Hindawi
Security and Communication Networks
Volume 2019, Article ID 2156906, 13 pages
https://doi.org/10.1155/2019/2156906

Research Article

WebMTD: Defeating Cross-Site Scripting Attacks Using Moving Target Defense

Amirreza Niakanlahiji[1] **and Jafar Haadi Jafarian**[2]

[1]*UNC Charlotte, USA*
[2]*University of Colorado Denver, USA*

Correspondence should be addressed to Amirreza Niakanlahiji; aniakanl@uncc.edu

Received 30 June 2018; Revised 31 January 2019; Accepted 13 February 2019; Published 14 May 2019

Academic Editor: Mamoun Alazab

Existing mitigation techniques for cross-site scripting attacks have not been widely adopted, primarily due to imposing impractical overheads on developers, Web servers, or Web browsers. They either enforce restrictive coding practices on developers, fail to support legacy Web applications, demand browser code modification, or fail to provide browser backward compatibility. Moving target defense (MTD) is a novel proactive class of techniques that aim to defeat attacks by imposing uncertainty in attack reconnaissance and planning. This uncertainty is achieved by frequent and random mutation (randomization) of system configuration in a manner that is not traceable (predictable) by attackers. In this paper, we present WebMTD, a proactive moving target defense mechanism that thwarts various kinds of cross-site scripting (XSS) attacks on Web applications. Relying on built-in features of modern Web browsers, WebMTD randomizes values of certain attributes of Web elements to differentiate the application code from the injected code and disallow its execution; this is done without requiring Web developer involvement or browser code modification. Through rigorous evaluation, we show that WebMTD has very a low performance overhead. Also, we argue that our technique outperforms all competing approaches due to its broad effectiveness, transparency, backward compatibility, and low overhead.

1. Introduction

Despite numerous proposed works on detection and prevention of XSS attacks [1–6], they are still among the most common threats on Web applications; examples are the recently discovered XSS vulnerabilities on Amazon [7], EBay [8], Twitter [9], Drupal CMS [10], and WordPress Tooltipy plugin [11].

According to a recent report by Akamai on the state of the Internet in 2018 [12], XSS is still among the top three most prevalent classes of attacks on Web applications. According to this report, in a period of 6 months (Nov. 2017 to April 2018), over 6 million XSS attacks have been discovered, which accounts for over 8% of attacks on Web applications. The occurrence of over 1 million XSS attacks per month emphasizes the gravity of mitigating these attacks as well as limited effectiveness of existing XSS mitigation methodologies.

Methodologies for countering XSS attacks could be broadly divided into two categories: (I) input validation or sanitization techniques that prevent injection of malicious code (i.e., JavaScript) but are highly susceptible to evasion [13]; (II) code differentiation techniques that prevent execution of injected code, including BEEP [3], ISR [14], CSP 1.0 [4] and 2.0 [15], Noncespaces [5], and xJS [6]. However, as demonstrated through our extensive evaluation, existing techniques suffer from several shortcomings, which limits their widespread adoption by real-world applications. Most importantly, almost all existing approaches either require developer involvement in the defense process [3, 4], entail modification of Web browser [6], suffer from high overhead (execution time, more traffic overhead) [3], or are susceptible to evasion [4].

In XSS attacks, an attacker designs and implements the exploit code on her side and then either feeds it to the Web application or sends a crafted URL directly to the users of

the Web application. In other words, the attacker's code will be executed on another machine and at a different time. These types of attacks inherently suffer from time of check to time of use (TOCTOU) design flaw [16]. TOCTOU is a class of software bug caused by changes in a system between the checking of a condition (such as a security credential) and the use of the results of that check. While attackers have traditionally benefited from TOCTOU design flaws [17], defenders can benefit from this flaw by altering certain system parameters in the gap between the implementation of the attacker's exploit and its execution on the target, to defeat several classes of code injection attacks on Web applications.

A novel class of proactive defense techniques, known as moving target defense (MTD) [18], takes advantage of this gap between time of check (exploit development) and time of use (exploit execution) by randomizing (or mutating [18]) the environment after time of check and before time of use, in a manner that invalidates prerequisites of the exploits. For example, by randomizing IP addresses of networks, RHM ensures that exploits built based on scanning at a previous time [19] (time of check) will not be executable at a later time (time of use). Or, by randomizing the base addresses of memory segments of a process, ASLR [20] ensures that exploits that are developed on attacker's side (time of check) are not executable on the target (time of use).

In this paper, we present an MTD approach, called WebMTD, that defeats various types of XSS attack on Web applications, including persistent, nonpersistent, and DOM-based XSS, in a manner that is transparent to the Web application, Web browsers, and developers.

WebMTD exploits TOCTOU [16] to differentiate injected JavaScript code from the application code. To this aim, WebMTD changes the Web application from the time of developing an injection code (TOC) to the time of its execution on a target machine (TOU) in order to identify this injected code and disallow its execution.

WebMTD makes a Web application XSS-resistant by automatically adding new attributes to certain HTML elements and randomizing their values over time. In addition, it automatically inserts necessary security check functions into the Web application and instructs Web browsers to invoke them before interpreting client-side code blocks or event handlers. These security check functions validate the authenticity of the code blocks and handlers.

In summary, WebMTD has the following properties: (1) It prevents the execution of various types of XSS attacks. (2) It has low overhead. Through rigorous evaluation, we show that our approach has very negligible execution overhead on web browsers, and also the transmission delay is very negligible. The space overhead is also in order of several hundred bytes. (3) It is transparent to Web applications and developers; WebMTD could be implanted in any legacy or new Web application in an automated manner and without demanding any restrictive coding practices. (4) It is transparent to Web browsers; WebMTD works with any modern Web browser that fully complies with HTML5 specification. Moreover, users with unsupported browsers can still use the WebMTD-enabled Web application but without the code injection resistance capability.

In the rest of paper, first, we present a background on XSS attacks in Section 2. Then, we present a summary of related works in this area in Section 3. Next, in Section 4, we discuss how WebMTD counters such attacks. In Section 5, we provide a quantitative evaluation of WebMTD in addition to comparing it with competing solutions, including xJS, ISR, BEEP, Noncespaces, and CSP 1.0 and 2.0. In Section 6, we conclude the paper.

