International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

Automation of Mutated Cross Site Scripting

Anchal Tiwari[1], **J. Jeysree**[2]

[1]Information Security and Cyber Forensics, Masters in Technology, SRM University Chennai, India

[2]Department of Information Technology, Assistant Professor, SRM University Chennai, India

Abstract: ***Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it. In browsers Mutation event occur when there is a change in the DOM Structure of the browsers. There are various ways in which DOM structure could be changed among which innerHTML property is discussed specifically. mXSS is a new class of XSS vectors, the class of mutation-based XSS (mXSS) vectors, which may occur in innerHTML andrelated properties. mXSS affects all three major browserfamilies: IE, Firefox, and Chrome.mXSS could be placed in major browser families and effecting major web applications. In this paper we apply the idea of mutation-based testing technique to generate adequate test data sets for testing XSSVs. Our work addresses XSSVs related to web-applications that use PHP and JavaScript code to generate dynamic HTML contents. Finally there would be the development of an automatic tool which would generate mutants automatically, automatically testing the web application and finally giving the output.***

Keywords: Cross Site Scripting, Mutation in DOM, innerHTML propriety

1. Introduction

Cross Site Scripting (XSS) is one of the worst vulnerabilities in web-based applications XSS Vulnerabilities (XSSVs) involves the generation of dynamic Hyper Text Markup Language (HTML) contents (*i.e.*, attributes of tags) with invalidated inputs. XSS attacks exploit the vulnerabilities through inputs that might contain HTML tags, JavaScript code, and so on. These inputs are interpreted by browsers while rendering web pages. As a result, the intended behaviour of generated web pages alters through visible (*e.g*

Creation of pop-up windows) and invisible (*e.g.*, cookie bypassing) symptoms. In biological terms mutationis a permanent change of the nucleotide sequence of the genome of an organism. In DOM Structure *Mutation* is an ownership kind of optimization of HTML code implemented differently in each of major browsers. Mutation events occur in following ways:

DOMSubtreeModified
DOMNodeInserted
DOMNodeRemoved
DOMNodeRemovedFromDocument
DOMNodeInsertedIntoDocument
DOMAttrModified
DOMCharacterDataModified

All these modification is done by using various functions one of them is innerHTML. innerHTML is a DOM node's property that gets or sets the inner HTML code of an HTML element. It is commonly used in JavaScript to dynamically change or read from a page. Server- and client-side XSS filters share the assumption that their HTML output and the browser-rendered HTML content are mostly identical. In this paper, we show how this premise is *false* for important classes of web applications that use the inner HTML property to process user-contributed content. Instead, this very content is *mutated* by the browser, such that a harmless string that passes nearly all of the deployed XSS filters is subsequently transformed into an active XSS attack vector by the browser layout engine itself.

The information flow of an mXSS attack is shown in Figure 1. The attacker carefully prepares an HTML or XML formatted string and injects it into a web application.

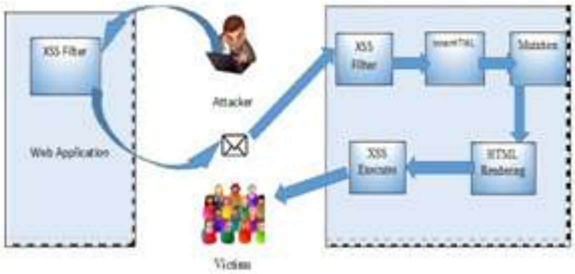


Figure 1: Information flow in an mXSS attack.

This string will be altered or even rewritten in a server-side XSS filtered, and will then be passed to the browser. If the browser contains a client-side XSS filtered, the string will be checked again. At this point, the string is still harmless and cannot be used to execute an XSS attack. However, as soon as this string is inserted into the browser's DOM by using the innerHTML property, the browser will *mutate* the string. This mutation is highly unpredictable since it is not part of the specified innerHTML handling, but is a proprietary optimization of HTML code implemented directly in each of the major browser families. The mutated string now contains a valid XSS vector, and the attack will be executed on rendering of the new DOM element. Both server and client side filters were unable to detect this attack because the string scanned in these filters did not contain any executable code.

A web application is vulnerable if it inserts user-contributed input with the help of innerHTML or related properties into the DOM of the browser. It is difficult to statistically evaluate the number of websites affected by mutated xss, since automated testing fails to reliably detect all these attack prerequisites. If innerHTMLis only used to insert trusted code from the web application itself into the DOM, it is not vulnerable. However, it can be stated that Problem Descriptionamongst the 10.000 most popular web pages, roughly one third uses the innerHTML property, and about 65% use Java- Script libraries like jQuery, who abet mXSS attacks by using the *innerHTML*property instead of the corresponding DOM methods.

