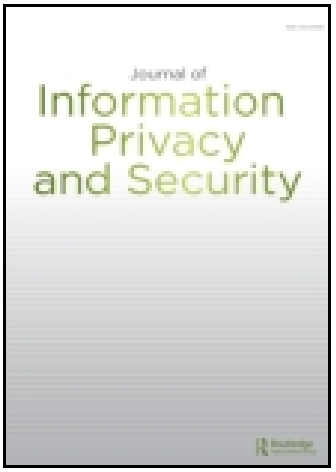

Journal of Information Privacy and Security
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/uips20

Cross-Site Scripting (XSS) Abuse and Defense: Exploitation on Several Testing Bed Environments and Its Defense

B. B. Gupta[a], S. Gupta[a], S. Gangwar[a], M. Kumar[a] & P. K. Meena[a]
[a] National Institute of Technology Kurukshetra, India
Published online: 07 Jul 2015.

To cite this article: B. B. Gupta, S. Gupta, S. Gangwar, M. Kumar & P. K. Meena (2015) Cross-Site Scripting (XSS) Abuse and Defense: Exploitation on Several Testing Bed Environments and Its Defense, Journal of Information Privacy and Security, 11:2, 118-136, DOI: 10.1080/15536548.2015.1044865

To link to this article: http://dx.doi.org/10.1080/15536548.2015.1044865

Journal of Information Privacy and Security, 11: 118–136, 2015

ISSN: 1553-6548 print / 2333-696X online
DOI: 10.1080/15536548.2015.1044865

Routledge
Taylor & Francis Group

Cross-Site Scripting (XSS) Abuse and Defense: Exploitation on Several Testing Bed Environments and Its Defense

B. B. Gupta, S. Gupta, S. Gangwar, M. Kumar, and P. K. Meena
National Institute of Technology Kurukshetra, India

Today cyber physical systems (CPS) facilitate physical world devices to integrate with several Internet data sources and services. In the contemporary era of Web 2.0 technologies, web applications are being developed on several advanced technologies (e.g., AJAX, JavaScript, Flash, ASP.net). However, due to the frequent usage in daily life, web applications are constantly under attack. Cross-site scripting (XSS) attacks are presently the most exploited security problems in the modern web applications. XSS attacks are generally caused by the improper sanitization of user-supplied input on the applications. These attacked use vulnerabilities in the source code, resulting in serious consequences such as stealing of session-identifications embedded in cookies, passwords, credit card numbers, and several other related personal credentials. This article describes a three-fold approach: 1) testing the vulnerabilities of XSS attack on the local host server Apache Tomcat by utilizing the malicious scripts from XSS cheat sheet website; 2) exploiting the same vulnerabilities on Web Goat; and 3) exploiting encoded versions of the injected scripts for testing the level of XSS attack prevention capability. Based on the observed results, further work is also discussed.

Today modern web applications developed under advanced technologies (e.g. JavaScript [Flanagan, 2001], ASP.net [MacDonald & Szpuszta, 2005]) are changed into complex programs. Moreover these complex programs are now no longer restricted to execution on the web server-side. Also web applications incorporate a considerable amount of JavaScript code that is to be transferred and executed on the client-side web browser. However, web applications are becoming the best possible target for the attackers for performing several malicious activities, such as session hijacking and stealing session identifications (IDs). According to several reports published in common vulnerabilities exposure (CVE) database (CVE, 2007), web application flaws are the leading top most vulnerabilities.

Today the whole world is like cyber village—a lot of information and confidential data that reside on the websites and online servers. But unfortunately web applications are exposed to several varieties of security risks (Howard & LeBlanc, 2005; Livshits & Erlingsson, 2007). Recently, it has been observed that poorly developed software produces protection concerns. The quantity

Correspondence should be addressed to B. B. Gupta, Department of Computer Engineering, National Institute of Technology Kurukshetra, Kurukshetra-136119, India. E-mail: gupta.brij@gmail.com

of viruses that could produce web security concerns is directly related to the dimension and complexity of installed web applications and server. Mostly, all multifaceted programs either have wide variety of loopholes or some sort of already stored weaknesses In addition to these points, web servers are intrinsically multifaceted programs. Websites are also multifaceted and deliberately request even greater communication with the community of global users. Thus, the chances of security loopholes are abundant and emergent. Cross-site scripting (XSS) is one of the most dangerous and most exploited attacks in the recent times (Athanasopoulos, Krithinakis, & Markatos, 2010; Fogie et al., 2006). According to the recent survey done by CVE, the XSS attack is affecting the highest number of websites (Figure 1). Almost 65% of websites are exposed to XSS vulnerabilities discovered in modern web applications.