2. Background

In this section, we briefly provide necessary background knowledge about XSS attacks. Generally, in XSS attacks, the attacker designs and implements the exploit code on her side and then either feeds it to the Web application or sends a crafted URL directly to the users of the Web application. To launch this attack, an attacker injects a script block instead of entering a legitimate input. For example, instead of entering an email address in email field, an attacker enters `<script>alert("test")</script>`. This value could be either stored in the database (in stored XSS), reflected in the response document (in reflected XSS), or directly consumed by the Web browser (in DOM-based XSS) [21]. If this input value is received by the Web browser at a later time, e.g., by retrieving it from the database, the Web browser would not be able to differentiate this script block from original application script blocks and would interpret it, thus showing the alert message. This injected JavaScript code could be used to steal cookies, to deface the document, or to submit unauthorized forms. In essence, XSS is mainly used to evade the *same-origin* policy enforced by modern Web browsers and hence compromises the integrity of the vulnerable Web application.

In addition, XSS attacks can also be used to launch other attack vectors such as CSRF (Cross-Site Request Forgery), in order to evade existing widely used mitigation techniques such as CSRF cookies. In CSRF, the attacker forces an authenticated user to execute unwanted actions such as transferring money and sending message to other users on the Web application. For example, an attacker might inject a JavaScript code by exploiting an XSS vulnerability in the Web application. When a user visits the page that contains the injected code, the injected code will be executed. This code can invoke arbitrary Web API on behalf of the user.

3. Related Work

Methodologies for countering XSS attacks could be broadly divided into two categories: (I) input validation or sanitization techniques and (II) code differentiation techniques. Input validation or sanitization techniques use a filtering module that looks for scripting commands or metadata characters in untrusted inputs, and filter or sanitize any such input before these inputs are processed by the Web application [1, 22–28]. However, these techniques suffer from several major shortcomings, including potentials of false negatives (malicious inputs going undetected), overhead on developers, and potentials of false positives (legitimate input is rejected) [1, 29].

In contrast, code differentiation techniques focus on differentiating application code from the injected ones and disallowing execution of the latter. These approaches do not suffer from the shortcoming of input validation techniques [29]. Notable examples of these approaches are BEEP [3], ISR [14, 30], CSP 1.0 [4] and 2.0 [15], Noncespaces [5], and xJS [6].

BEEP [3] prevents XSS attacks by only allowing whitelisted JavaScript blocks to be executed on a client's browser. Before deploying a Web application, BEEP computes the hash values of all JavaScript blocks in a Web page and makes a whitelist from them. This whitelist and some checking code will be embedded in the Web page. A BEEP-aware browser runs the checking code before executing any script block on the page. If computed hash value of a script block cannot be found in the whitelist, then the checking code prevents its execution.

Kc et al. introduced Instruction Set Randomization (ISR) [14] to prevent code injection attacks in machine code. The basic idea of ISR is to generate a process-specific instruction set for a vulnerable system by encrypting instruction. An attacker who does not know the instruction set cannot inject and execute any code on the vulnerable system. In [30], Boyd and Kc et al. expanded their original ISR work by demonstrating the generality of ISR; they presented how ISR can be implemented to prevent code injection attacks against Perl scripts and SQL queries. Although they did not present how this can be implemented for JavaScript engines embedded in browsers which are targets of XSS attacks, we believe that the same idea can be implemented in browsers to prevent XSS attacks; hence we consider ISR as a related work.

Athanasopoulos et al. [6] introduce xJS framework to mitigate XSS attacks. The basic idea is to transpose JavaScript code blocks to another domain on the Web server at runtime and to reverse the transposition on the client browser. In their implementation, they used XOR operation to transpose JavaScript blocks in static HTML documents. Upon requesting an HTML document, xJS XORs each script block on the Web page with a secret key. Then, it includes the key in an HTTP response header. On the client's browser, the encrypted code block will be again XORed with the transmitted secret key to retrieve the original code blocks.

Stamm et al. proposed Content-Security-Policy (CSP)[4], which is now implemented in all major Web browsers such as Firefox, Chrome, and Safari. In CSP, a developer or maintainer of a Web application determines the policies for each Web page. Each CSP policy specifies which type of resources in a page must be loaded by the browser. By default, CSP-enabled browser prevents inline script blocks in a Web page. Only external script blocks (i.e., `<script>` elements that have `src` attribute) may be executed. They will be executed if their `src` attributes point to trusted locations.

In CSP 2.0, use of inline scripts is discouraged, but allowed. The developer must explicitly whitelist scripts using a randomly generated nonce. Similar to our approach, only whitelisted scripts with valid nonce are executed. Alternatively, an inline script could be whitelisted by specifying its hash. In both cases, contrary to our approach, developer's involvement is required in the policy definition process, and the policy enforcement process is not transparent to Web server, Web application, and Web browser. In both CSP 1.0 and 2.0, JavaScript codes inside event handlers of HTML elements are not allowed. This limits the applicability of the model, as event handler attributes (e.g., `onclick`, `onchange`) are widely used in Web applications.

Gundy et al. introduced Noncespaces technique [5] to prevent XSS attacks. In Noncespaces, a random XML namespace is generated for each requested XHTML document and all trusted XHTML element tags in that document will be modified to start with the generated namespace. Only XHTML elements with proper namespace will be rendered or executed. To enforce this rule, either a Web browser must be modified or a Web proxy must be placed in front of a client's Web browsers. Noncespaces approach can effectively prevent XSS attack while its runtime overhead on clients' Web browsers is much lower than BEEP, since only tag names on a Web page must be checked. In addition, changing the Web application does not have any cost, since no offline processing is needed upon changing the Web application.

In Section 5.6, we discuss weaknesses of these approaches in detail; we provide an in-depth comparison of these approaches with WebMTD to show that none of the existing works provide an effective solution to XSS that is transparent to Web server, Web application, and Web browser and does not rely on Web developers or system administrators.

4. WebMTD

The root cause of XSS attacks is that the internal JavaScript engines in browsers have no reliable means to differentiate between the Web application code and the code injected by an attacker. In other words, the JavaScript engine executes *any* code that is passed by the Web browser. The Web browser, on the other hand, trusts any code that is coming from the Web server.

The main goal of WebMTD is to address this problem efficiently by enabling a Web browser to reliably verify the authenticity of a client-side code block (e.g., JavaScript code block), before letting the code block be executed by the internal interpreters in the client-side environment.

To achieve this goal, WebMTD, first, must be able to reliably identify the application code. Second, it must be able to mark these codes so that (I) attackers cannot mimic it, and (II) it does not change the logic of the Web application. Finally, WebMTD must implant necessary security check functions in the Web application to enforce verification of a code's mark before its execution.

