WebMTD: Defeating Web Code Injection Attacks using Web Element Attribute Mutation

Amirreza Niakanlahiji
UNC Charlotte
aniakanl@uncc.edu

Jafar Haadi Jafarian
University of Colorado Denver
haadi.jafarian@ucdenver.edu

ABSTRACT

Existing mitigation techniques for Web code injection attacks have not been widely adopted, primarily due to incurring impractical overheads on the developer, Web applications, or Web browsers. They either substantially increase Web server/client execution time, 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 a broad class of code injection attacks on Web applications, including cross-site scripting (XSS), HTML code injection, and server-side code injection attacks, in a manner that is transparent to developers, Web applications and browsers. Relying on built-in features of modern Web browsers, WebMTD randomizes 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 and browser code modification. Through rigorous evaluation, we show that WebMTD has very low performance overhead. Also, we argue that our technique outperforms all competing approaches due to its broad effectiveness, transparency, and low overhead. We claim that these qualities make WebMTD an ideal technique for defeating Web code injection attacks on real-world production Web applications.

CCS CONCEPTS

• **Security and privacy → Web application security**;

KEYWORDS

Moving Target Defense; Web code injection attack; XSS attack

1 INTRODUCTION

Despite numerous proposed works on detection and prevention of Web code injection attacks [2, 6, 15, 16, 25, 27], including cross-site scripting (XSS), cross-site request forgery (CSRF), HTML content injection and server-side script injection, they are still one of the most common attack vectors on Web applications; examples are the recently discovered XSS vulnerabilities on Amazon [4] and Ebay [7] Websites. According to OWASP [21], XSS, the most prevalent type of Web code injection attacks, is the third Web application security risk.

Methodologies for countering code injection attacks could be broadly divided into two categories: (I) input validation techniques that prevent injection of malicious code, but are highly susceptible to evasion [23]; and, (II) code differentiation techniques that prevent execution of injected code, including BEEP [15] , ISR [17], CSP [25], Noncespaces [27] and xJS [2]. However, as demonstrated through our extensive evaluation in Section 6, existing techniques suffer from several shortcomings, thus limiting their widespread adoption by real-world applications. Most importantly, almost all existing approaches require either developer involvement in the defense process [15, 25], require modification of Web server or Web browser [2], suffer from high overhead (execution time, more traffic overhead) [15], or are susceptible to evasion [25]. In addition, none of the existing approaches covers all classes of Web injection attacks, covering one or two classes of such attacks.

In Web code injection 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 [5]. TOCTOU is a class of software bugs 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 traditionally attackers have benefited from TOCTOU design flaws [19], 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 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) [14] takes advantage of this gap between time of check (exploit development) and time of use (exploit execution) by randomizing (or mutating [14]) 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, Random Host Mutation (RHM) ensures that exploits built based on scanning at a previous time [13] (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 [26] ensures that exploits that are developed on attacker's side (time of check) are not executable on the target (time of use).

MTD'17, October 30, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-5176-8/17/10…$15.00
https://doi.org/10.1145/3140549.3140559

In this paper, we present an MTD approach, called WebMTD, that defeats a broad number of Web code injection attacks on Web applications, including XSS, CSRF, HTML code injection, and server-side code injection, in a manner that is transparent to the Web application, Web browsers, and developers. WebMTD exploits TOCTOU [5] to differentiate injected code from the application code. By giving attackers a taste of their own medicine, 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 code-injection-resistant by automatically adding new attributes to its HTML elements and new variables to its server-side code blocks. The values of these attributes and variables are generated randomly and keep changing over time. For example, to thwart XSS, an HTML attribute called runtimeId is added to each script block. The value of runtimeId is generated randomly during the processing of a web page; hence two instances of a web page contain two different runtimeId values.

In addition to introducing random variables, WebMTD automatically inserts necessary security check functions into the Web application and instructs Web servers and Web browsers to invoke them before interpreting server/client-side code blocks and rendering HTML elements. These security check functions validate the authenticity of the code blocks and elements.

In summary, WebMTD has the following properties:

- **Preventing a broad class of Web code injection attacks.** It can mitigate all kinds of XSS, CSRF, HTML code injection, and server-side code injection attacks.
- **Low Overhead.** Through rigorous evaluation, we show that our approach has very negligible execution overhead on both the Web server and clients, and also the transmission delay is very negligible. The space overhead is also in order of several hundred bytes.
- **Transparency 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.
- **Transparency 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.