XSS attacks are the vulnerabilities found in the recent websites in which a malicious attacker run the malicious JavaScript code in the web browser of client's machine to access its sensitive credentials (Alcorna, 2006; Arulsuju, 2011; Gupta, 2012). In October 2005, the "Samy XSS worm" turned out to be the first major XSS worm to exploit XSS vulnerabilities for the dissemination of several bugs (Mook, 2005). Suddenly, the Samy XSS worm infected more than 1 million individual user accounts on MySpace.com, which is considered to be one of the most well-liked social-media websites. The Samy worm tainted the website by utilizing the JavaScript viral cipher, thus preparing Samy.

It is very significant to investigate the amount of the risk related with XSS malware vulnerabilities and the procedures that several organizations can use to guard themselves against XSS attacks and their trusted users, particularly when the XSS malware instigates from trusted web applications and several other trustworthy locations (Grossman, Hansen, & Fogie, 2007). The main contributions of this article are:

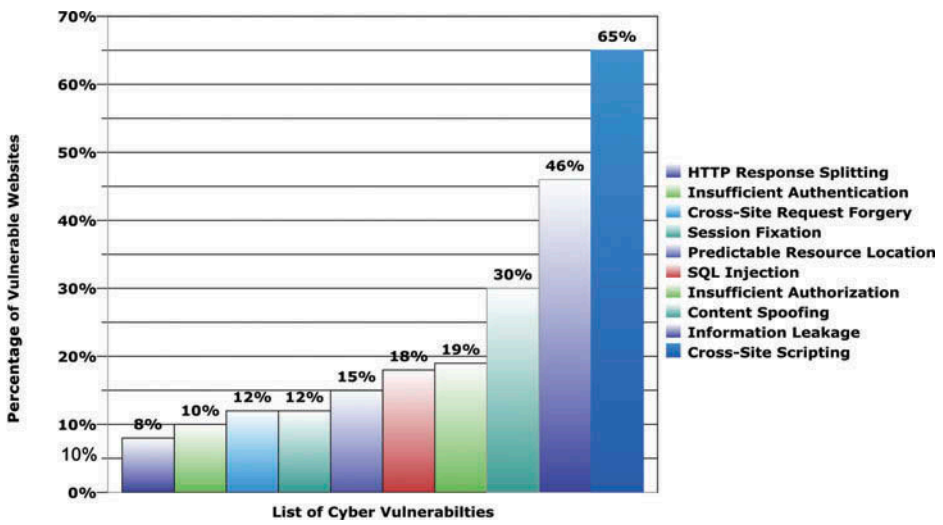

FIGURE 1 Percentage probability of websites vulnerable by different class of cyber vulnerabilities.

- Presented a detailed background on cross-site scripting (XSS) attacks and their categories.
- Focus on the exploitation techniques of reflected and stored XSS attacks.
- Present a detailed literature survey on the detection and mitigation techniques of XSS attacks.
- Test and exploit the vulnerabilities of XSS attack by injecting real-world malicious scripts on different testing bed environments such as Apache Tomcat Server, OWASP Web Goat, and Net Beans.
- Test and evaluate the capability of encoded version of scripts for mitigating the effect of XSS vulnerabilities on web applications.
- Based on the observed results, discuss future research work for the requirement of mitigation of XSS vulnerabilities.

BACKGROUND AND OVERVIEW OF XSS ATTACKS

The key requirement to recognize related to XSS attacks is that they are extremely the most widespread vulnerability discovered in conventional web applications, located in more than 80% of all web applications (Barhoom & Kohail, 2011; Faghani & Saidi, 2009). Although XSS has been well thought-out to be temperate, it is a harsh vulnerability for most of the web applications (Grossman et al., 2007). Once the XSS vulnerabilities are discovered during the software development lifecycle by Web programming developers and numerous security professionals, the next job of these security professionals is to propose some defensive methodologies for mitigating these vulnerabilities.