2. Problem Description

2.1 The innerHTML Property

The use of *innerHTML*and *outerHTML*is supported by each and every one of the commonly used browsers in the present landscape. Consequently, the W3C started a specification draft to unify *innerHTML* rendering behaviours across browser implementations. An HTML element's *innerHTML*property deals with creating HTML content from arbitrarily formatted strings on write access on the one hand, and with serializing HTML DOM nodes into strings on read access on the other.

Syntax to get innerHTML
var content = element.innerHTML; /* To get the inner HTML of an element */
Where,content contains the serialized HTML code describing all of the element's descendants.
Syntax to set innerHTML
element.innerHTML = content; /* To set the inner HTML of an element */

To use *innerHTML*, the DOM interface of element is enhanced with an innerHTML attribute/property. Setting of this attribute can occur via the *element*.innerHTML= *value* syntax, and in this case the attribute will be evaluated immediately. A typical usage example of innerHTML is shown in Listing 1: when the HTML document is first rendered, the <p> element contains the "First text" text node. When the anchor element is clicked, the content of the <p> element is replaced by the "New <b>second</b> text." HTML formatted string.

Example on inner HTML usage

```
<script type =" text / javascript ">
var new = "New <b>second <\/b> text .";
function Change ()
{
document .all. myPar .innerHTML = new;
}
</script >
<p id =" myPar "> First text .</p>
<a href =" javascript : Change ()">
Change text above !
</a>
```

Browser fixes code before adding it to the DOM! This can be useful if the programmer wrote incorrect code because the browser fixes the code first, but it’s also very useful for attackers

2.2 Attack Vector and Attack body

In general, an mXSS attack can be separated into the attack vector and the attack body an attack body is the main code for executing the intention (e.g., it can invoke JavaScript interpreter) after exploiting a vulnerability successfully, and it is often applied by obfuscation techniques beyond the detections. An attack vector is the medium for introducing the attack body. If imagining a XSS exploit as a missile, the attack vector is like the guided device of the missile, and the attack body is like the warheadof the missile. Hence, an attacker can promote the attack body to be interpreted for malicious intension by using the right or efficient attack vectors.

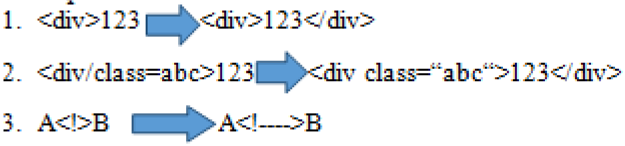
Table 1: The samples of mutated XSS attack (The *attack vectors* are separated by commas, and the *attack bodies* are denoted as italic.)

Mutated XSS Attack Samples	
1.	”>,alert(123)<*iframe/src=http://xssed.com*>alert(123)</scrihttp://pt>alert(123)
2.	”>,’></div>alert(123)<input><*script*>*alert(123)*</*script*></marquee>alert(123)”>
3.	>”>,</p>alert(123)<marquee><*script*>*alert(123)*</*script*></title>alert(123)
4.	”/>,</ScRiPt>alert(123)<title><*script*>*alert(123)*</*script*></SCRIPT>alert(123)
5.	>”>,</form>alert(123)<b><script>alert(123)</script></input>alert(123)” t type=”hidden” />

3. Work Around Problem

Following are the example which proves the existence of mutation and confirms the possibility of mXSS attacks. This section describe a set of *innerHTML*-based attacks it is discovered during the research on DOM mutation and string transformation. The code is presented purposefully appearing as sane and inactive markup before the transformation occurs, while it then becomes an active XSS vector executing the example method xss() after that said transformation. This way server and client-side XSS filters are being elegantly bypassed.

Examples:



1. <div>123 → <div>123</div>
2. <div/class=abc>123 → <div class=“abc“>123</div>
3. A<!>B → A<!---->B

4. Vulnerable code:
   .innerHTML= „ ..<imgclass=“INPUT“>1234 ..“;
   After fixing:
   <imgclass=„input“>1234</img>
5. Attacker input:
   ´´src=x onerror=alert(1)
   The generated code:
   <imgclass=“´´src=x onerror=alert(1)“>1234
6. The generated code:
   <imgclass=“´´src=x onerror=alert (1)“>1234
   Now the code gets fixedbefore it is added to the DOM by .innerHTML. Browser notice that there are already ´´to enclose the class, thus ““ can be removed!