Our contribution in this work can be summarized as the followings:

- We present WebMTD, an approach for defeating Web code injection attacks. In order to make our discussion concrete, we first focus on countering XSS attacks as the primary and most insidious example of Web code injection attacks. In Section 5, we extend our discussion to CSRF, server-side Web code injection and HTML code injection attacks.
- We compare our approach with state-of-the-art anti-Web code injection attacks, including xJS, ISR, BEEP, Noncespaces, and CSP.
- We present a classification of MTD randomization approaches and present our intuition for crafting seamless, effective and affordable MTD techniques for securing applications.

In the rest of paper, first, we present a background on Web code injection 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 XSS attack. Then, in Section 5, we extend our discussion to server-side code injection, CSRF, and HTML code injection attacks. In Section 6, we provide a quantitative evaluation of WebMTD in addition to comparing it with competing anti-XSS solutions. In Section 7, we conclude the paper.

2 BACKGROUND

Generally, in Web code injection 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. A most notorious class of Web code injection attacks is cross-site scripting. 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) [18]. 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 *same origin* policy enforced by modern Web browsers and hence compromises the integrity of the vulnerable Web application.

HTML code injection is another class of Web code injection; which is similar to XSS. In an HTML injection attack, instead of injecting JavaScript code, the attacker injects arbitrary HTML elements on the vulnerable page; hence changing the structure of the Web page. For example, an attacker can inject a new form and entice the victim to provide his personal information.

In a server-side code injection attack (*e.g.*, PHP code injection), the attacker attempts to inject server-side code (*e.g.*, PHP) that will be executed by the respective interpreter on the server. A successful server-side injection yields control of the Website to attackers, including the ability to execute dangerous OS commands.

In a cross-site request forgery (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, injected code will be executed. This code can invoke arbitrary Web API on behalf of the user.

3 RELATED WORK

Methodologies for countering code injection attacks could be broadly divided into two categories: (I) input validation techniques and (II) code differentiation techniques. Input validation techniques use a filtering module that looks for scripting commands or meta-data characters in untrusted inputs, and filter any such input before these inputs are processed by the Web application [6]. 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) [31]. 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 [31]. Notable examples of these approaches are BEEP [15] , ISR [17], CSP [25], Noncespaces [27] and xJS [2].

BEEP [15] 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 make 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.

Gaurav *et al.* introduced Instruction Set Randomization (ISR) [17] 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. Although ISR does not address XSS attacks specifically, its idea can be used to prevent XSS; hence we consider ISR as a related work.

Athanasopoulos *et al.* [2] 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)[25], 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.

Gundy *et al.* introduced Noncespaces technique [27] 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.

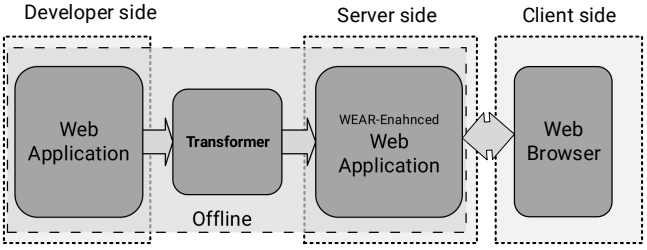

Figure 1: WebMTD Architecture

In the previous approaches, the code differentiation is done by the browser; however code differentiation can be performed on the server-side to prevent CSRF attacks. Synchronizer (CSRF) token is a widely adopted approach to prevent CSRF attacks. It is implemented in many CSRF prevention frameworks and libraries such as OWASP CSRFGuard [24]. The basic idea in synchronizer token is to store a secret token in an HTML document and attach this token to the request generated by its scripts or Web forms. The server-side code verifies the existence and authenticity of the embedded secret token in the receiving request. However, many researchers and security professionals [11, 28] have shown its inadequacy as attackers can retrieve the token and reuse it to make new forged request by launching an XSS attack prior to the CSRF attack. Our approach is similar to the synchronizer token, but as it also prevent XSS and other injection attacks, attackers cannot read and reuse the tokens embedded by WebMTD. This clearly shows why an effective approach must cover all kinds of Web injection attacks.

In Section 6.3, 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 both browser and Web server, and does not rely on Web developers or system administrators. In addition, WebMTD can thwart several other types of Web code injection attacks, including server-side injection and HTML code injection attacks.

4 WEBMTD AGAINST XSS

The root cause of Web code injection attacks is that code interpreters (*e.g.*, PHP interpreter, JavaScript engine) have no reliable means to differentiate between the Web application code and the code injected by an attacker. For example, in PHP code injection, PHP interpreter executes *any* code that is passed by the Web server. Web server, on the other hand, considers any file that contains a PHP block in a Web application directory as a part of that application. Similarly, in XSS attacks, the JavaScript engine executes *any* code that is passed by the Web browser. 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 server or a Web browser to reliably verify the authenticity of a server-side code block (*e.g.*, PHP code block) or a client-side code block (*e.g.*, JavaScript code block) respectively, before letting the code block to be executed by the corresponding interpreters in the server-side or client-side environment.

