RESEARCH FEATURE

Mitigating Cross-Site Scripting Attacks with a Content Security Policy

Imran Yusof and Al-Sakib Khan Pathan, International Islamic University Malaysia

A content security policy (CSP) can help Web application developers and server administrators better control website content and avoid vulnerabilities to cross-site scripting (XSS). In experiments with a prototype website, the authors' CSP implementation successfully mitigated all XSS attack types in four popular browsers.

Among the many attacks on Web applications, cross-site scripting (XSS) is one of the most common. An XSS attack involves injecting malicious script into a trusted website that executes on a visitor's browser without the visitor's knowledge and thereby enables the attacker to access sensitive user data, such as session tokens and cookies stored on the browser.[1] With this data, attackers can execute several malicious acts, including identity theft, keylogging, phishing, user impersonation, and webcam activation.

Even major application services such as Facebook, Google, PayPal, and Twitter suffer from XSS attacks, which have grown alarmingly since they were first reported in a 2003 Computer Emergency Response Team advisory. The Open Web Application Security Project ranked XSS third on its 2013 list of top 10 Web vulnerabilities (the latest list as of February 2016), calling it the "most prevalent Web application security flaw."[2] Underscoring the widespread risk of XSS intrusions, WhiteHat Security's May 2013 *Web Security Statistics Report* noted that 43 percent of Web applications were vulnerable to this kind of attack (www.whitehatsec.com/assets/WPstatsReport_052013.pdf).

Researchers have proposed a range of mechanisms to prevent XSS attacks, with content sanitizers dominating those approaches. Although sanitizing eliminates potentially harmful content from untrusted input, each Web application must manually implement it—a process prone to error. To avoid this problem, we use a different technique. Instead of sanitizing harmful scripts before they are injected into a website, we block them from loading and executing with a variation of the *content security policy* (CSP), which provides server administrators with a white list of accepted and approved resources. The Web application or website will block any input not on that list and thus there is no need for sanitizing. The white list also guards against data exfiltration and extrusion—the unauthorized downloading of data from a website visitor's computer.

Our variation of CSP 1.0, a World Wide Web Consortium (W3C) standard, uses directives such as `report-uri`, which lets server administrators either save policy

0018-9162/16/$33.00 © 2016 IEEE

violations to a log file or receive them as email. In the report-only mode, this directive lets administrators conduct a dry run of the website with a particular CSP and note XSS attack types—information that can be shared in the security community to inform both practical case implementations and research projects.

Existing defense mechanisms tend to focus on preventing one or two of the three XSS attack types, but our CSP is the first that we know of to mitigate all three. To test its effectiveness, we created a prototype website on a simple Web application server, configured a CSP header with a `report-uri` directive, and incorporated the header in an .htaccess file (a configuration file that specifies how a webpage should be accessed). We then conducted a series of experiments on our local host machine by injecting XSS vectors into the website. In every case, our CSP prevented XSS attacks, even with 50 unique XSS vectors.

TYPES OF CROSS-SITE SCRIPTING ATTACKS

An XSS attack can be persistent or nonpersistent, or it can be based on a document object model (DOM).

Persistent XSS

Figure 1 shows a persistent XSS attack, also known as a stored XSS or Type-I XSS attack. This attack type involves injecting malicious script into a website, which stores the script in its database.[3] If the script is not correctly filtered, it will appear to be a part of the Web application and run within a user's browser under the application's privileges. A persistent XSS attack does not need a malicious link for successful exploitation; simply visiting the webpage will compromise the user.

