML elements.

The properties above provide no protection against vulnerabilities in JavaScript code such as DOM based XSS [2]. These vulnerabilities are caused by some unsafe JavaScript objects. The directive *string2html* and the directive *string2js* disable these dangerous JavaScript objects which convert a string to HTML and JavaScript code, respectively. The directive *scriptstrict* makes the web browser able to apply the strict mode to all the JavaScript code in the web page. The directive *htmlstrict* disables the unsafe HTML usages defined in Problem 5. These directives are able to avoid vulnerabilities, mitigate attacks, and help developers to improve their programming habits.

A combination of multiple-object directives and object directives is able to disable the unneeded JavaScript objects with serious security risks defined in Table 1 according to the least authority principle. The disabled JavaScript objects cannot be exploited by attackers any more. It also makes different web pages own different JavaScript APIs which can be invoked. That is, it allows a better separation of JavaScript even if they are from the same origin as an extension to the same origin policy.

Another important security property is that all the PMHJ policies only limit the exiting ability of HTML and JavaScript code in web applications. They provide no new functionality or ability to a web page. An attacker cannot make a web page less secure than it would have been without PMHJ even if by maliciously injecting the PMHJ policy into the web page. Therefore, PMHJ only strengthens the security of web applications by restricting what it can do under control of an attacker. The ability disabled by PMHJ cannot be exploited by attackers, so that the web page using PMHJ is at least safer than the web page without using PMHJ.

One or more JavaScript objects need to be disabled by the directives including *string2html*, *string2js*, and the multiple-object directives. A JavaScript object discussed in PMHJ stands for a kind of ability or functionality provided by web browsers. That is, when a JavaScript object is disabled by PMHJ, the JavaScript code in the web page cannot obtain the ability of the JavaScript object any more. It is an important property which should be ensured in the implementation of web browser. A serious issue is that, according to the same origin policy, a call of JavaScript objects in other HTML documents with the same origin is possible. For example, *window.eval*, *window.parent.eval*, and *window.frames[0].eval* can be called by the code, which are all aliases for the same function in the native code of the web browser. And other access paths specified by web standards or provided by some nonstandard browser features for a particular release should be taken into consideration in the browser implementation of PMHJ.

We emphasize that there are great differences of insecure HTML usages and unsafe JavaScript objects in different web browsers, such as the *applet* element obsoleted by W3C but still supported by many web browsers, the *expression* or *behavior* attribute of CSS supported by old IE, the XBL bindings ever used in Firefox, and many specific JavaScript objects with different names or functionality such as experimental HTML5 API. That is, the definitions of E and U are specific to the web browser. HTML5 is developing rapidly and web browsers are updating frequently, so that the definitions of E and U should be maintained and improved according to the specific web browser.

6. Implementation

PMHJ needs to be supported by both web applications and web browsers. The key technology to implement PMHJ in web applications and browsers is discussed, respectively, as follows.

To support PMHJ, every the protected HTML element in the web page needs to add a *nonce* attribute. The unsafe usages of HTML and JavaScript disabled by the PMHJ policy need to be replaced by some more safe usages, such as adding a *click* event to replace *javascript:protocol URLs*, using *addEventListener* to add events instead of using event handler attributes of HTML elements, using a function as the first parameter of *setTimeout* and *setInterval* instead of a string, replacing *write* and *innerHTML* with the usages recommended by HTML5 (e.g., *createElement* and *createTextNode*), and *eval* being replaced by the recommended safe alternatives [22] (e.g., *JSON.parse*). In addition, the *PMHJPolicy* header field needs to be set in the HTTP response header on the server side.

The browsers supporting PMHJ are necessary and crucial. As we know, WebKit [27] is one of the most popular web browser engines employed by Chrome, Safari, and Opera. We constructed a prototype implementation of PMHJ by the modification of WebKit r174650. The implementation totally adds approximately 6700 lines of C++ codes to patch various parts of the code of WebKit. Specifically, the several key parts are as follows.