The design of WebMTD is based on two basic observations. First, the interval after development phase and before the end of deployment phase of a Web application is a safe time frame to identify all legitimate code blocks of a Web application. No new code block will be added after development phase and attackers have not yet had the chance to inject their codes. Second, client-side code injection attacks have two separate steps. In the first step, an attacker injects her code or craft a URL containing malicious code; in the second step, her victims run the code on their system. If the markings depend on time and are not guessable or retrievable, then the attacker cannot replicate them and hence he will be defeated.

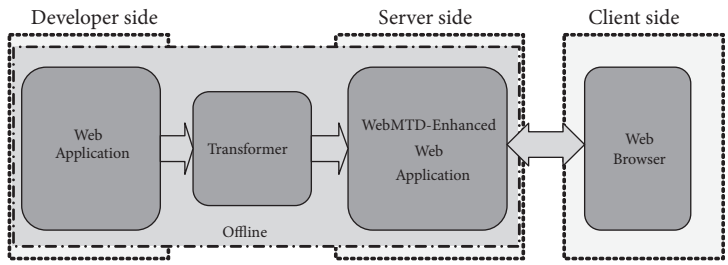

Figure 1: WebMTD architecture. This figure is reproduced from Niakanlahiji et al. (2017) (under the Creative Commons Attribution License/public domain).

```
<img onload="alert('XSS')" src="XSS.png">
```

Code 1: Original injected HTML element.

Figure 1 shows the overall architecture of WebMTD. The WebMTD transformer is responsible for marking all client-side codes in the input Web application. Moreover, it should add the necessary code to make these markings unique over time. Finally, it should add security check functions that must be executed by browsers to verify the markings. This transformation process is performed before deploying the Web application on a production system.

The current implementation of WebMTD transforms Web applications written in PHP (as a server-side programming language) and JavaScript (as a client-side programming language) and works seamlessly with all modern browsers such as `Firefox` and `Chrome`; it does not need to modify the browser code or to install an add-on to extend their functionality.

Traditionally, XSS attack focused on injection of external or internal script blocks; however, nowadays, other types of XSS attacks are also prevalent. Instead of injecting an inline or external script block, the attacker may inject an HTML element into the DOM structure of a Web page, such as an `img` or `a` tag with an event attribute that performs a malicious operation. By reviewing a list of all potential HTML-element-based attacks, we divide them into two broad categories based on the attribute name which includes the injected code:

(i) Using event attributes of an injected HTML element. For example, see Code 1.

This includes injection of any HTML element with any potential event attribute. Note that if the event handler of an element is assigned using `addEventListener` Web API in a JavaScript code block, it will be validated as part of the (inline/external) script block to which it belongs. Therefore, our focus here is only on HTML elements with explicit event attributes, such as the above example.

(ii) Using `Javascript: URL` as the value of a source attribute. For example, `src` attribute of `img` or `iframe` tags, or `href` attribute of `link` tag (see Code 2).

```
<iframe src="javascript:alert('XSS')"></
    iframe>
```

Code 2: Original injected HTML element.

```
<IMG src="javascript:alert('XSS')" >
```

Code 3: Original injected HTML element.

In older browsers, this attack vector could exploit the `src` or other similar attributes of many different types of HTML elements such as `img` or `bgsound`. However, based on our thorough investigation, in modern browsers this attack is only viable for `iframe` element. Therefore, in our methodology we only consider this element. For example, XSS techniques such as the example shown in Code 3 are *not* viable in modern browsers anymore.

To prevent the execution of injected JavaScript code, WebMTD relies on `MutationObserver` interface, which is part of DOM 3 standard and it is implemented by latest versions of almost all modern web browsers such as `Chrome`, `Firefox`, and `Safari`. WebMTD also handles `beforescriptexecute` event, introduced in HTML5 [31] specification, to address some issues observed in `Firefox` browser.

During transformation phase, WebMTD marks all HTML elements in the Web application that are either script blocks or have event handlers or use `JavaScript: URL` as their source. It also adds two security checks, written in JavaScript, to the web application. These checks are invoked by the Web browsers to validate the added markings. Any JavaScript code that is not part of an element with a valid marking will be discarded before being executed. In

```
<script language="javascript">
```

Code 4: Original script element.

```
<script
runtimeId="<?php echo $id4trustedBlocks;? >"
language="javascript">
```

Code 5: Elements with runtimeId attribute.

this manner, we enable browsers to distinguish between legitimate and injected JavaScript codes.

To mark elements that are either script block or have event attributes or `JavaScript: URL` as the source, the transformer scans all the code files in the Web application and rewrites start tags of such elements. Upon detecting the start tag of such element, `WebMTD transformer` inserts a new attribute, which is called `runtimeId`, into the tag. The exact value of this attribute is determined at runtime by the Web application when a client requests the page that contains the element.

Suppose that the `script` tag shown in Code 4 is found in a PHP file.

After rewriting, Code 4 is changed to Code 5.

Or, Suppose that the elements shown in Code 6 are found in a PHP file.

After rewriting, Code 6 is changed to Code 7.

In addition to inserting this attribute into each element of interest, `WebMTD transformer` embeds the PHP code block shown in Code 8 at the beginning of each PHP code file that represents a Web page.

Upon execution of this block, a random number is generated (on the server-side) which will be assigned to all `runtimeId` attributes on that page.

Moreover, `WebMTD transformer` locates the place of the `head` element and inserts the JavaScript code block shown in Code 9 as its first child. In this manner, WebMTD ensures that this script block is the first code block interpreted by client browsers.

Upon execution of Code 9, a `MutationObserver` object is instantiated by passing a callback function to the `MutationObserver` constructor. This callback function is called when DOM changes matching the options passed to `observe` method occur. In WebMTD, any change to the DOM tree of the page triggers the provided callback. As mentioned before, this script block is placed in the beginning of `<head>` element; thus, the callback is called for body element and for each of its descendant elements. When attackers inject a script block or an HTML element with an on-event attribute, they are basically changing the DOM tree of the page; thus WebMTD callback function is also called for such injected elements.

The callback only checks elements that are added to the DOM tree; elements removed from the tree are not examined. If the newly added element (A) is a script block, (B) has an event attribute, or (C) is an `iframe`, then the callback checks whether it has a valid `runtimeId` attribute. If the attribute is missing or the value is not valid, the callback prevents the execution of embedded JavaScript code by changing the type of the script block or by setting the value of event or source attribute to `null`.