To achieve this goal, WebMTD, first, must be able to reliably identify the application code blocks. Second, it must be able to

mark these blocks in a manner 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 Web server (to mitigate server-side code injections and CSRF) or Web browser (to mitigate XSS and HTML code injections) to enforce verification of a code block's mark before its execution.

The design of WebMTD is based on two basic observations. First, after development phase and before the end of deployment phase of a Web application is a safe timeframe to identify all code blocks of that application. No new code block will be added after development phase and attackers have not yet had the chance to inject their code blocks. Second, server- and client-side code injection attacks have two separate steps. In the first step, an attacker injects her code; in the second step, she runs the code in the case of server-side code injection, or her victim runs the code in the case of XSS, CSRF, and HTML code injection. If the markings depend on time and not guessable or retrievable, then the attacker cannot replicate them and hence she will be defeated.

Figure 1 shows the overall architecture of WebMTD. The WebMTD transformer is responsible for marking all code blocks in the input Web application, adding the necessary code to make these markings unique over time, and adding security check functions that must be executed by Web servers and browsers to verify the markings. This transformation process is performed before launching the Web application on a production system.

WebMTD prevents the execution of injected codes on both server-side and client-side environments. For the rest of this section, we focus our discussion on prevention of XSS attacks. Later, in the discussion section (Section 5), we discuss how WebMTD prevents the execution of server-side code injection, CSRF and HTML code injection attacks. The current implementation of WebMTD transforms PHP Web applications and works seamlessly with Firefox browser; it does not need to modify Firefox browser code or to install any add-on to extend its functionality. WebMTD relies on `beforescriptexecute` event, which is introduced in HTML5 [29] specification, to execute its security checks on a browser. So far, this feature has been implemented by Firefox, but since it is a part of HTML5 standard, it is expected to be included by other major browsers. One of the reasons that have delayed adoption of this feature by other vendors is that its necessity has been the subject of debate, thus making it a low priority feature for inclusion. We claim that WebMTD is a solid example of its necessity, especially for security purposes.

During transformation phase, WebMTD marks all the code blocks in the Web application to make all the legitimate code blocks recognizable by their respective interpreters. In addition, it adds some security checks that will be invoked by the interpreters to verify the markings. For XSS prevention, WebMTD marks all the `script` elements in a Web application and makes them distinguishable from the script elements that might be injected by attackers at runtime. In order to mark `script` elements, the transformer scans all code files in the Web application and rewrites all start tags of `script` elements. Upon detecting a `script` tag, `WebMTD transformer` inserts a new attribute, which is called `runtimeId`, to the tag. The exact value of this attribute is determined at runtime when the page that contains the script block is requested by a client. In other words, every time a client request a page, a new `runtimeId` value is generated randomly and assigned to the `runtimeId` attributes. The transformer marks both external and inline JavaScript script blocks in the input Web application.

Suppose that the following `script` tag is found in a PHP file:

Code 1: Original script element

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

After rewriting, the above start tag is changed to:

Code 2: Script element with runtimeId attribute

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

In addition to inserting this attribute into each of the script blocks, `WebMTD transformer` embeds the following PHP code block at the beginning of each PHP code file that represents a Web page.

Code 3: Security check function

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

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

Moreover, `WebMTD transformer` locates the place of the `head` element and inserts the following JavaScript block as its first child. Therefore, this inserted block is the first script block that will be interpreted by the browser.

Code 4: Security check function

```
<script type="text/javascript">
// Install an event handler
window.addEventListener('beforescriptexecute',
function(e) {
   var id= "<?php echo $id4trustedBlocks;?>";
   src = e.target.src;
   // Examine the ready-to-execute script block
   if(e.target.hasAttribute("runtimeId") == true && e.target
     .getAttribute("runtimeId")== id){
      return e;
   }
   else{
   // Halt the execution of the script block
      return e.preventDefault();
   }
}, true);
</script>
```

The inserted `script` block installs an event handler for `beforescriptexecute` event. This event is raised before execution of each script block; so the event handler is called before all script blocks on the page, whether they are inline or external. 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 will disallow the execution of that script block.

To illustrate how this can prevent an attacker, without loss of generality, 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 is **59b6d298f36ae**:

Code 5: Attacker's instance

```
<script type="text/javascript">
// Install an event handler
window.addEventListener('beforescriptexecute',
function(e) {
   var id= "59b6d298f36ae";
   src = e.target.src;
   // Examine the ready-to-execute script block
   if(e.target.hasAttribute("runtimeId") == true && e.target
     .getAttribute("runtimeId")== id){
      return e;
   }
   else{
   // Halt the execution of the script block
      return e.preventDefault();
   }
}, true);
</script>
...
<script runtimeId="59b6d298f36ae" type="text/javascript">
...
</script>
```

Then the attacker crafts the following malicious code and injects it to the back-end database by using the XSS vulnerability on the page.