Firstly a C++ class with 1086 lines of codes is used to parse the PMHJ policy in the response header. Then the protected HTML elements are verified strictly according to policies (i), (iii), (iv), (v), and (vi) when the browser is parsing the HTML code of web pages where 1827 lines of code are changed in the original HTML parser. In particular, since the SHA-2 algorithms have been supported by WebKit, policy (v) can take advantage of the exiting hash algorithms. Policy (ii) can

Table 3: The web pages used for the experiment.

Website	Web page
Google.com	*mail.google.com, google.com, translate.google.com, accounts.google.com, news.google.com*
Facebook.com	*facebook.com, apps.facebook.com, web.facebook.com, m.facebook.com, developers.facebook.com*
Baidu.com	*baidu.com, tieba.baidu.com, zhidao.baidu.com, baike.baidu.com, image.baidu.com*
Hao123.com	*hao123.com, movie.hao123.com, tv.hao123.com, m.hao123.com, soft.hao123.com*
Amazon.com	*amazon.com, smile.amazon.com, aws.amazon.com, developer.amazon.com, kdp.amazon.com*
Wikipedia.org	*en.wikipedia.org, ja.wikipedia.org, ru.wikipedia.org, de.wikipedia.org, zh.wikipedia.org*
Sina.com.cn	*blog.sina.com.cn, news.sina.com.cn, sina.com.cn, sports.sina.com.cn, house.sina.com.cn*
Twitter.com	*twitter.com, analytics.twitter.com, mobile.twitter.com, support.twitter.com, ads.twitter.com*
Linkedin.com	*linkedin.com, in.linkedin.com, help.linkedin.com, business.linkedin.com, uk.linkedin.com*
Youtube.com	*youtube.com, youtube.com/channel/UC-9-kyTW8ZkZNDHQJ6FgpwQ, youtube.com/channel/UCl8dMTqDrJQ0c8y23UBu4kQ, youtube.com/channel/UCOpNcN46UbXVtpKMrmU4Abg, youtube.com/channel/UClgRkhTL3_hImCAmdLfDE4g*
Taobao.com	*taobao.com, item.taobao.com, bbs.taobao.com, s.taobao.com, world.taobao.com*
Msn.com	*msn.com, prodigy.msn.com, u.msn.com, cn.msn.com, zone.msn.com*
Live.com	*login.live.com, mail.live.com, outlook.live.com, onedrive.live.com, account.live.com*
Qq.com	*qzone.qq.com, news.qq.com, house.qq.com, t.qq.com, mail.qq.com*
Weibo.com	*weibo.com,s.weibo.com, open.weibo.com, e.weibo.com, desktop.weibo.com*
Bing.com	*bing.com, bing.com/maps, bing.com/videos, bing.com/images, bing.com/knows*
Yandex.ru	*yandex.ru, mail.yandex.ru, market.yandex.ru, metrika.yandex.ru, news.yandex.ru*
Yahoo.com	*mail.yahoo.com, search.yahoo.com, yahoo.com, news.yahoo.com, login.yahoo.com*
Mail.ru	*e.mail.ru, my.mail.ru, go.mail.ru, mail.ru, news.mail.ru*
Ebay.com	*ebay.com, my.ebay.com, k2b-bulk.ebay.com, signin.ebay.com, payments.ebay.com*

be implemented by deleting directly the *nonce* attribute values of HTML elements after the web browser has checked them.

Policies (vii), (viii), and (ix) can be implemented by disabling one or more JavaScript objects in the web browser. The traditional methods to disable JavaScript objects in web sandboxes such as rewriting, wrappers, and filters [28] cannot work well in PMHJ. We instrument the native code of JavaScript objects. When a JavaScript object is called the instrumented code decides whether it is allowed by the PMHJ policy of the web page where the JavaScript object is invoked. The 44 JavaScript objects defined by the set $U \cup J \cup H$ are instrumented with a total of 1390 lines of code. Some JavaScript objects such as *ActiveXObject* which are not supported by the original WebKit are introduced (actually empty functions with the same name) for the sake of test and evaluation. Policy (x) can directly utilize the strict mode for each block of script in the JavaScript engine. The implementation of policy (xi) can refer to the report mechanism of policy violation in CSP.