The fixed code:

```
<imgclass=´´src=xonerror=alert(1) >1234</img>
```

It‘s possible to execute JS-code even if “getsencoded.

Figure 2: Proof of concept for mXSS attack

7. innerHTML-access to an unknown element causes mutation and unsolicited JavaScript excution

```
<!-- Attacker Input -->
<article xmlns ="urn:imgsrc=x onerror=xss()//" >123
<!-- Browser Output -->
<imgsrc=x onerror=xss()//:article xmlns="urn:imgsrc=x
onerror=xss()//" >123 </ imgsrc=x onerror =xss () //: article
>
```

The result of this structural mutation and the pseudo-namespace allowing white-space is an injection point. It is through this point that an attacker can simply abuse the fact that an attribute value is being rendered despite its malformed nature, consequently smuggling arbitrary HTML into the DOM and executing JavaScript.

4. Attack Surface

The attacks outlined in this paper target the client-side web application components, e.g. JavaScript code, that use the *innerHTML*property to perform dynamic updates to the content of the page. Rich text editors, web email clients, dynamic content management systems and components that pre-load resources constitute the examples of such features.

In this section the conditions under which a webapplication is vulnerable is described in detail. The basic conditions for a mutation event to occur are the serialization and deserialization of data. As mentioned earlier, mutation in the serialization of the DOM-tree occurs when the *innerHTML*property of a DOM-node is accessed. Subsequently, when the mutated content is parsed back into a DOM-tree, e.g. when assigned to *innerHTML*or written to the document using document.write, the mutation is activated.

In order for an attacker to exploit such a mutation event, it must take place on the attacker supplied data. This condition makes it difficult to statistically estimate the number of vulnerable websites, however, the attack surface can be examined through an evaluation of the number of websites using such vulnerable code patterns.

vulnerable code patterns

```
// Native JavaScript / DOM code
a. innerHTML = b. innerHTML ;
a. innerHTML += 'additional content ';
a. insertAdjacentHTML (' beforebegin ', b.
innerHTML );document . write (a. innerHTML );
// Library code
$( element ). html (' additional content ');
```

Note though that almost all applications applied with an editable HTML area are prone to being vulnerable.

5. Automation

Automation technique could be performed by observing the client side code and then searching for the DOM rendering elements specifically for innerHTML. The code will be crawling the website and fetching the vulnerable innerHTML prosperity and attacking the input field with the mutants. The successful attack will be reported to the tester/attacker.

6. Future Work

As the future progress of this research this tool could be intergrated with the modern web vulnerability testing tools. Which would ensure total security with this novel attack.

7. Conclusion

The paper describes a novel attack technique based on a problematic and mostly undocumented browser behaviour that has been in existence for more than ten years initially introduced with Internet Explorer 4 and adopted by other browser vendors afterwards.The discussed browser behaviour results in a widelyusable technique for conducting XSS attacks against applications otherwise immune to HTML and JavaScript injections. These internal browser features transparently convert benign markup, so that it becomes an XSS attack vector once certain DOM properties such as *innerHTML*and *outerHTML*are being accessed or other DOM operations are being performed. As this kind of attack is labelled as Mutation based XSS (mXSS), the paper is dedicated thoroughly by introducing and discussing this very attack.Subsequently, it is analysedthat the attack surface and an action plan is proposed for mitigating the dangers via several measurements and strategies for web applications, browsers and users, while server- as well as client-side XSS filters have become highly skilled protection tools to cover and mitigate various attack scenarios, mXSSattacks pose a problem that has yet to be overcome by the majority of the existing implementations. A string mutation occurring during the communication between the single layers of the communication stack from browser to web application and back is highly problematic. Given its place and time of occurrence, it cannot be predicted without detailedcase analysis.

References

[1] Mario Heiderich “mXSS Attacks: Attacking well secured Web-Applications by using innerHTMLMutations”

[2] Yi-Hsun Wang “Structural Learning of Attack Vectors for Generating Mutated XSS Attacks”

[3] Hossain Shahriar and Mohammad Zulkernine ”*MUTEC*: *Mu*tation-based *Te*sting of *C*ross Site Scripting”

[4] Ben Stock “Precise Client-side Protection against DOM-based Cross-Site Scripting” [Online].

Available:https://www.usenix.org/conference/usenixsecurity14/technical sessions/presentation/stock

[5] Andrea Avancini, Mariano Ceccato"Security Testing of Web Applications: A Search Based Approach for Cross-Site Scripting Vulnerabilities"