In all modern browsers, except `Firefox`, changing the `type` attribute of a script element to anything other than `text/JavaScript` prevents the execution of JavaScript code. To handle Firefox, WebMTD relies on another event, `beforescriptexecute`, introduced in HTML 5 specification. WebMTD registers an event handler for `beforescriptexecute` event, which is raised before execution of any inline or external script block. Upon invocation, the event handler examines the `runtimeId` attribute of the corresponding script block and if the attribute is missing or the value is not valid, it disallows the execution of the script block by returning `e.preventDefault()` object.

To illustrate how this can prevent an attacker, suppose a Web application has only one page, and the page is XSS vulnerable. An attacker requests the page. Upon investigating the page, the attacker realizes that the `runtimeId` value is `59b6d298f36ae` (see Code 10).

Then the attacker crafts the malicious code shown in Code 11 and injects it into the database by exploiting the XSS vulnerability on the page.

After that, a normal user visits the vulnerable web page. The HTML code of the Web page will be as shown in Code 12.

In this HTML document, the `runtimeId` of the injected script block (`59b6d298f36ae`) is different from the `runtimeId` that is specified in the security check (`59b6d485c9f5b`); hence the malicious block will not be executed, while the legitimate script block will be executed. For other types of injection attack, the same analogy holds and the attacker cannot reuse the learned information to evade the system.

Since the generated `runtimeId` changes for every client request, the injected element will not have a `runtimeId` valid value. Theoretically, the attacker may still make a lucky guess for the `runtimeId` value, but its chance is equal to the success probability of the *Birthday attack*, which is substantially low (for unique IDs with 13 hex characters, this probability is $1/2^{26}$).

5. Evaluation

In this section, we present our evaluation of WebMTD. We first introduce and define our criteria and metrics for evaluation. Then, using these fourfold criteria, we present a thorough evaluation of WebMTD. Finally, we compare WebMTD with competing anti-XSS approaches, including BEEP, CSP 1.0 and 2.0, xJS, ISR, and Noncespaces.

5.1. Criteria. In this section, we introduce our criteria and metrics for evaluating our approach and comparing it to other alternative approaches.

5.1.1. Effectiveness. Effectiveness is measured in terms of the classes of XSS attacks that are thwarted by a solution, as well

```
<a id="link1 " onclick ="alert( 'benign ')" >link</
    a>
...
<iframe src ="http://..." ></ iframe>
```

Code 6: Original elements.

```
<a id="link1" onclick ="alert( 'benign ')"
    runtimeId ="<? php echo $id4trustedBlocks;?>
    ">link</a>
...
<iframe src ="http://..." runtimeId ="<? php echo
     $id4trustedBlocks;? >"></ iframe>
```

Code 7: Original elements.

```
<? php
 $id4trustedBlocks=uniqid();
 ?>
```

Code 8: Setting `id4trustedBlocks` value.

as its resistance to evasion techniques that are potentially used by attackers.

5.1.2. Overhead. Placing a new security solution in front of a Web application can be costly due to imposing different types of overhead during and after its deployment. We consider four types of overheads.

(i) *Deployment overhead*: It reflects the time and resources that a developer or a system administrator must invest to deploy a solution.

(ii) *Round-trip latency*: It measures the round-trip delay that is experienced by end users as a result of deploying a solution.

(iii) *Execution overhead on clients*: It measures the execution time overhead that a solution imposes on the Web browser.

(iv) *Space overhead*: It measures the space overhead that is imposed by a solution, as a result of expanding the code base.

5.1.3. Transparency. Transparency metrics show whether applications on the server and client sides must be changed to enable an anti-XSS solution. This is evaluated with respect to the following:

(i) *Browser code modification*. It determines whether a client's Web browser code must be modified. Demanding browser code modification is very prohibitive, because not only does it need vendors' cooperation for deployment, it also needs clients' cooperation for updating the browsers.

(ii) *Developer involvement*. It indicates whether Web developers must adhere to new coding practices while implementing a Web application.

5.1.4. Backward Compatibility. Backward compatibility metrics reflect the degree of flexibility that a solution shows to applications and technologies that do not comply with its requirements.

(i) Web browser. This metric indicates whether deploying a solution prevents users with unsupported Web browsers from accessing the Web application.

(ii) Web application. This metric indicates whether a solution can be applied to existing Web applications without affecting their functionalities.

5.2. Security Effectiveness. WebMTD prevents cross-side scripting attacks by obstructing the execution of any injected JavaScript code block on users' browsers, either as an inline or as an external script block, or by injecting an HTML element with event attribute or `JavaScript: URL` into DOM. We evaluated WebMTD effectiveness by applying it to a few well-known open-source Web applications that have known available XSS attacks.

We collected 11 confirmed XSS attacks against popular open-source Web applications, including 2 exploits on `osTicket` [32], 2 exploits on `osCommerce` [33], 3 exploits on `wordpress` [34] and its plugins, and 4 exploits on `Joomla` and its components [35] from `exploit-db.com` exploit database. In addition, we introduced 3 XSS vulnerabilities on `osCommerce`. On each XSS vulnerability, we test a variety of XSS codes with different types and evasion techniques. Table 2 provides examples of the injected XSS codes which vary in terms of type and evasion techniques. Avid readers are referred to [36] for a complete list of known XSS exploit codes. We examined each of these XSS exploits against each XSS vulnerability. WebMTD successfully blocked all of these exploits.

```
<script type="application/ javascript">
var runtimeId = "<? php echo $id4trustedBlocks
    ;?>";
var events = {'onabort': true , 'onafterprint':
   true , /*the rest is discarded*/ }

Function validateNode (node) {
 if ( node.nodeType == 1) {
 if ( node.tagName == "SCRIPT" && (!node.
    hasAttribute (" runtimeId ") || node.
    getAttribute (" runtimeId ") != runtimeId ))
 { node.type = 'nojscode'; }
 else {
 for( var j = 0; j < node.attributes.
 length; j++) {
 if (events[ node.attributes [j].name] &&
 (! node.hasAttribute (" runtimeId ") ||
 node.getAttribute (" runtimeId ") !=
 runtimeId )) {
  node.setAttribute ( node.attributes [j
  ].name, null );
 }
 }
 if ( node.getAttribute (" src ") != null &&
 (! node.hasAttribute (" runtimeId ") ||
 node.getAttribute (" runtimeId ") !=
 runtimeId )) {
  node.setAttribute (" src ", null );
 }
 }
 }
}
var dom_observer = new MutationObserver (
function (mutations) {
 mutations.forEach (({ addedNodes }) => {
 addedNodes.forEach (node => {
 validateNode (node); })
 })
});
var container = document.documentElement ||
    document.body ;
var config = { attributes: true , childList:
    true , characterData: true,subtree: true };
dom_observer.observe (container, config );

window.addEventListener ('beforescriptexecute',
function (e) {
 return ( e.target.hasAttribute (" runtimeId ")
 && e.target.getAttribute (" runtimeId ")==
 runtimeId ) ? e : e.preventDefault ();
}, true );
</script>
```

Code 9: Security check function using MutationObserver.