To sum up, the implementations of PMHJ in modern web browsers such as WebKit only need a little change since most security infrastructures have been well built after supporting CSP. And our implementation is just used as a proof-of-concept implementation without a poof of the safety impact on other parts of web browsers, which is a difficult issue. However, it is enough for illustration and testing.

At last, we introduce an easy and convenient method to design the PMHJ policy automatically. Given a web page, let the directives *allobject* and *report-$URL* be the PMHJ policy. Then, via the policy violation reports, we can get all the unused JavaScript objects which need to be disabled. In a similar way, by the most strict policy $P = \{$*allelement*, *allobject*, *string2js*, *htmlstrict*, *string2html*, *htmlstrict*, *scriptstrict*, *report-$URL*$\}$, we can get the HTML elements which need to add a *nonce* attribute and the unsafe usages of HTML and JavaScript which need to be replaced by the safe usages. It removes the heavy burden on the developer for specifying the right policy by hand which is prone to errors.

7. Evaluation

To evaluate the usability, we implement PMHJ on one hundred web pages from Alexa top 20 websites (the most popular five web pages are chosen from each website). All HTML and JavaScript code is considered including the third-party code in the web page. The tested web pages are listed in Table 3, and a part of the test results of ten web pages are shown in Figures 2 and 3. Combining with the experiment results, the PMHJ policies in real-world web applications are evaluated below.

In fact PMHJ is an access control mechanism supported by both web applications and web browsers to provide a second-line of defense against malicious HTML and JavaScript code in vulnerable web applications. The goal of PMHJ is not to eliminate or resolve vulnerabilities directly. It distinguishes the developer's HTML elements from the attacker's HTML elements or the damaged HTML elements via the element directive, the nonce directive, and the hash

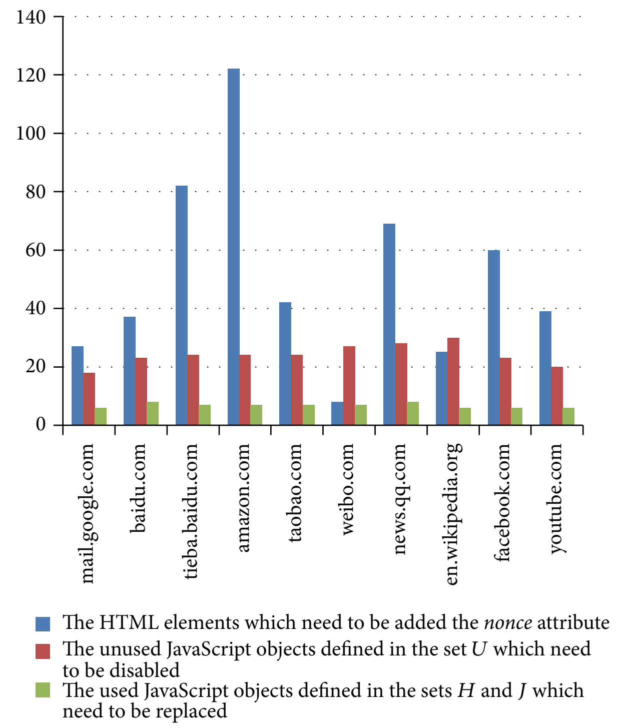

Figure 2: The number of HTML elements and JavaScript objects involved in PMHJ.