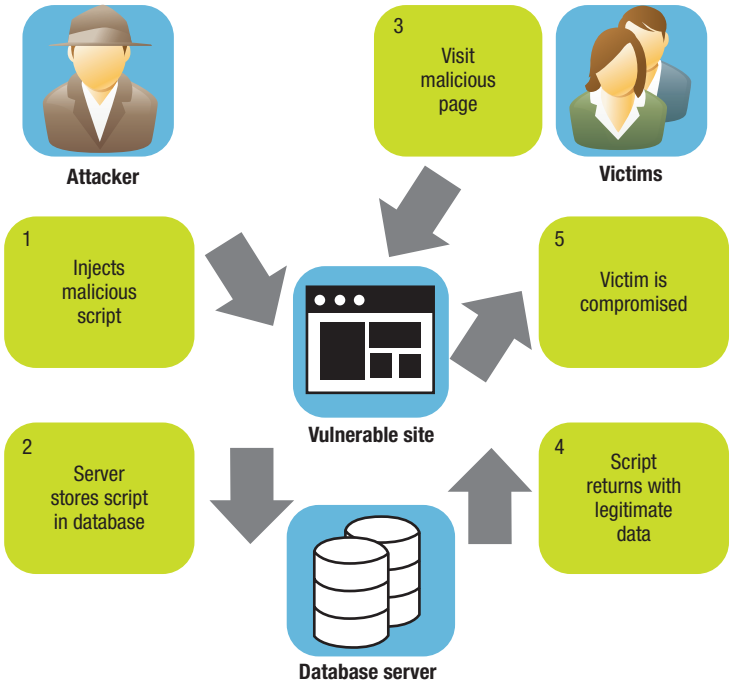


FIGURE 1. Typical scenario of a persistent cross-site scripting (XSS) attack. Each time a user visits a webpage injected with malicious script, the stored script exploits the user's browser privileges to access sensitive information.

Persistent XSS is often difficult to detect and is considered more harmful than the other two attack types. Because the malicious script is rendered automatically, there is no need to target individual victims or lure them to a third-party website. Consequently, attackers can easily hide their activity; for example, in a blog, they could embed the script in a seemingly innocuous comment. All visitors to that site would then unknowingly put their browser—and the sensitive data stored on it—at risk.

Nonpersistent XSS

Figure 2 shows a nonpersistent, or reflected, XSS attack,[3] which occurs when a website or Web application passes invalid user inputs. Usually, an attacker hides malicious script in the URL, disguising it as user input, and lures victims by sending emails that prompt users to click on the crafted URL. When they do, the harmful script executes in the browser, allowing the attacker to steal authenticated cookies or data. In the figure, we assume that victims have authenticated themselves at the vulnerable site.

DOM-based XSS

A webpage is composed of various elements, such as forms, paragraphs, and tables, which are represented in an object hierarchy. To update the structure and style of webpage content dynamically, all Web applications and websites interact with the DOM, a virtual map that enables access to these webpage elements. Compromising a DOM will cause the client-side code to execute in an unexpected manner.

A DOM-based, or Type-0, XSS attack executes in the same manner as a nonpersistent XSS attack except for step 3 in Figure 2.[3] In a DOM-based attack, rather than having the server carry the malicious payload in its HTTP response, the attacker encodes a malicious value in a URL and sends it to the victim. The attack occurs when the victim's browser executes the malicious code from the modified DOM. On the client side, the HTTP response does not change but the script executes maliciously. This exploit works