Code 6: Crafted script block

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

After that, a normal user visits the vulnerable web page. The HTML code of the Web page will be

Code 7: User's instance

```
<script type="text/javascript">
// Install an event handler
window.addEventListener('beforescriptexecute',
function(e) {
   var id= "59b6d485c9f5b";
   src = e.target.src;
   // Examine the ready-to-execute script block
   if(e.target.hasAttribute("runtimeId") == true && e.target
     .getAttribute("runtimeId")== id){
      return e;
   }
   else{
   // Halt the execution of the script block
      return e.preventDefault();
   }
}, true);
</script>
...
<script runtimeId="59b6d485c9f5b" type="text/javascript">
...
</script>
...
<script runtimeId="59b6d298f36ae" type="text/javascript" >
[With malicious code]</script>
```

In this HTML document, the `runtimeId` of the injected script block is different from the `runtimeId` that is specified in the security check (var id= "59b6d485c9f5b";) hence the malicious block will not be executed, while the benign 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.

5 EXTENDING WEBMTD TO OTHER ATTACKS

The applications of WebMTD goes beyond XSS to include other types of code injection attacks, such as server-side code injection, CSRF and HTML code injection attacks.

5.1 Server-side Code Injection

To prevent PHP code injection attacks, we add the following statement to the beginning of each PHP block and instruct PHP interpreter to first check for this statement and check whether the value of `$__TokenID__` variable is valid for the running Web application. This variable can be set every time that a Web server is started and the PHP interpreter can be instructed to learn the identifier by adding a property to `php.ini`.

```
$__TokenID__ = '[[placeholder]]';
```

If an attacker injects a PHP block, for example in an image file, and tries to execute it on the server, it would fail because the interpreter detects the missing `$__TokenID__` or with an invalid value, and as a result, does not execute it.

5.2 HTML Code Injection

In order to prevent HTML code injection, we extend WebMTD by adding `runtimeId` attributes to all HTML elements on a Web document. Code 7 is extended by adding a new event handler for body `onload` event. This event handler adds `display:none;` CSS rule to all HTML elements that do not have valid `runtimeId` value. In this way, all injected HTML code will be hidden. Although this implementation can thwart a wide variety of HTML code injection attacks, it has its own drawbacks. A better implementation would be to use `MutationObserver` object which is part of DOM4 specification [30] and is expected to be included in major Web browsers shortly.

5.3 CSRF

To thwart CSRF attacks, all HTTP requests must contain *runtimeId* and the corresponding web method must first check the validity of the provided *runtimeId* for the current session. This approach is quite similar to using synchronizer token, but the main difference is that we also defend against XSS and HTML injection, so the attacker cannot retrieve and reuse *runtimeId* on the fly to bypass the validation check required for CSRF prevention on the server side. This clearly shows why an effective approach must cover all kinds of Web injection attacks.

5.4 Why not SQL injection?

SQL Injection, although usually launched through Web applications, is in essence a Database code injection attack. While MTD has been

Table 1: Inline/External JavaScript Block Usage

Website	No. of inline script blocks	No. of external script blocks	Percentage of inline blocks	Size of inline script blocks (IS)	Size of external script blocks (ES)	Ratio (IS/(IS+ES))
google.com	10	2.9	0.78	30.5 kB	429.9 kB	0.066
youtube.com	11.3	6.4	0.64	37.4 kB	408.3 kB	0.084
facebook.com	13.8	11.75	0.54	70.7 kB	929.6 kB	0.071
baidu.com	12.3	5.1	0.71	17.7 kB	397.5 kB	0.043
yahoo.com	11.2	19.7	0.36	159.5 kB	836.2 kB	0.16
Top 100 sites	14.9	10.1	0.60	43.6 kB	715.4 kB	0.057
osCommerce	8.8	4.6	0.65	16.1 kB	305.9 kB	0.05

used to defeat SQL injection [17], there is a fundamental difference between SQL injection and Web injection attacks, that entails a different MTD approach than those proposed against Web injection attacks.

In general, randomizing system properties for MTD systems could be done in two major ways. Firstly, when a threat model relies on certain system attributes that could be changed safely and without affecting benign operations of the system (*e.g.*, IP addresses for network reconnaissance, address space layout for buffer overflow attacks), these parameters could be changed without requiring the system to inform benign users about such changes. This is significant because in these MTD approaches attackers are distinguished from benign users, not based a secret knowledge, but based on how they use the system. For example, ASLR relies on the fact that benign users of the system do not need to know specific address space positions of key data areas (memory segments), such as the base of the executable. Attackers, on the other hand, would need to know these addresses to craft their buffer overflow exploits. Therefore, these addresses could be randomized seamless to benign users of the system. This seamless randomization is why ASLR is such a widespread and popular technique.