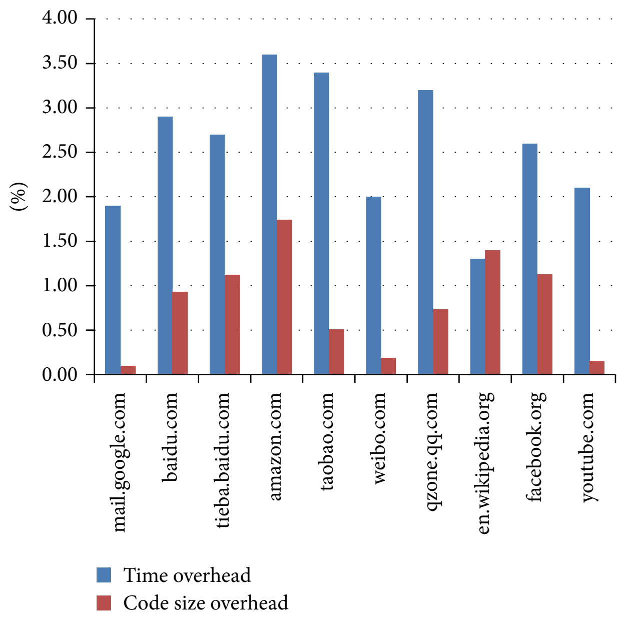

Figure 3: The overhead of time and code size in web pages using PMHJ.

directive. It avoids vulnerabilities by disabling the unsafe usages that may incur vulnerabilities via the *string2html* directive, the *string2js* directive, and the *scriptstrict* directive. It mitigates attacks by defining which HTML or JavaScript usages in the web page should be allowed or forbidden by web browsers via the *htmlstrict* directive, the object directive, and the multiple-object directive.

Considering the compatibility with the existing web platform, the web browser supporting PMHJ parses the web page without using PMHJ in the original way, and the web page using PMHJ can still run normally on the browser without supporting PMHJ where the PMHJ policy is ignored. As proved in Section 5, even simple PMHJ policies used in web pages are able to effectively strengthen the security. Thus, PMHJ can be gradually adopted by web applications and web browsers, and the whole time proves beneficial to adoptive web pages.

The test results show that the number of unused JavaScript objects defined in the set U which need to be disabled by multiple-object directives and object directives ranges from 17 to 31 (25.1 on average), and the number of used JavaScript objects defined in the set J and H which need to be disabled and replaced by more safe usages ranges from 2 to 8 (5.8 on average), as shown in Figure 2. This result indicates that there is only a few JavaScript objects defined in the set U used in web pages meaning that exposing them to attackers is unnecessary and dangerous. It also shows that PMHJ is urgently needed in real-world web applications since the unsafe usages of HTML and JavaScript are popular. For example, the event handler attribute is used in 93 percent of the test web pages and the insecure JavaScript object *document.write* is used in 81 percent of the test web pages.

To employ PMHJ (especially for a strict policy), it indeed needs a lot of work to change the original code of HTML and JavaScript, which mainly includes adding the nonce and hash attributes and replacing the disabled HTML and JavaScript usages. But, combining the strictest policy and the policy violation reports discussed in Section 6, the PMHJ policy for a given web page can be semiautomatically obtained by changing the policy or the code step by step. The whole process is not complex such as adding a nonce attribute for the protected element. For example, the test result shows that the number of HTML elements which need to be added to the *nonce* attributes ranges from 15 to 176 (49.1 on average). That is, the PMHJ policy is flexible and easy to be deployed into web applications.

The code size of web pages is a major concern for performance especially for mobile devices with limited storage capacity and for web applications where resources are transferred over the network in general. The test result shows that the overhead of code size ranges from 0.09% to 1.8% (0.7% on average); see Figure 3 for more details. The results are mainly affected by the size of original code since the code size added by PMHJ is too small compared with the original code.

The web page using PMHJ has little impact on the rendering efficiency. According to the test results, compared with the original web page and browser, only the web page or the web browser using PMHJ has a little influence on run time (the rendering time from a URL to the visual image) which can be ignored. And both the web page and web browser using PMHJ result in the run time increase by 1.1% to 4.9% (less than 2.6% on average); see more details in Figure 3. PMHJ has a much better performance in implementation than the test result of CSP [24] which needs to move all inline scripts into external scripts. It needs a further study on how the rendering time is affected by PMHJ due to different methods involved in PMHJ (complex HTML or JavaScript usages for different web pages which may interact together). In our experiments, the machine configurations are 2.70 GHz processor and 4 GB of memory. We take the average result in three times for each web page because of being easily affected by the network.