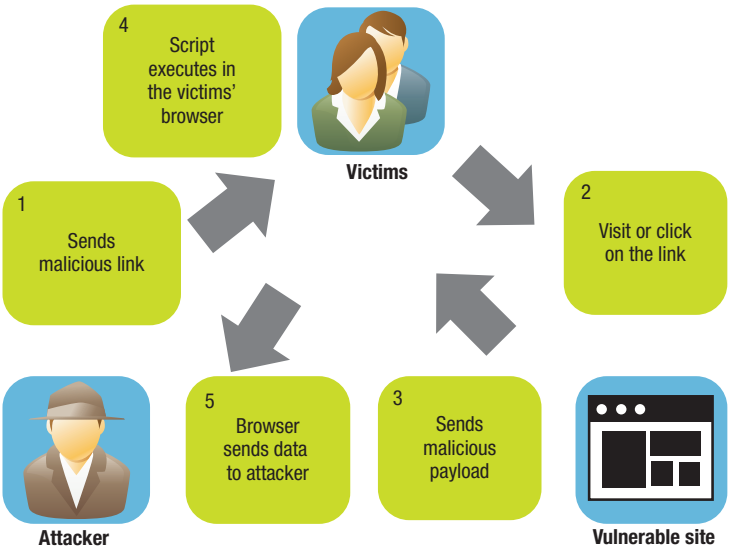


FIGURE 2. Typical scenario of a nonpersistent XSS attack. Victims authenticate themselves at the site and the attacker lures them into loading a malicious link. The link then executes malicious code with the user's credentials.

only if the browser does not modify the URL characters.[3]

A DOM-based XSS attack is the most advanced type and is not well known. Indeed, much of the vulnerability to this attack type stems from the inability of Web application developers to fully understand how it works.

UNDERSTANDING CONTENT SECURITY POLICIES

CSPs are not a new idea. Initially developed by the Mozilla Foundation, the CSP concept was first applied in Firefox 4 in 2011 and became a W3C candidate recommendation in November 2012 (www.w3.org/TR/2012/CR-CSP-20121115). Written by a Web application developer or server administrator, a CSP specifies how content interacts on the website so that the browser knows what website resources it is permitted to execute and render. The motivation behind having CSPs is to mitigate XSS attacks without the need to substantially modify the application's source code.

A CSP plugs the loophole of a same origin policy (SOP). In a typical Web-application security model, an SOP permits scripts running on pages from the same site; the page is typically specified as protocol, hostname, and port number. For example, in "http://www.example.com:8000," the protocol is http, the hostname is www.example.com, and the port is 8000. The SOP will consider only this combination as a legitimate origin, and pages with the same combination can access each other's DOMs with no restrictions. The SOP is so named because it trusts any content from the same origin—whatever the server from that source delivers. It is also the loophole the attacker can exploit by injecting malicious code into the legitimate, SOP-trusted page.

A CSP closes this loophole through its white list, which guides the browser to execute only the listed resources. Thus, even if the attacker finds a way to inject a script into the trusted origin, it would not match the resources and content in the white list and would therefore be rejected.

How a CSP works

A browser that employs a CSP follows only the CSP's *directives*—language constructs that specify how a compiler (or assembler or interpreter) should process its input. By default, a CSP prohibits the use of inline script on the website, such as this inline XSS attack payload (all payload and vector examples are in JavaScript):

```
><img src="%#" onerror="alert
   (1)">X</a>
```

Also, a CSP prevents the use of JavaScript's `eval` family of functions. An example of an `eval` XSS attack payload is

```
<div id=confirm(1) onmouseover=
   eval(id)>X</div>
```

Figure 3a shows a typical CSP header, which consists of directives, URIs, and keywords. Any element that violates the specified policy is blocked. Thus, even if an attacker manages to inject malicious script into the CSP-protected website, the client-side browser blocks script execution, as Figure 3b shows.

Browser support

Table 1 shows all of the browser versions that currently support CSPs (http://caniuse.com/#feat=contentsecurity policy). According to the W3C specification, the Content-Security-Policy header is the standard header and is used by most later browser versions, including Firefox version 23 and later, Chrome version 25 and later, Safari version 7 and later, and iOS Safari version 7.1 and later.