The second type occurs when randomization could *not* be done without affecting benign operations of the system. A primary example of this is SQL injection. This is because, contrary to XSS injection, building a SQL injection query does not depend on any system property that could be safely randomized in a user-oblivious manner. Instead, SQL injection relies on the static nature of SQL commands which are not changeable without synchronization (*e.g.*, shared key) between a Database server and its clients (*e.g.*, Web applications). Here, an alternative intuitive approach, which has also been presented in the literature [17], would be to randomize values of the relevant parameters, but keep the benign users informed of these changes. A primary example of this is instruction set randomization (ISR) techniques [17] which use a key to randomize instruction sets (*e.g.*, machine instructions) of the system. An attacker who does not know the key to the randomization algorithm will inject code that is invalid for that randomized processor/interpreter. But using a key for randomizing commands on server-side and de-randomizing them at client-side is, in essence, closer to traditional cryptography than MTD. Also, it requires secure mechanisms for key generation and distribution among legitimate users, in addition to the cost of decryption on the client-side; these exert substantial implementation and maintenance overhead on the system.

6 EVALUATION

In this section, we present our evaluation of WebMTD. However, in this section, we focus our evaluation on XSS attacks, due to their higher severity and prevalence, as well as lack of space. However, WebMTD is equally effective against other Web code injection attacks and incurs similar overhead. Therefore, we first investigate the effectiveness of WebMTD in preventing exploitation of XSS vulnerabilities in several well-known Web applications. Then, we perform several experiments to evaluate performance overhead of deploying WebMTD. Finally, we compare WebMTD with competing anti-XSS approaches, including BEEP, CSP, xJS, ISR, and Noncespaces.

6.1 Security Effectiveness

WebMTD prevents XSS attacks by obstructing the execution of any injected JavaScript code block on users' browsers. This is achieved by adding a runtimeId attribute to each script block in the web application and validating its value on the user's browser. The runtimeId value is generated randomly and are different on any two instances of a Web page. In other words, runtimeId value is only valid for the Web page instance that it appears in; If an attacker retrieve the runtimeId value from her web page instance and reuse it to craft a script block, the script block will not be valid, as the

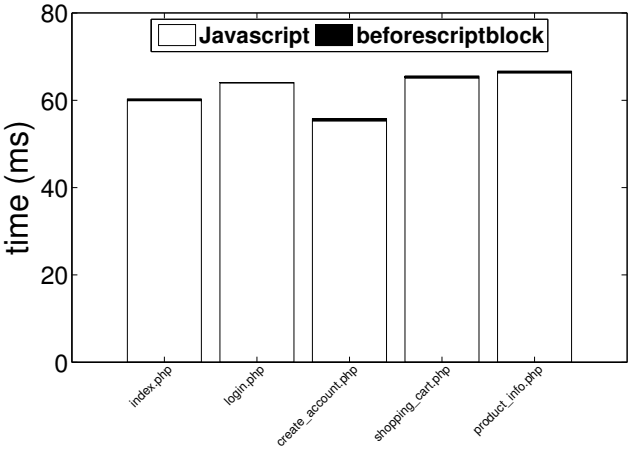

Figure 2: Avg. execution time for original JavaScript code vs. WebMTD code

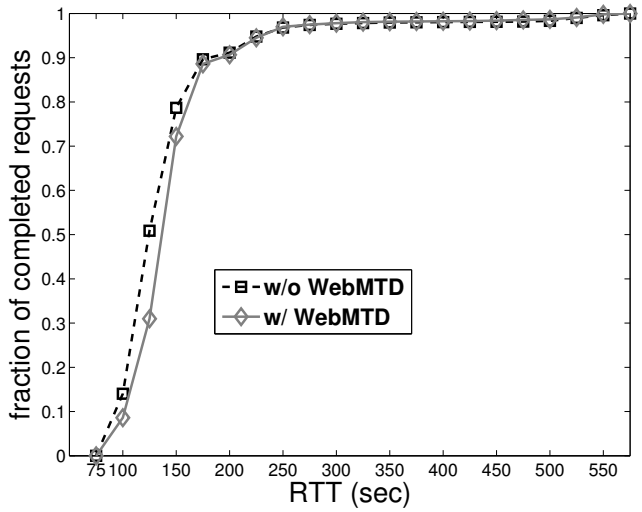

Figure 3: Cumulative Distribution Function of RTTs for 10,000 serial HTTP requests