8. Related Work

BEEP [20] (Browser-Enforced Embedded Policies) computes the hash value of each script block in the web page as a whitelist. The whitelist will be responded with the web page to the browser. Then the related JavaScript functions will be rewritten on the browser side, which makes the hash value of each script block will be computed. The code of the script block will not run if the hash value is not in the whitelist. BEEP only protects JavaScript code in the web page and has an obvious efficiency problem when the protected elements increase because of computing the hash values. The *nonce* attribute used in PMHJ can effectively avoid these problems.

SOMA [10] (the Same Original Mutual Approval Policy) restricts resource inclusions in web pages by requiring approval from both the target site and resource provider. A whitelist of the resource servers which the browsers are enable to request is provided by the web page. When the browser requests the resource from the server in the whitelist, the resource server replies with "yes" or "no" according to its policy. Then the browser decides whether or not to allow requests by checking the results of the yes/no queries. SOMA increases a lot of the network communication between browsers and resource providers, which will greatly reduce the rendering efficiency of web pages.

Noncespaces [29] uses XML namespaces to help web browsers to distinguish between trusted and untrusted content in order to prevent cross-site scripting attack. A web application using Noncespaces randomizes the XML namespace prefixes of tags in each document before delivering it to the browser. As long as the attacker is unable to guess the random prefix value, the browser can distinguish between trusted content created by the web application and untrusted content provided by an attacker. Noncespaces is limited to XHTML documents, so other document types not based on XML may not benefit from this technique.

DSI [19] is a client-server architecture that enforces document structure integrity in a way to provide robust XSS defense with no false positives with a minimal effort from the web developer. It models XSS as a privilege escalation vulnerability as opposed to a sanitization problem and employs parser-level isolation for confinement of user-generated data throughout the lifetime of the web application. DSI is an ideal solution of XSS which lacks availability in practice since it relies on a complex implementation on the server side and browser side involving too many difficult problems to ensure security.

Blueprint [26] presents a system for parsing document content using a trusted and cross-platform JavaScript parser, rather than built-in parsers of web browsers. It views HTML parsers in different browsers as untrustworthy because of browser quirks and views the cross-site scripting problem as fundamentally arising from this. It provides the browser with a blueprint of the structure of the page, and a JavaScript library builds the page from the blueprint rather than trusting the browser's HTML parser. Blueprint has significant performance problems, which are inherent to its approach in avoiding use of the browser's parser.

CONTEGO [30] (Capability-Based Access Control) demonstrates that the security problem of the web page is caused by the lack of access control on capability and proposes an access control model based on the capability to prevent against malicious HTML and JavaScript code. The capability of security risks is divided into eleven categories and is confined on single HTML element. However, this kind of fine-grained access control is short of usability so that it cannot be widely applied to the existing web applications.

DCS [11] (Data-Confined Sandbox) gives a solution for the security problem caused by complex communication channels in the web platform. It strictly manages almost all the communication channels of web pages on the browser side. However, DCS depends on the fact that the web application employs a design similar to the process management of operating systems; that is, all communication is mediated by a web page like the master process. This design is not suitable for all web applications and not easy to be adopted by developers.

CSP has been supported by almost all web browsers and is treated as the most promising security mechanism for web applications. The analysis and enhancement of CSP are currently a hot research topic. Weinberger et al. [24] evaluate the efficacy of the large scale web applications using CSP by retrofitting Bugzilla and HotCRP. Weissbacher et al. [31] and Patil and Frederik [13] try to explain the trends and challenges of CSP adoption in real-world web applications. The CSP suborigin for each web page is proposed by Akhawe et al. [32]. It is actually a strengthening and development of the same origin policy and the HTML sandbox and has been paid great attention by the Chromium project team. Hanna et al. [33] have found a serious flaw: the insecure use of *postMessage* in Facebook Connect and Google Friend Connect leads to severe vulnerabilities to increase the attack surface for web applications in unexpected ways. It can be solved by extending CSP with a directive to specify origins allowed to send messages to the web page. In order to solve the problem of insecure server side assembly of JavaScript code, Johns [34] proposes a solution as a supplement of CSP by combining the prepared templates of JavaScript with the stable cryptographic checksums for scripts. Compared with these work, PMHJ is the first systematic analysis and enhancement of CSP against malicious HTML and JavaScript code.