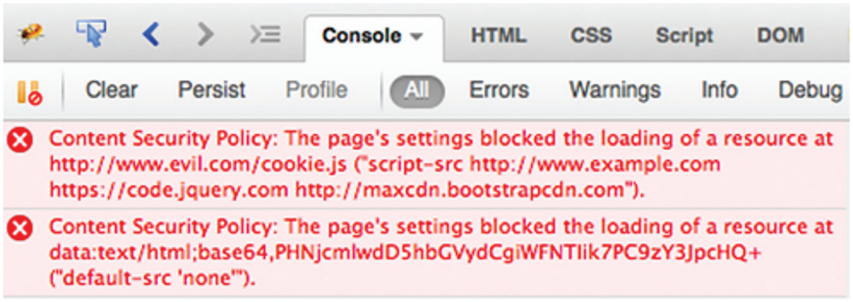
The X-Content-Security-Policy and X-WebKit-CSP headers are used mainly for earlier browser versions. To ensure that all of the browser versions read the header, the server administrator can configure both the X-Content-Security-Policy and Content-Security-Policy headers; the browser will always choose the policy in the Content-Security-Policy header if it supports the standard header.

Source directives

CSP source directives control how a client-side browser should behave when it comes across various types of

```
Content-Security-Policy: “default-src ‘none’; script-src ‘self’
   https://code.jquery.com maxcdn.bootstrapcdn.com; style-src ‘unsafe-inline’
   maxcdn.bootstrapcdn.com;img-src ‘self’; connect-src ‘self’; media-src
   ‘self’; font-src ‘self’ maxcdn.bootstrapcdn.com; report-uri http://www
   .example.com/csp_report.php”
```

(a)



(b)

FIGURE 3. Blocking malicious script with a content security policy (CSP). (a) A sample CSP header, which is a set of directives, URIs, and keywords that tells the browser what it is allowed to load and execute on the website. (b) The website blocks the script that violates the directives in the CSP header.

protected website content—from JavaScript to connection locations. Of the source directives we chose to describe, the most common are default, script, and style.

Default source. Web application developers or server administrators use the default source, or `default-src`, directive to define the white list of resources. Sample policies using this directive are

```
Content-Security-Policy: default-
   src ‘self’
```

which permits client browsers to load all resources only from the Web application's own origin (protocol, hostname, and port number), and

```
Content-Security-Policy: default-
   src ‘none’
```

which specifies with the keyword `none` that no resource is allowed to load.

Script source. The `script-src` directive controls the loading of JavaScript on the website. The first part of the sample policy

```
Content-Security-Policy: default-
   src ‘none’; script-src script
   .example.com javascript.example
   .com
```

specifies a `default-src` of ‘none’. The second part permits the client browser to load script from script.example.com and javascript.example.com. The second part overwrites the `default-src` policy—that is, no resource (script) is permitted to load except from script.example.com and javascript.example.com.

Style source. The `style-src` directive controls the use of Cascading Style Sheets (CSS) and other styles on a webpage. The policy

```
Content-Security-Policy: default-
   src ‘none’; style-src ‘unsafe-
   inline’ maxcdn.bootstrapcdn.com
```

allows the use of inline style and the style sheets from bootstrapcdn.com only. It disallows the loading of any other sources, such as the connect, frame, and media sources.

Object source. The `object-src` directive defines the white list of sources from which the browser is allowed to download resources while constructing the <applet>, <object>, and <embed> tags. The sample policy

```
Content-Security-Policy: default-
   src ‘none’; object-src plugins
   .example.com applet.example.com
```

first defines a `default-src` of ‘none’ and then permits two valid locations where the object can be loaded.

Image source. The `img-src` directive specifies the domains from which images can be downloaded. In the sample policy

```
Content-Security-Policy: default-
   src ‘none’; img-src img.example
   .com images.example.com
```

images can be downloaded only from img.example.com and images.example.com; all other image-source domains will be blocked.

Media source. The `media-src` directive specifies what media can be downloaded. In the sample policy

```
Content-Security-Policy: default-
   src ‘self’; media-src video
   .example.com youtube.com
```

media sources are restricted to video.example.com and youtube.com; all other resources must come from the Web application's own origin.

Font source. The `font-src` directive specifies what fonts can be loaded. For example, the policy

```
Content-Security-Policy: default-
   src ‘self’; font-src ‘self’
   http://fonts.gstatic.com
```

TABLE 1. Browser versions that support content security policies (CSPs).