5.3. Overhead. To evaluate the performance overhead of WebMTD, we measure it in terms of the aforementioned four overhead metrics.

Our test environment consisted of two virtual machines each with 2 GB of RAM and one virtual CPU with 10 GB of hard disk. The first VM acts as our Web server, on which we installed `CentOS` 7, `Apache httpd server v2.4`, and `MySQL` server. In addition, we hosted two versions of a popular open-source online store, `osCommerce` [33] on this server. The first one is the original (unmodified)

```
<script type="text/javascript">
var runtimeId = "59b6d298f36ae";
[codes for MutationObserver and
    beforescriptexecute]
...
</script>
...
<script runtimeId="59b6d298f36ae" type="text/
     javascript">
...
</script>
```

Code 10: Attacker's instance.

```
<script runtimeId="59b6d298f36ae" type="text/
     javascript" >
[With malicious code] </script>
```

Code 11: Crafted script block.

```
<script type="text/javascript">
var runtimeId = "59b6d485c9f5b";
[codes for MutationObserver and
    beforescriptexecute]
...
</script>
...
<script runtimeId="59b6d485c9f5b" type="text/
     javascript">
[Some legitimate code]
</script>
...
<script runtimeId="59b6d298f36ae" type="text/
     javascript" >
[Some malicious code]
</script>
```

Code 12: User's instance.

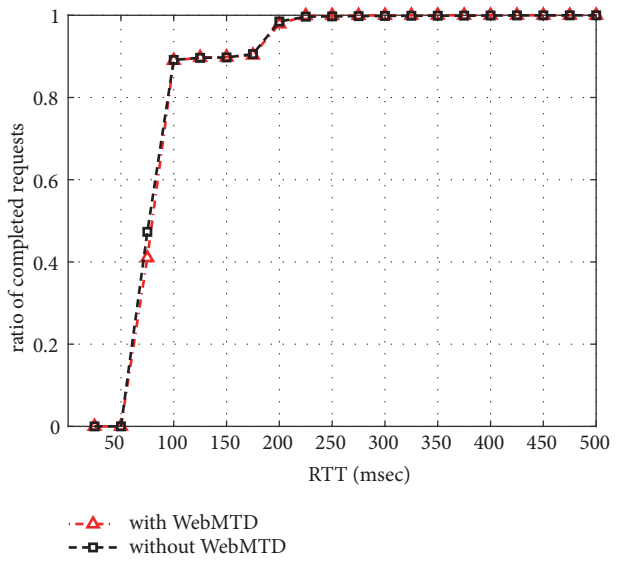

Figure 2: Cumulative distribution function of RTTs for 10,000 serial HTTP requests.

`osCommerce v2.3.3.4`, and the other one is WebMTD-enhanced version of the same software. The second VM acts as our client environment and is used for measuring performance metrics.

We used `Apache JMeter v3.1`, built-in `performance tool` in `Firefox v60`, and `ReloadMatic` add-on for this purpose. To ensure that our measurements are realistic, we hosted these two machines in two separate networks across the Internet.

Deployment Overhead. The WebMTD validation codes and attributes are applied to a given Web application in the postdevelopment stage and in an automated manner. It imposes no coding restrictions on developers and requires no changes to the Web server. It also requires no changes to Web browsers. Therefore, WebMTD has a very low deployment overhead for clients, administrators, developers, and vendors.

Round-Trip Latency. This metric basically measures the latency experienced by clients in receiving replies to their HTTP requests. Figure 2 shows the cumulative distribution function (CDF) of round-trip latencies for 10,000 serial HTTP requests to both the original `osCommerce` and its WebMTD-enhanced version. The requests were all sent to `index.php` which has roughly the same number of `JavaScript` blocks as any other page of `osCommerce`. The round-trip latency is calculated using the `Apache JMeter` on the measurement machine. In our context, round-trip latency denotes the time interval between initiating an HTTP request and receiving the response. This includes transmission time, propagation delay, and processing time. The propagation delay time is the same for both versions because it is a function of medium and end-to-end distance; the transmission time depends on the Web traffic size which basically reflects the size of HTTP request and its HTTP response. While the HTTP requests sent to both versions are essentially the same, the sizes of HTTP responses might be slightly different due to WebMTD added code to dynamic HTML documents. However, as discussed in Section 5.3, the size of added code for each HTML document is 1,784 bytes on average. On a 16 Mbps link (average Bandwidth in USA [37]) this results in ≃ 0.5ms delay.

The remaining difference between round-trip time latencies shows the processing time overhead of WebMTD-enhanced version as compared to the original one. As depicted in Figure 2, for completing 20% of requests, this difference is 1ms, for 50% is 1ms, and for 80% is 2 ms. This shows that the extra processing time due to WebMTD added code is significantly low, thus making our approach practical for real-world applications.

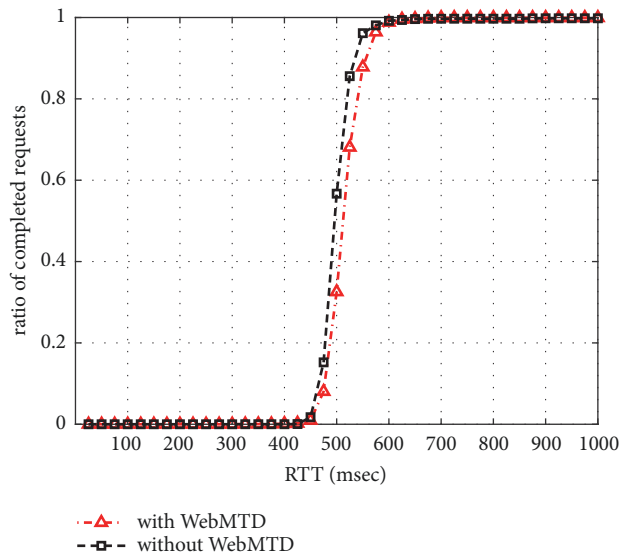

Figure 3: Cumulative distribution function of RTTs for 1,000 10-concurrent HTTP requests.