There are other efforts to help developers to understand or design the CSP in real-world web applications. Veracode [13] and Chen et al. [35] publish statistical reports of security headers certainly including CSP used in Alexa top 1 million websites. Javed [36] gives an automated aiding system for the construction of CSP policies in web applications named AiDer. Unfortunately, AiDer cannot recognize dynamic changes of DOM. Patil et al. [37] propose UserCSP, namely, a Firefox extension, that uses dynamic analysis to automatically infer CSP policies and gives savvy users the authority to enforce client-side policies on web pages. However, UserCSP is error-prone to rely on a self-design of CSP with complex client logic and is limited to the specified version of Firefox.

Aside from the content protection systems above, various mechanisms have been proposed to lock down the capacity of JavaScript APIs for different goals. Due to the novel but vulnerable HTML5 APIs such as *postMessage* [33] and the lack of proper defense for these APIs, web browsers are in trouble between defining a powerful JavaScript API and limiting the ability of malicious code. A solution in use is that a call of high-risk JavaScript APIs must be authorized by the user. Several proposals have developed new primitives for web applications to adapt to mash-up applications [38, 39]. However, the widely used tools to isolate untrusted third-party content are web sandboxes [40] such as Google Caja [41] by limiting the JavaScript APIs which violate the isolation policy. PMJA focuses on the least authority of high-risk JavaScript APIs instead of paying attention to the security of a single HTML5 API or the goal of isolation.

9. Future Work

PMHJ has been systemically presented in this paper to provide a robust protection against malicious HTML and JavaScript code, but extensions to PMHJ can be created to increase its usability or add additional protections.

It is an important work to well combine PMHJ and CSP such as how to resolve the policy conflicts that may be caused by using PMHJ and CSP in the same web page. The PMHJ policy with a higher priority to the CSP policy is recommended due to the fact that the PMHJ policy is stricter than the CSP policy in methodology, when they are conflicting. Considering that CSP has been applied to Chrome Extensions and Chrome Apps, it is valuable to apply PMHJ to these new types of HTML5 applications.

It may also prove useful for web applications to verify more HTML elements such as *a* and *div*. While these elements would not restrict any content types embedded on the page, they still dictate behavior of the site and may be desirable for many existing web applications.

More issues in real-world deployment should be taken into consideration such as the encoding of the nonce and hash attribute value. It is a heavy but meaningful work to discuss and prove the security properties of the browser implementation for PMHJ or CSP.

To aid in constructing a proper PMHJ policy for complex web pages, a more automatic tool such as UserCSP for CSP should be created. It can create a policy based on the expected behavior of the web page via a static or dynamic analysis, and any deviations from the policy can be reported.

Since PMHJ and CSP cannot thoroughly prevent some value-level attacks, it is our future work to give a client-side sanitation mechanism combining with the basic ideas of PMHJ and DSI. It requires a progress not only in engineering but in theory.

10. Conclusions

PMHJ not only is a systemic enhancement of CSP but also works independently to provide a strong and comprehensive protection against malicious HTML and JavaScript code in vulnerable web applications. It solves the problems of malicious injection attacks and node-split attacks of HTML elements, systemically disables the unsafe HTML usages to make the content of the web page able to be loaded only in a safe way, and provides a method to limit the insecure JavaScript APIs incurring injection vulnerabilities and allow developers to flexibly rein high-risk JavaScript APIs with powerful ability in each web page. PMHJ is easy to be deployed into web applications with a small code size overhead and has little influence on the rendering efficiency.