Header	Firefox	Chrome	Safari	Internet Explorer	iOS Safari	Android	Chrome Android
Content-Security-Policy	23+	25+	7+	—	7.1+	4.4+	47
X-Content-Security-Policy	4.0+	—	—	10+ (limited)*	—	—	—
X-WebKit-CSP	—	14+	6+	—	5.1+ (limited)*	—	—

*Supports only partial CSP

```
Content-Security-Policy: “default-src ‘none’; script-src ‘self’ https://code.jquery.com
   maxcdn.bootstrapcdn.com; style-src ‘unsafe-inline’ maxcdn.bootstrapcdn.com http://fonts.googleapis.com;
   img-src ‘self’; connect-src ‘self’; media-src ‘self’; font-src ‘self’ maxcdn.bootstrapcdn.com
   http://fonts.gstatic.com; report-uri http://www.example.com/csp_report.php”
```

(a)

```
{
  “csp-report”: {
    “blocked-uri”: “http://www.evil.com/cookie.js”,
    “document-uri”: “http://www.example.com/”,
    “original-policy”: “default-src ‘none’; script-src http://www.example.com https://code.jquery.com
http://maxcdn.bootstrapcdn.com; style-src ‘unsafe-inline’ http://maxcdn.bootstrapcdn.com
http://fonts.googleapis.com; img-src http://www.example.com; connect-src http://www.example.com; media-
src http://www.example.com; font-src http://www.example.com http://maxcdn.bootstrapcdn.com
http://fonts.gstatic.com; report-uri http://www.example.com/csp_report.php”,
    “referrer”: “”,
    “violated-directive”: “script-src http://www.example.com https://code.jquery.com
http://maxcdn.bootstrapcdn.com”
    }
}
```

(b)

FIGURE 4. Using the `report-uri` directive. (a) Sample CSP with `report-uri` directive and (b) violation report sent through HTTP Post in JavaScript Object Notation (JSON).

allows fonts from fonts.gstatic.com, but all other resources must come from the Web application's own origin.

Frame source. The `frame-src` directive defines the white list of sources from which the browser is allowed to render frames. The policy

```
Content-Security-Policy:
   default-src ‘self’;
   frame-src ‘self’youtube.com
   video.com
```

permits frames from youtube.com and video.com; all other resources must come from the Web application's own origin.

Connect source. The `connect-src` directive permits connections to the XMLHttpRequest, WebSocket, and EventSource locations. Any effort to open connections to other locations is prohibited. The policy

```
Content-Security-Policy:
   default-src ‘none’; connect-src
   xhr.example.com
```

permits outgoing connections to xhr .example.com and blocks any other resource.

Report-only mode with the report-uri directive

In report-only mode, the CSP tells the browser not to block a disallowed element. Although seemingly counterproductive, this mode along with the `report-uri` directive creates a dry run of a CSP configuration with a specified endpoint. Through the directive, the browser can report a policy violation or the existence of harmful content to the server during that run. The server then stores or processes the report for further action.

Figure 4a shows a sample CSP with the `report-uri` directive, which specifies where the browser should send violation reports; in this case, it is to the csp_report.php file. The reports are sent through HTTP Post in JavaScript Object Notation (JSON). Figure4b shows a possible violation report for the sample CSP.

Figure 5a shows PHP source code for saving policy violations to a `csp-report.log` file, and Figure 5b shows the actual violation report in JSON.

Figure 6 shows PHP code for sending the policy violation report to the email address suretarget@gmail.com, and Figure 7 shows the sent report. As the figures indicate, when a policy violation occurs, the report is sent to the server administrator. Other information, such as the attacker's user agent and IP address, would be also logged so that the administrator can take the appropriate follow-up actions.