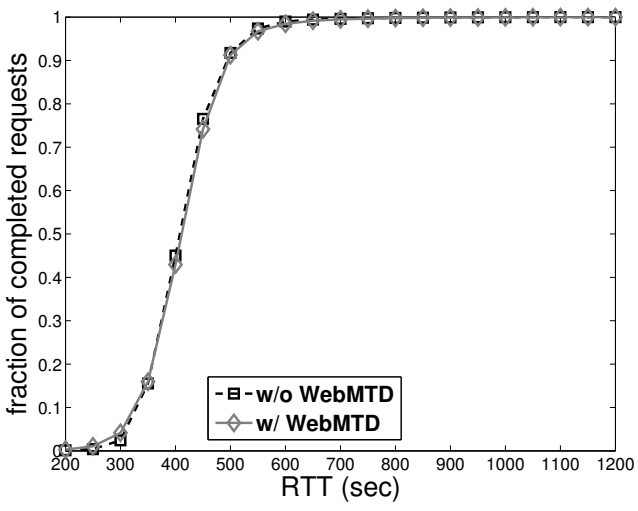

Figure 4: Cumulative Distribution Function of RTTs for 1,000 10-concurrent HTTP requests

runtimeId will not be same to the valid runtimeId value on that new page.

We evaluated WebMTD effectiveness by applying it on a few well-known open source Web applications that have known available XSS attacks. In addition, we have introduced 3 XSS vulnerabilities on one of them to cover all kinds of XSS attacks. We collected 11 confirmed XSS attacks against popular open source Web applications, including 2 exploits on `osTicket` [8], 2 exploits on `osCommerce` [20], 3 exploits on `wordpress` [12] and its plugins, and 4 exploits on `Joomla` and its components [22] from `exploit-db.com` exploit database. In addition, we introduced 3 new XSS vulnerabilities on osCommerce. We examined each of the exploits to ensure that they are valid XSS exploits before testing WebMTD effectiveness against them. WebMTD blocked all of these attacks.

6.2 Performance Overhead

To evaluate the performance overhead of WebMTD, we consider three different metrics: (I) round-trip latency, (II) client-side script execution time, and (III) size of rendered HTML document.

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` [20] on this server. The first one is the original (unmodified) `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 used for measuring performance metrics. We used `Apache JMeter v3.1`, built-in `Performance tool` in `Firefox v47` , and `Firefox ReloadEvery` add-on for this purpose. To ensure that our measurements are realistic, we hosted these two machines in two separate networks across the Internet.

Round-trip Latency. Figure 3 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 a 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 request sent to both versions are essentially the same, the size of HTTP responses might be slightly different due to WebMTD added code to dynamic HTML documents. However, as discussed in Section 6.2, the size of added code for each HTML document is 800 Bytes on average. On a 16 Mbps link (average Bandwidth in USA [1]) this results in $\simeq$ 0.3ms 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 3, for completing 20% of requests, this difference is 11ms, for 50% is 15ms and for 80% is 8 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.

The same argument holds for concurrent requests, as depicted by Figure 4. In this experiment, we sent the same number of requests (10,000), but in 1000 batches of 10 concurrent requests. Note that the CDF functions have even less statistical distance compared to the serial case, due to better utilization of server resources as a result of parallelism.

Script Execution Time. We also calculated the time that is needed for executing the `beforescriptexecute` event handler embedded in the document by WebMTD. This event handler is called

before execution of each script block on each document. To measure this time, we used built-in Performance monitor tool in Firefox and `ReloadEvery` add-on [10]. We randomly chose 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, and (II) `beforescriptexecute` event handler. The former denotes the execution time of the `JavaScript` code blocks, which are the same for both the original and modified versions, while the latter shows the extra time required for execution of WebMTD added codes. The former is a function of a number of JavaScript statements on the execution path, while the latter depends on the number of *executed* code blocks.

Figure 2 shows the average times 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 event handler is significantly negligible. On average handling an `beforescriptexecute` event by WebMTD requires 0.06 ms. Given that the average number of script blocks on modern applications are in order of tens (As shown in 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.

HTML Document Size. The added event handler for `beforescriptexecute` has 13 lines of code, inserting 288 bytes of code in *minified* format to each dynamic HTML document. In addition, it will add a new attribute to each script block on the document. Considering the average number of script blocks on a page (Table 1), WebMTD increases the size of a typical dynamic HTML document only by 1%.

6.3 Comparison with other Techniques

In this section, we compare the state-of-the-art solutions against Web code injection attacks including BEEP, CSP, 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 [31], 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.