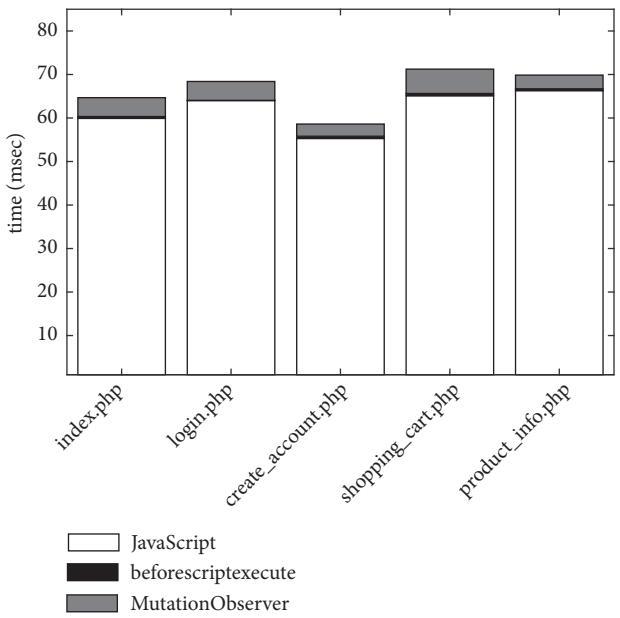

Figure 4: Avg. execution time for original JavaScript code vs. WebMTD code.

The same argument holds for concurrent requests, as depicted by Figure 3. In this experiment, we sent the same number of requests (10,000), but in 1,000 batches of 10 concurrent requests. As depicted in Figure 3, for completing 20% of requests, this difference is 8ms, for 50% is 15ms, and for 80% is 23 ms. Note that the round-trip delay, although slightly higher than the serial requests, is still negligible.

Script Execution Time. We also calculated the time that is needed for executing the `beforescriptexecute` event handler embedded in the document by WebMTD and also the `MutationObserver` API. `beforescriptexecute` event handler is called before execution of each script block on each document.

To measure the execution times of these handlers, we used built-in performance monitor tool in Firefox and `ReloadMatic` add-on [38]. We randomly selected 5 pages in `osCommerce` and reloaded each page 100 times by the add-on while recording execution times of the pages with `performance tool`. For each page load, we extracted execution times for (I) JavaScript blocks, (II) `beforescriptexecute` event handler, and (III) `MutationObserver` API. The first item denotes the execution time of the `JavaScript` code blocks, which is the same for both the original and modified versions, while the sum of second and third items shows the extra time required for execution of WebMTD added codes. The first item is a function of a number of JavaScript statements on the execution path, the second item depends on the number of *executed* code blocks, and the third item depends on the number of HTML elements with event handlers and `iframe` elements.

Figure 4 shows the average time for each of these two categories on these pages. Note that compared to the average execution time for JavaScript blocks, the extra time needed for executing the WebMTD added codes is negligible.

On average handling the `beforescriptexecute` event by WebMTD requires 0.05 ms. Given that the average number of script blocks on modern applications is in order of tens (as shown in Table 1, 26 code blocks on average for top 100 Websites in Alexa ranking list), the extra execution time for the event handler is in order of a few milliseconds. Also, on average executing the `MutationObserver` API function requires 4 ms.

HTML Document Size. The added event handler for `beforescriptexecute` inserts 174 bytes of code in *minified* format to each dynamic HTML document. The `MutationObserver` API code in *minified* format inserts 911 bytes to each dynamic HTML document. Each document on average includes 24 script blocks, 3 event attributes, and 1 `iframe`. Therefore, an extra 700 bytes is required for adding `runtimeId` and value to each document (see (Table 1)). The total added JavaScript code for each dynamic HTML document is 1,785 bytes. Therefore, WebMTD on average increases the size of a typical dynamic HTML document only by 1%.

5.4. Transparency

Browser Modification. Users do not need to update their browsers or install a new add-on to make their browsers WebMTD-enabled. WebMTD relies on built-in features of modern browsers to enforce its validation logic before execution of any JavaScript code on the browsers. In other approaches, the users need to configure their environments by adding a new add-on, updating their browsers, or using a web proxy. All investigated approaches in the literature, except WebMTD, require browser code modification.

Table 1: Inline/external JavaScript block usage.

Website	No. of elm. with handlers	No. of iframes	No. of inline script blocks	No. of external script blocks	Ratio of inline blocks
google.com	3	0	10	2.9	0.78
youtube.com	0	3	11.3	6.4	0.64
facebook.com	6	2	13.8	11.75	0.54
baidu.com	1	0	12.3	5.1	0.71
yahoo.com	5	0	11.2	19.7	0.36
Top 100 sites (avg)	3	1	14.9	10.1	0.60
osCommerce	2	0	8.8	4.6	0.65

Table 2: Types of XSS exploit codes and examples.

Type	Sub-type	Example codes	Defense
Script block	External	<script src=http://.../XSS.js></script>	Viable on all browsers Blocked by WebMTD
	Inline	<script>alert('XSS')</script>	Viable on all browsers Blocked by WebMTD
Inline script code	Elements w/ `javascript: URL` as source	<img src="javascript:alert('XSS');" > <img src=javascript:alert(String.fromCharCode(88,83,83))> <bgsound src="javascript:alert('XSS');" > <link rel="stylesheet" href="javascript:alert('XSS');" > ... <IFRAME SRC="javascript:alert('XSS');" ></IFRAME>	Now, only viable for iframe. Blocked by WebMTD
	Elements w/ event attributes	<img src=# onmouseover="alert('XSS')" > <img src= onmouseover="alert('XSS')" > <img onmouseover="alert('XSS')" > <img src=/ onerror="alert(String.fromCharCode(88,83,83))" ></img> <body onload="alert('XSS')" > <img src=/ onrror="alert('XSSd')" >	Viable on all browsers Blocked by WebMTD

Developer Involvement. Web developers do not need to follow a restrictive coding practice to make their application WebMTD-enabled. In contrast, to make web applications ready for other approaches, developers must follow very restrictive rules, which can make the web development process more costly. For example, to use BEEP, developers must not generate JavaScript code blocks in their server-side code. To use CSP 1.0, users must not use inline JavaScript code blocks or on-event handler attributes. In CSP 2.0, users are not still allowed to have event attributes and must explicitly whitelist legitimate inline script code blocks. To use Noncespaces, developers must avoid using XML namespaces for XHTML element names. These requirements mean these approaches cannot be applied to the legacy Web application without requiring code refactoring by developers.