EXPERIMENTAL EVALUATION

To test the effectiveness of our CSP, we created a prototype website that runs on an

```
1  <?php
2
3  $csp = file_get_contents('php://input');
4  $csp = json_decode($csp, true);
5  $msg .= json_encode($csp, JSON_PRETTY_PRINT |
6     JSON_UNESCAPED_SLASHES);
7
8  file_put_contents('csp-report.log', $msg,
      FILE_APPEND | LOCK_EX);
9 ?>
```

(a)

```
1 {
2  "csp-report": {
3     "blocked-uri": "http://www.evil.com/cookie.js",
4     "document-uri": "http://www.example.com",
5        "original-policy": "default-src 'none';
6        script-src http://www.example.com https://code.jquery.com
7              http://maxcdn.bootstrapcdn.com;
8           style-src 'unsafe-inline' http://maxcdn.bootstrapcdn.com
9                http://fonts.googleapis.com;
10          img-src http://www.example.com;
11          connect-src http://www.example.com;
12          media-src http://www.example.com;
13          font-src http://www.example.com
                http://maxcdn.bootstrapcdn.com
14           http://fonts.gstatic.com;
15       report-uri http://www.example.com/csp_report.php",
16       "referrer": "",
17       "violated-directive": "script-src http://www.example.com
18          https://code.jquery.com
19          http://maxcdn.bootstrapcdn.com"
20  }
21 }
```

(b)

FIGURE 5. Saving CSP violations. (a) Code to save violations and (b) violation report in a csp-report.log file. The report is an endpoint in the csp-report.log file; in line 3, the execution of cookie.js from evil.com is blocked because it has violated the specified CSP.

```
1 <?php
2
3 $recipient = "suretarget@gmail.com";
4 $sender    = $_SERVER['SERVER_ADMIN'];
5 $subject   = "CSP Violation Report";
6 $headers   = 'From: ' .
7              $_SERVER['SERVER_ADMIN']
                  . "\r\n" .
8              'Reply-To: ' .
9              $_SERVER['SERVER_ADMIN']
                  . "\r\n" .
10             'X-Mailer: PHP/'
                  .phpversion();
11
12 $csp      = file_get_contents
      ('php://input');
13
14 $user_agent = $_SERVER['HTTP_USER_
      AGENT'];
15 $ip_address = $_SERVER['REMOTE_ADDR'];
16
17 $msg = "The user agent: {$user_agent},
      IP address:
18     {$ip_address} reported the
          following content
19        security policy (CSP)
             violation:\n\n";
20
21 $csp = json_decode($csp, true);
22 $msg .= json_encode($csp,
      JSON_PRETTY_PRINT |
23       JSON_UNESCAPED_SLASHES);
24
25 mail($recipient, $subject, $msg,
      $headers);
26
27 ?>
```

FIGURE 6. PHP sample source code for emailing a CSP violation to a server administrator.

Apache webserver and permits untrusted HTML input through a user profile page. Figure 8 shows the CSP header we used to alter the webserver's configuration. A similar configuration can be set in any specific header page.

Because the `default-src` directive is `'none'`, the CSP disallows all resources to load except white list and trusted resources. Any policy violation invokes the `report-uri` directive.

We used a collection of XSS cheat sheets from previous work[3] and 50 unique XSS vectors to simulate the

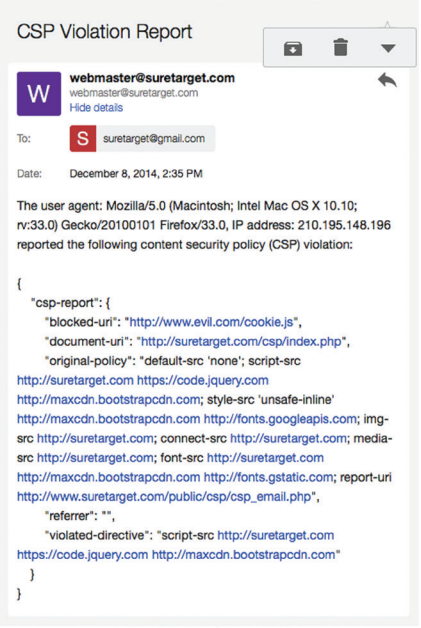


FIGURE 7. Sample email to notify the server administrator of a CSP violation. Additional information is included so that the administrator can learn more about the violation.

three different attack types, conducting one simulation with the 50 XSS vectors for each of four browsers: Chrome v40.0.2214.115 (64-bit), Firefox v35.0.1, Safari v8.0.2 (10600.2.5), and Opera v27.0.1689.69 in OS X Yosemite 10.10.