Competing Interests

The authors declare that there is no competing interests regarding the publication of this paper.

Acknowledgments

This work is supported by the 863 National High Technology Research and Development Program of China (no. 2012AA012902).

References

[1] X. Li and Y. Xue, "A survey on server-side approaches to securing web applications," *ACM Computing Surveys*, vol. 46, no. 4, article 54, 2014.

[2] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov, *XSS Attacks: Cross Site Scripting Exploits and Defense*, Syngress, 2011.

[3] N. Nikiforakis, L. Invernizzi, A. Kapravelos et al., "You are what you include: large-scale evaluation of remote Javascript inclusions," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*, pp. 736–747, ACM, Raleigh, NC, USA, October 2012.

[4] H. Shahriar, K. Weldemariam, M. Zulkernine, and T. Lutellier, "Effective detection of vulnerable and malicious browser extensions," *Computers & Security*, vol. 47, pp. 66–84, 2014.

[5] I. Hydara, A. B. M. Sultan, H. Zulzalil, and N. Admodisastro, "Current state of research on cross-site scripting (XSS)—a systematic literature review," *Information and Software Technology*, vol. 58, pp. 170–186, 2015.

[6] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with BEK," in *Proceedings of the 20th USENIX Conference on Security (SEC '11)*, p. 1, USENIX Association, Berkeley, Calif, USA, 2011.

[7] J. Weinberger, P. Saxena, D. Akhawe et al., "A systematic analysis of XSS sanitization in web application frameworks," in *Computer Security—ESORICS 2011*, vol. 6879 of *Lecture Notes in Computer Science*, pp. 150–171, Springer, Berlin, Germany, 2011.

[8] T. Scholte, D. Balzarotti, and E. Kirda, "Have things changed now? An empirical study on input validation vulnerabilities in web applications," *Computers & Security*, vol. 31, no. 3, pp. 344–356, 2012.

[9] S. Gupta and B. B. Gupta, "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, pp. 1–9, 2015.

[10] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, "SOMA: mutual approval for included content in web pages," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*, pp. 89–98, ACM, October 2008.

[11] D. Akhawe, F. Li, W. He et al., "Data-confined HTML5 applications," in *Computer Security—ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9–13, 2013. Proceedings*, vol. 8134 of *Lecture Notes in Computer Science*, pp. 736–754, Springer, Berlin, Germany, 2013.

[12] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International World Wide Web Conference (WWW '10)*, pp. 921–929, Raleigh, NC, USA, April 2010.

[13] K. Patil and B. Frederik, "A measurement study of the content security policy on real-world applications," *International Journal of Network Security*, vol. 18, no. 2, pp. 383–392, 2016.

[14] Veracode, "Security Headers on the Top 1,000,000 Websites: October 2014 Report," October, 2014, https://www.veracode.com/blog/2014/10/security-headers-top-1000000-websites-october-2014-report.

[15] M. West, A. Barth, and D. Veditz, *Content Security Policy Level 2: W3C Candidate Recommendation*, 2015, http://www.w3.org/TR/CSP2/.

[16] Z. C. Schreuders, *Functionality-based application confinement [Ph.D. thesis]*, Murdoch University, Perth, Australia, 2012.

[17] A. Zarras, A. Kapravelos, G. Stringhini et al., "The dark alleys of madison avenue: understanding malicious advertisements," in *Proceedings of the ACM Conference on Internet Measurement Conference (IMC '14)*, pp. 373–380, Vancouver, Canada, November 2014.

[18] M. Finifter, J. Weinberger, and A. Barth, "Preventing capability leaks in secure javascript subsets," in *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS '10)*, 2010.

[19] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: a robust basis for cross-site scripting defense," in *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS '09)*, San Diego, Calif, USA, February 2009.

[20] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th International World Wide Web Conference (WWW '07)*, pp. 601–610, Bnaff, Canada, May 2007.

[21] M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications*, No Starch Press, 2012.

[22] 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)*, pp. 961–970, ACM, 2009.