5.5. Backward Compatibility. WebMTD relies on a few internal features of modern web browsers such as `beforescriptexecute` and `MutationObserver` to prevent the execution of XSS attacks. However, some users may browse a WebMTD-enabled Website with a browser that lacks those features. In that case, our event handler `beforescriptexecute` and `MutationObserver` are never called. These functions are responsible for validating the token IDs for script blocks; thus our approach is deactivated in such cases. Still these users can interact with the web application as the scripts on the web application can be directly executed by the JavaScript engines. We only added a new attribute to script blocks and HTML elements without changing the script codes.

Legacy web applications can be transformed with WebMTD rewriter without developers' involvement as we do not impose any special coding practices.

5.6. Comparison with Other Techniques. In this section, we compare the state-of-the-art solutions against Web code injection attacks including BEEP, CSP 1.0, CSP 2.0, ISR, xJS, and Noncespaces with WebMTD.

BEEP overhead on a Web browser is not negligible. A client browser must compute the hashes of all script blocks in a Web page. This will cause a delay in loading of the page which can be significant on less powerful devices like smartphones or tablets. In addition, as mentioned by other authors [29], BEEP requires the hash value of a script block to be computed statically. As a result, BEEP does not allow execution of legitimate script blocks which are dynamically generated by the server-side program.

Table 3: Comparison of WebMTD vs. existing techniques against client-side code injections.

Criteria		BEEP	xJS	Noncespaces	ISR	CSP 1.0	CSP 2.0	WebMTD
Effectiveness	External blocks	yes	yes	yes	-	yes	yes	yes
	Inline blocks	yes	yes	yes	-	prohibited	yes	yes
	Elements with event attributes	yes	yes	yes	-	prohibited	prohibited	yes
	Elements with JavaScript: URL	no	yes	yes	-	prohibited	prohibited	yes
Overhead	Deployment overhead	low	low	low	low	high	high	low
	Round-trip latency	low	medium	medium	high	low	low	low
	Execution overhead on browser	high	medium	medium	high	low	low	low
	Space overhead	low	low	low	low	low	low	low
Transparency	Browser modification?	yes	yes	yes	yes	yes	yes	no
	Developer involved?	no	no	yes	no	yes	yes	no
Backward Compatibility	Web browser	yes	no	yes	no	yes	yes	yes
	Web application	no	yes	no	yes	no	no	yes

CSP 1.0 is not widely used [39] in modern Web applications because it needs the involvement of the developers of the system. They need to define the policies. In addition, it dictates several restrictions on the code; for example, no inline JavaScript block can be used in the Web application. These restrictions can significantly affect the performance of large Web applications [29] and make it harder to implement CSP for legacy Web applications. We have examined ten pages from each of Alexa top 100 Websites to see how prevalent the use of inline script block is among popular Web sites. As it is shown in Table 1, 0.60 percent of scripts in these Websites are inline and hence solutions such as CSP 1.0 that require abandoning of inline scripts cannot be used in practice.

CSP 2.0 is the current successor of CSP. The main change in CSP 2.0 is allowing developers to use inline scripts by adding nonce or hash sources to `script-src` policy [40]. To do so, developers must either (1) manually add a nonce attribute to all inline scripts and specify the nonce value in Content-Security-Policy at runtime or (2) compute the hashes of all inline scripts and add them to Content-Security-Policy. Although the nonce approach is similar to our approach, still CSP 2.0 does not allow usage of JavaScript code as event attributes or `JavaScript: URL`. The only way to do so in CSP 2.0 is to specify `'unsafe-inline'` as a `script-src` policy, which essentially allows execution of inline scripts, event handlers, or `JavaScript: URL`s, without validating them. Therefore, CSP 2.0 either disallows usage of event handlers or `JavaScript: URL`s, which makes it prohibitive and violates backward compatibility (see Table 1), or allows them in an insecure manner, thus making the Web application susceptible to XSS attacks.

ISR could be used to defeat XSS attacks, by randomizing the JavaScript instruction set. However, this approach is very expensive as it requires encryption of JavaScript statements on the server-side and decryption on the browser side. These encryption/decryption operations are costly. In addition, a Web server needs to have a JavaScript parser on its side and this parser must parse all the JavaScript code blocks in the requested page. Therefore, implementing ISR can be costly with regard to execution time. ISR also needs a key management protocol and both parties must be aware of the randomization.

xJS is an adaptation of ISR for Web applications in which JavaScript blocks are transformed by XORing with a key. However, its prototype only transforms HTML documents which are only vulnerable to a few types of XSS attacks. Modern Web applications use server-side programming languages such as PHP and JSP to dynamically generate HTML documents on the fly. If we use xJS to transform dynamically generated HTML documents, it will not be able to prevent persistent or reflected XSS attacks, because xJS cannot differentiate between a legitimate script block and an injected script block in a generated HTML document and hence encrypts both of them with the secret key, and those blocks will again be encrypted on the client's browser.

Noncespaces can interrupt normal operation of a Web application. One of the advantages of using namespaces in XHTML documents is that XHTML documents can be extended by including fragments from other XML-based languages such as MathML [41] or SVG [42]. Altering the namespace prefixes of tags in an XHTML document can change the interpretation of the corresponding elements in that document and hence can hinder normal operation of a Web application.

Table 3 summarizes our results on comparing these techniques with WebMTD.

6. Conclusion

In this paper, we present WebMTD, a light-weight defense mechanism for Web applications that is capable of thwarting various types of cross-site scripting (XSS) attacks. Through rigorous evaluation, we show that deploying WebMTD does not affect the overall performance of a Web application, mainly due to its low overhead regarding processing time on both the Web server and the browser, and exchanged HTTP traffic.

In addition to its high performance, WebMTD deployment is affordable and straightforward. It does not require any involvement or knowledge on the developer's side. The

code changes required for deploying WebMTD are applied automatically and in postdevelopment stage. Therefore, it can be applied to both legacy and new Web applications. Moreover, WebMTD does not require Web browsers' modification, because it only relies on the built-in capabilities of modern Web browsers; hence Websites can deploy WebMTD without requiring users to adopt WebMTD-aware Web browsers or install any additional add-on.