Figure 9 summarizes the simulation results before and after applying our CSP. As we expected, CSP protection eliminated the XSS attack vectors in all the simulations, blocking a range of attack types, including XSS attacks originating from the server. Before we applied the CSP, 37 XSS attacks were successful in Firefox while 19 XSS attacks got by Chrome, Safari, and Opera—even with their built-in XSS auditors.

RELATED WORK IN XSS ATTACK PREVENTION

Others have proposed mechanisms to prevent XSS attacks.[4] Noxes, a client-side tool that acts as a Web proxy, disallows requests that do not belong to the website and thus thwarts stored XSS attacks.[5]

Browser-enforced embedded policies (BEEPs) let the Web application developer embed a policy in the website by

```
<IfModulemod_headers.c>
  Header set Content-Security-Policy: “default-src ‘none’;
  script-src ‘self’ https://code.jquery.com maxcdn.bootstrapcdn.com; style-src
  ‘unsafe-inline’ maxcdn.bootstrapcdn.com http://fonts.googleapis.com; img-src ‘self’;
  connect-src ‘self’; media-src ‘self’; font-src ‘self’maxcdn.bootstrapcdn.com http://
  fonts.gstatic.com; report-urihttp://www.example.com/csp_report.php”
</IfModule>
```

FIGURE 8. CSP header in our prototype website. We wrote the header in a syntax compatible with an .htaccess file and used it to modify the Apache webserver.

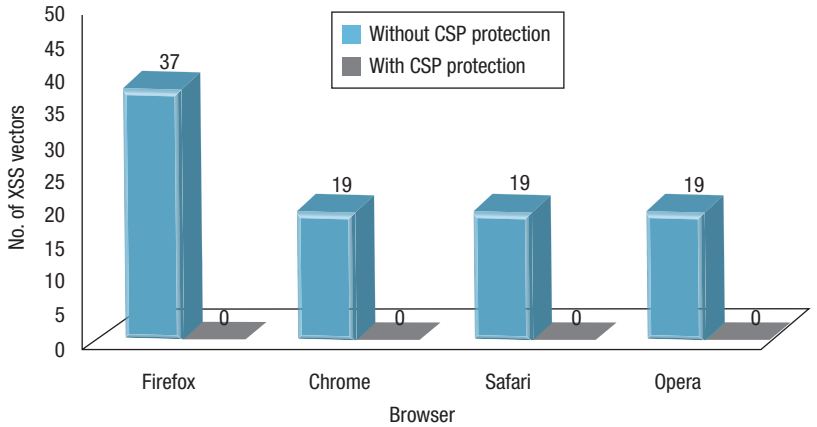


FIGURE 9. Test results. Without CSP protection, as many as 37 XSS vectors were successful (Firefox). Even the XSS auditor in Chrome, Safari, and Opera could not eliminate all XSS vectors. However, applying CSP protection eliminated all XSS vectors for each browser.

specifying which scripts are allowed to run.[6] With a BEEP, the developer can put genuine source scripts in a white list and disable source scripts in certain website regions.

Document Structure Integrity (DSI) is a client-server architecture that restricts the interpretation of untrusted content.[7] DSI uses parser-level isolation to isolate inline untrusted data and separates dynamic content from static content. However, this approach requires both servers and clients to cooperatively upgrade to enable protection.