[23] D. Flanagan, *JavaScript: The Definitive Guide*, O'Reilly Media, 2012.

[24] J. Weinberger, A. Barth, and D. Song, "Towards client-side HTML security policies," in *Proceedings of the 6th USENIX Conference on Hot Topics in Security (HotSec '11)*, 2011.

[25] H. Gilbert and H. Handschuh, "Security analysis of SHA-256 and sisters," in *Selected Areas in Cryptography*, M. Matsui and R. J. Zuccherato, Eds., vol. 3006 of *Lecture Notes in Computer Science*, pp. 175–193, Springer, Berlin, Germany, 2004.

[26] M. T. Louw and V. N. Venkatakrishnan, "Blueprint: robust prevention of cross-site scripting attacks for existing browsers," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pp. 331–346, IEEE, Berkeley, Calif, USA, May 2009.

[27] Y. Zhu, *The Insider WebKit Technology*, Publishing House of Electronics Industry of China, Beijing, China, 2014.

[28] S. Maffeis, J. C. Mitchell, and A. Taly, "Isolating JavaScript with filters, rewriting, and wrappers," in *Computer Security—ESORICS 2009*, M. Backes and P. Ning, Eds., vol. 5789 of *Lecture Notes in Computer Science*, pp. 505–522, Springer, Berlin, Germany, 2009.

[29] M. Van Gundy and H. Chen, "Using randomization to enforce information flow tracking and thwart cross-site scripting attacks," in *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS '09)*, San Diego, Calif, USA, February 2009.

[30] T. Luo and W. Du, "Contego: capability-based access control for web browsers," in *Trust and Trustworthy Computing*, pp. 231–238, Springer, Berlin, Germany, 2011.

[31] M. Weissbacher, T. Lauinger, and W. Robertson, "Why is CSP failing? trends and challenges in CSP adoption," in *Research in Attacks, Intrusions and Defenses*, vol. 8688 of *Lecture Notes in Computer Science*, pp. 212–233, Springer, Berlin, Germany, 2014.

[32] D. M. Akhawe, *Towards high assurance HTML5 applications [Ph.D. dissertation]*, University of California Berkeley, Berkeley, Calif, USA, 2014.

[33] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song, "The Emperor's new APIs: on the (In)secure usage of new client-side primitives," in *Proceedings of the 4th Web 2.0 Security and Privacy Workshop (W2SP '10)*, Oakland, Calif, USA, May 2010.

[34] M. Johns, "Script-templates for the content security policy," *Journal of Information Security and Applications*, vol. 19, no. 3, pp. 209–223, 2014.

[35] P. Chen, N. Nikiforakis, L. Desmet, and C. Huygens, "Security analysis of the Chinese web: how well is it protected?" in *Proceedings of the Workshop on Cyber Security Analytics, Intelligence and Automation*, pp. 3–9, ACM, November 2014.

[36] A. Javed, "Csp aider: an automated recommendation of content security policy for web applications," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, Calif, USA, May 2011.

[37] K. Patil, T. Vyas, F. Braun, M. Goodwin, and Z. Liang, *Poster: UserCSP—User Specified Content Security Policies*, SOUPS, 2013.

[38] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang, "Towards fine-grained access control in JavaScript contexts," in *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS '11)*, pp. 720–729, Minneapolis, Minn, USA, July 2011.

[39] L. Ingram and M. Walfish, "Treehouse: javascript sandboxes to help web developers help themselves," in *Proceedings of the USENIX Conference on Annual Technical Conference (USENIX ATC '12)*, pp. 153–164, 2012.

[40] J. G. Politz, S. Eliopoulos, A. Guha et al., "ADsafety: type-based verification of JavaScript Sandboxing," in *Proceedings of the 20th Conference on Security (USENIX '11)*, p. 12, San Francisco, Calif, USA, August 2011.

[41] M. S. Miller, M. Samuel, B. Laurie et al., "Safe active content in sanitized JavaScript," Tech. Rep., Google, 2008.