CSP is not widely used [32] in modern Web applications because it needs the involvement of the developers or maintainer of the system. They need to define the policies. In addition, it dictates several restrictions on the code such as no inline JavaScript block can be used in the Web application. These restrictions can significantly affect the performance of large Web applications [31] 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 usage of inline script block is among popular Websites. As it is shown in table 1, 0.60 percent of scripts in these Websites are inline and hence solutions such as CSP that require abandoning of inline scripts cannot be used in practice.

ISR could be used to defeat XSS attacks, by randomizing the JavaScript instruction set. However, this approach is very expensive because it requires encryption of JavaScript statements on the server side, and decryption on the browser side. It also needs a key management protocol and both parties must be aware of the randomization.

xJS is an adaptation of ISR for Web applications. However, it only transforms statics HTML documents which are only vulnerable to a few types of XSS attack. Modern Web applications use server-side programming languages such as PHP and JSP to dynamically generate HTML documents. If xJS is used to transform dynamically generated HTML documents in such Web applications, it will not be able to prevent persistent or reflected XSS attacks because the server-side code cannot differentiate between a legitimate script block and an injected script block in a generated HTML document and hence encrypted 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 [3] or SVG [9]. 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 2 summarizes our results on comparing these techniques with WebMTD. We define three metrics for our comparison: (I) overhead (II) backward compatibility, and (III) transparency.

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 define three metrics to capture these overheads and use them in our comparison.

- **Deployment overhead:** it reflects the time and resources that a developer or a system administrator must invest to deploy a solution. Among the solutions, CSP has a high development overhead, mainly because it demands a developer or a system administrator to define a set of policies to deploy the Web application.
- **Execution overhead:** it measures the execution time overhead that a solution imposes on a Web server or browser. In ISR, JavaScript code blocks must be encrypted on the server side and decrypted on the client side. These encryption/decryption operations are costly. In addition, 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 regarding execution time. In BEEP, a Web browser must hash a script block and compare it with a whitelist before being able to execute it. This incurred execution overhead on Web browsers is noticeable by users, especially on mobile devices with low processing power.
- **Space overhead.** It measures the HTTP traffic overhead that is imposed by a solution. All proposed solutions have low space overhead.

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.

Table 2: Comparison of the state-of-the-art anti-Web code injection solutions and WebMTD

	Overhead			Backward compatibility		Transparency	
	Deployment	Execution time	Space (Page size)	Browser	Web Application	Browser modification?	Developer involved?
BEEP	low	high	low	yes	no	yes	yes
CSP	high	low	low	yes	no	yes	yes
ISR	low	high	low	no	yes	yes	no
xJS	low	medium	low	no	yes	yes	no
Noncespaces	low	medium	low	yes	no	yes	yes
WebMTD	**low**	**low**	**low**	**yes**	**yes**	**no**	**no**

- **Web Browser.** This metric indicates whether deploying a solution prevents users with unsupported Web browsers from accessing the Web application. In both ISR and xJS, the browser must decrypt each of the script blocks in the Web document to execute it and hence a Web browser that is not aware of such defense solution cannot load the Web pages correctly.
- **Web Application.** This metric indicates whether a solution can be applied to existing Web applications without affecting their functionalities. CSP does not allow execution of inline JavaScript blocks in a Web application. As shown in table 1, current Web applications are heavily dependent on inline scripting; hence applying CSP to existing Web applications requires a significant change in them. BEEP does not allow execution of script blocks that are generated at runtime by the server-side application. Noncespaces assumes that Web applications do not rely on namespaces and changes the namespace of elements in an XHTML document. However, this can change the semantics of XHTML elements, thus affecting Web application functionalities.

Transparency. Transparency metrics show whether applications on the server and client sides must be changed to enable an anti-Web code injection solution.

- **Browser code modification.** It determines whether a client's Web browser code must be modified. Demanding browser code modification is very prohibitive, because it entails vendor's cooperation for mass adoption. All approaches, except WebMTD, require browser code modification.
- **Developer involvement.** It indicates whether Web developers must adhere to new coding practices while implementing a Web application. To use BEEP, developers must not generate JavaScript code blocks in their server-side code. To use CSP, they must not use inline JavaScript code blocks. To use Noncespaces, the developer must avoid using of namespaces for XHTML element names. These requirements mean these approaches cannot be applied to the legacy Web application without requiring code refactoring by developers.

6.4 Limitations

WebMTD can prevent code injection attacks when it is implemented completely (i.e. by enforcing validation of all HTML elements and server-side code blocks), otherwise the system would still be susceptible to some type of attacks. For example, if we only implement the XSS part only, an attacker can inject inline script to launch XSS attack. In inline script, an attacker inject an HTML element such as `img` instead of injecting a script block. As an another example, if we only verify client side elements, an attacker may try file inclusion attack to override PHP files to remove client side checkings.