Appendix

We have uploaded two versions of `osCommerce v2.3.3.4`, one unmodified version and one WebMTD-enabled version at

`http://149.56.12.232/oscommerce-2.3.3.4/catalog`

and

`http://149.56.12.232/oscommerce-webmtd/catalog`

respectively.

In addition to the inherent vulnerabilities, we added 3 XSS vulnerabilities in

`product_info.php`

page. Avid readers are encouraged to investigate our implementation and evaluate its effectiveness.

Data Availability

The evaluation data used to support the findings of this study are available from the corresponding author upon request.

Disclosure

This submission is an extension of our peer-reviewed conference paper titled "WebMTD: Defeating Web Code Injection Attacks using Web Element Attribute Mutation" which was published in the Proceedings of the 2017 Workshop on Moving Target Defense, CCS 2017. This publication could be accessed online at: https://dl.acm.org/citation.cfm?doid=3140549.3140559.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] P. Bisht and V. Venkatakrishnan, "Xss-guard: precise dynamic prevention of cross-site scripting attacks," in *Proceedings of the International Conference on Detection of Intrusions and Malware*, pp. 23–43, Springer, 2008.

[2] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, p. 6, IEEE, 2006.

[3] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th International Conference on World Wide Web*, pp. 601–610, ACM, 2007.

[4] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*, pp. 921–930, ACM, 2010.

[5] M. Van Gundy and H. Chen, "Noncespaces: Using randomization to defeat cross-site scripting attacks," *Computers & Security*, vol. 31, no. 4, pp. 612–628, 2012.

[6] E. Athanasopoulos, V. Pappas, A. Krithinakis, S. Ligouras, E. P. Markatos, and T. Karagiannis, "xjs: practical xss prevention for web application development," in *Proceedings of the 2010 USENIX Conference on Web Application Development*, pp. 13-13, USENIX Association, 2010.

[7] Binishala, *amazon.com Security Vulnerability*, 2016, https://www.openbugbounty.org/incidents/152371/.

[8] Brute, *ebay.com security vulnerability*, 2016, https://www.openbugbounty.org/incidents/121171/.

[9] *Persistent dom-based xss in https://help.twitter.com via localstorage*, 2018, https://hackerone.com/reports/297968.

[10] L. O'Donnell, https://hackerone.com/reports/297968, 2018.

[11] T. Adams, *Reflected xss in Tooltipy (Tooltips for Wp) Could Allow Anybody to Do Almost Anything an Admin Can*, 2018, https://advisories.dxw.com/advisories/xss-in-tooltipy/.

[12] Akamai, *State of The Internet: Security - Web Attack Report Infographic*, 2018, https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-summer-2018-web-attack-infographic.pdf.

[13] T. Scholte, D. Balzarotti, and E. Kirda, "Quo vadis? A study of the evolution of input validation vulnerabilities in web applications," in *International Conference on Financial Cryptography and Data Security*, pp. 284–298, Springer, 2011.

[14] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM Conference on Computer And Communications Security*, pp. 272–280, 2003.

[15] W3C, *Content Security Policy Level 2*, 2016, https://www.w3.org/TR/CSP2/.

[16] M. Bishop, M. Dilger et al., "Checking for race conditions in file accesses," *Computing Systems*, vol. 2, no. 2, pp. 131–152, 1996.

[17] MITRE, "Cwe-367: Time-of-check time-of-use (toctou) race condition," https://cwe.mitre.org/data/definitions/367.html.

[18] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, vol. 54, Springer Science & Business Media, 2011.

[19] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "An effective address mutation approach for disrupting reconnaissance attacks," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 12, pp. 2562–2577, 2015.

[20] P. Team, *Pax Address Space Layout Randomization (aslr)*, 2003.

[21] A. Klein, "Dom based cross site scripting or xss of the third kind," *Web Application Security Consortium, Articles*, vol. 4, pp. 365–372, 2005.

[22] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*, pp. 124–145, Springer, Berlin, Germany, 2005.

[23] I. Papagiannis, M. Migliavacca, and P. Pietzuch, "Php aspis: using partial taint tracking to protect against injection attacks," in *WebApps' 11: Proceedings of the 2nd USENIX Conference on Web Application Development*, pp. 13–24, USENIX Association, 2011.

[24] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Proceedings of the NDSS*, vol. 2007, p. 12, 2007.

[25] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "SWAP: Mitigating XSS attacks using a reverse proxy," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems, SESS 2009*, pp. 33–39, Canada, May 2009.

[26] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, pp. 330–337, France, April 2006.

[27] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 171–180, ACM, 2008.

[28] M. Ter Louw and V. Venkatakrishnan, "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pp. 331–346, IEEE, 2009.

[29] J. Weinberger, A. Barth, and D. Song, "Towards client-side HTML security policies," in *HotSec*, 2011.

[30] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis, "On the general applicability of instruction-set randomization," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 3, pp. 255–270, 2010.

[31] *W. W. W. C. (W3C), Html5 a vocabulary and associated apis for html and xhtml. W3C recommendation 28 october 2014*, 2014, https://www.w3.org/TR/html5/scripting-1.html.

[32] Enhancesoft, *Osticket - Support Ticket System*, 2016, http://osticket.com/.

[33] osCommerce, *oscommerce*, 2016, http://oscommerce.com/.

[34] I. Automattic, *Wordpress*, 2016, http://wordpress.com/.

[35] Rochen, *Joomla!*, 2016, http://joomla.com/.

[36] R. Hansen, *Xss filter evasion cheat sheet*, 2018, https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

[37] Akamai, *Akamais [State of the Internet] - q3 2016 Report*, 2017, https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q3-2016-state-of-the-internet-connectivity-report.pdf.

[38] Pylo, *Reloadmatic add-on*, 2018, https://addons.mozilla.org/en-US/firefox/addon/reloadmatic/.

[39] M. Weissbacher, T. Lauinger, and W. Robertson, "Why is CSP failing? Trends and challenges in CSP adoption," in *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*, vol. 8688, pp. 212–233, Springer, Berlin, Germany, 2014.

[40] Binishala, *amazon.com Security Vulnerability*, 2016, https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src.

[41] R. Ausbrooks, S. Buswell, D. Carlisle et al., "Mathematical markup language (mathml) version 2.0. w3c recommendation," in *Proceedings of the World Wide Web Consortium*, vol. 2003, 2003.

[42] J. Ferraiolo, F. Jun, and D. Jackson, *Scalable Vector Graphics (SVG) 1.0 Specification*, Iuniverse, 2000.