Blueprint is a server-side application that encodes content into a model representation that the client-side part can process. However, applying Blueprint to Wordpress increased processing time on average 55 percent; applying it to MediaWiki increased processing time an average 35.6 percent.[8]

Implementing our CSP can help Web application developers specify allowable content type and resource locations and can be an early warning system for any policy violations, which greatly assists system administrators' website control.

With little or no modification to application source code, website visitors are assured of protection from the unauthorized downloading of the sensitive data stored in their browsers. The CSP's report-only mode along with the `report-uri` directive gives server administrators the option to test and configure their applications without breaking website functionalities.

Although our CSP has many benefits, it is not intended as a primary defense mechanism against XSS attacks. Rather, it would best serve as a defense-in-depth mitigation mechanism. A primary defense involves tailored security schemes that validate user inputs and encode user outputs. So far our work has involved CSP 1.0. In future work, we plan to investigate using directives with CSP 2.0, which as of February 2016 was still a W3C working draft.

ACKNOWLEDGMENTS

The work reported in this article is supported by the Networking and Distributed Computing (NDC) Lab and the Kulliyyah of Information and Communication Technology (KICT) at International Islamic University Malaysia (IIUM) under the internal lab-funded research project PCS1-I001-2014-4900.

REFERENCES

1. M. Johns, “Code Injection Vulnerabilities in Web Applications—Exemplified at Cross-Site Scripting,” PhD dissertation, Univ. of Passau, 2009; https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index/docId/144.
2. Open Web Application Security Project, “OWASP Top 10 – 2013: The Ten

Most Critical Web Application Security Risks," 2013; www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013.
3. I. Yusof and A.-S.K. Pathan, "Preventing Persistent Cross-Site Scripting (XSS) Attack by Applying Pattern Filtering Approach," *Proc. 5th IEEE Conf. Information and Communication Technology for the Muslim World* (ICT4M14), 2014, pp. 1–6.
4. L.K. Shar and H.B.K. Tan, "Defending against Cross-Site Scripting Attacks," *Computer*, vol. 45, no. 3, 2012, pp. 55–62.
5. E. Kirda et al., "Noxes: AClient-Side Solution for Mitigating Cross-Site Scripting Attacks," *Proc. 21st Ann. ACM Symp. Applied Computing* (SAC06), 2006, pp. 330–337.
6. T. Jim, N. Swamy, and M. Hicks, "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies," *Proc. 16th Int'l ACM Conf. World Wide Web* (WWW07), 2007, pp. 601–610.
7. Y. Nadji, P. Saxena, and D. Song, "Document Structure Integrity: A Robust Basis for Cross-Site Scripting Defense," *Proc. 6th Ann. Network & Distributed System Security Symp.*(NDSS09), 2009; www.cs.berkeley.edu/~dawnsong/papers/2009%20dsi-ndss09.pdf.
8. M.T. Louw and V.N. Venkatakrishnan, "Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers," *Proc. 30th IEEE Symp. Security and Privacy* (S&P09), 2009, pp. 331–346.

ABOUT THE AUTHORS

IMRAN YUSOF is an MS student in computer science at the Kulliyyah of Information and Communication Technology at International Islamic University Malaysia (IIUM) and a security consultant. His research interests include cybersecurity; Web-application vulnerabilities, such as SQL injection and XSS; and countermeasures to these attacks. Yusof received a BSc in computer science from IIUM. Contact him at suretarget@gmail.com.

AL-SAKIB KHAN PATHAN is a faculty member at large at the Islamic University of Madinah, and a visiting faculty member in the Computer Science and Engineering Department at the University of Asia Pacific, Bangladesh. His research interest includes wireless sensor networks, network security, and e-services technologies. Pathan received a PhD in computer engineering from Kyung Hee University. While conducting the research reported in this article, he was an assistant professor in the Computer Science Department at IIUM. He is a Senior Member of IEEE and editor of *IEEE Communications* and the *International Journal of Sensor Networks*. Contact him at spathan@ieee.org.