7 CONCLUSION

In this paper, we present WebMTD, a light-weight defense mechanism for Web applications that is capable of thwarting a broad class of Web code injection attacks such as XSS, CSRF and server-side code injection 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. It adds about 800 bytes to a typical HTML document with 25 script blocks. Moreover, on average it increases the execution time of each script block by 0.06 ms. 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 implanting WebMTD are applied automatically and in post-development 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.

A APPENDIX I

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.

REFERENCES

[1] Akamai. 2017. akamai's [state of the internet] - Q3 2016 report. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q3-2016-state-of-the-internet-connectivity-report.pdf. (2017).

[2] Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P Markatos, and Thomas Karagiannis. 2010. xJS: practical XSS prevention for web application development. In *Proceedings of the 2010 USENIX conference on Web application development*. USENIX Association, 13–13.

[3] Ron Ausbrooks, Stephen Buswell, David Carlisle, Stéphane Dalmas, Stan Devitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, et al. 2003. Mathematical markup language (MathML) version 2.0 . W3C Recommendation. *World Wide Web Consortium* 2003 (2003).

[4] binishala. 2016. amazon.com Security Vulnerability. https://www.openbugbounty.org/incidents/152371/. (2016).

[5] Matt Bishop, Michael Dilger, et al. 1996. Checking for race conditions in file accesses. *Computing systems* 2, 2 (1996), 131–152.

[6] Prithvi Bisht and VN Venkatakrishnan. 2008. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 23–43.

[7] Brute. 2016. ebay.com Security Vulnerability. https://www.openbugbounty.org/incidents/121171/. (2016).

[8] Enhancesoft. 2016. osTicket - Support Ticket System. http://osticket.com/. (2016).

[9] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. 2000. *Scalable vector graphics (SVG) 1.0 specification*. iuniverse.

[10] Jaap Haitsma. 2017. ReloadEvery Add-on. https://addons.mozilla.org/enUS/firefox/addon/reloadevery/. (2017).

[11] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. 2012. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 760–771.

[12] Automattic Inc. 2016. WordPress. http://wordpress.com/. (2016).

[13] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. 2015. An Effective Address Mutation Approach for Disrupting Reconnaissance Attacks. *IEEE Transactions on Information Forensics and Security* 10, 12 (2015), 2562–2577.

[14] Sushil Jajodia, Anup K Ghosh, Vipin Swarup, Cliff Wang, and X Sean Wang. 2011. *Moving target defense: creating asymmetric uncertainty for cyber threats*. Vol. 54. Springer Science & Business Media.

[15] Trevor Jim, Nikhil Swamy, and Michael Hicks. 2007. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*. ACM, 601–610.

[16] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 6–pp.

[17] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 272–280.

[18] Amit Klein. 2005. DOM based cross site scripting or XSS of the third kind. *Web Application Security Consortium, Articles* 4 (2005), 365–372.

[19] MITRE. [n. d.]. CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition. https://cwe.mitre.org/data/definitions/367.html. ([n. d.]).

[20] osCommerce. 2016. osCommerce. http://oscommerce.com/. (2016).

[21] OWASP. 2013. Top 10: Ten Most Critical Web Application Security Risks. (2013).

[22] Rochen. 2016. Joomla! http://joomla.com/. (2016).

[23] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. 2012. *Quo Vadis? A Study of the Evolution of Input Validation Vulnerabilities in Web Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 284–298. https://doi.org/10.1007/978-3-642-27576-0_24

[24] Eric Sheridan. 2011. OWASP: CSRFGuard Project. (2011).

[25] Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*. ACM, 921–930.

[26] PaX Team. 2003. PaX address space layout randomization (ASLR). (2003).

[27] Matthew Van Gundy and Hao Chen. 2012. Noncespaces: Using randomization to defeat cross-site scripting attacks. *computers & security* 31, 4 (2012), 612–628.

[28] B Vibber. 2014. CSRF token-stealing attack (user. tokens). *Mozilla, a ticket* (2014).

[29] World Wide Web Consortium (W3C). 2014. HTML5 A vocabulary and associated APIs for HTML and XHTML. W3C Recommendation 28 October 2014. https://www.w3.org/TR/html5/scripting-1.html. (2014).

[30] World Wide Web Consortium (W3C). 2015. W3C DOM4. W3C Recommendation 19 November 2015. https://www.w3.org/TR/dom/. (2015).

[31] Joel Weinberger, Adam Barth, and Dawn Song. 2011. Towards Client-side HTML Security Policies. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security (HotSec'11)*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=2028040.2028048

[32] Michael Weissbacher, Tobias Lauinger, and William Robertson. 2014. Why is CSP failing? trends and challenges in CSP adoption. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 212–233.