XSS is an assault method that tricks a website to respond to the attacker-supplied executable code, which executes in domain of a user's web browser (Gupta & Sharma, 2012; Gupta & Gupta, 2015). This approach clearly reflects that the user is the deliberate victim, with the attacker utilizing the exposed website. Normally, the XSS exploits a malicious script code, normally developed in HTML or JavaScript platform that will not run on the web server (Gupta & Gupta 2014). The web server is simply the host party, while the actual XSS attack runs within the victim's browser. Moreover, the XSS attack permits the theft of session-IDs embedded in cookies, which can then be reprocessed to session hijack user accounts. Online user accounts incorporate Internet banking, blogs, and numerous other web application features, which are accessed with the credentials of victim (e.g., user-IDs, passwords, credit card numbers). Recent studies on XSS vulnerabilities have also exposed that XSS vulnerabilities can seize whole access over the web browser. Figure 2 explains the exploitation of XSS vulnerability on the web browser of victim domain.

Here the attacker injects the malicious script as shown in the Step 1 of Figure 2 to the victim's domain. Later, the attacker tricks the victim to click on this malicious link as posted by the attacker. As a result the request for this malicious web page is transferred to the vulnerable web server of victim's domain. The vulnerable web server replies with the HTTP response web page incorporating the name of the injected script. Lastly this malicious script will be executed on the victim's web browser and transmit the sensitive credentials of the victim to the attacker's domain. As a result, the XSS attack is exploited on the victim's domain.

Normally, there are two techniques for victim users to become tainted by XSS attacks. Victims of XSS attack are either curious into clicking on a malicious crafted hyperlink (termed a *reflected XSS attack*) or they are inadvertently assaulted by just browsing a vulnerable web page injected

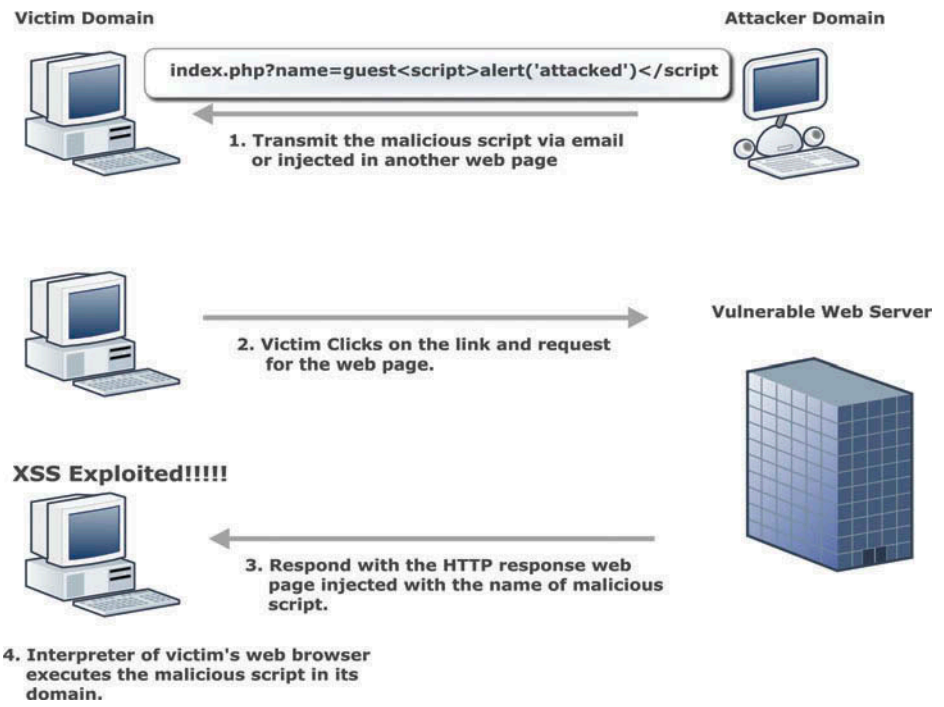

FIGURE 2 XSS attack exploitation on victim domain.

with the poorly written malicious code (termed a *stored XSS attack*). It has also been discovered that a victim's web browser need not to be prone to any well-known susceptibility. Therefore no level of amplification will aid web users, and the user will be reliant on a website's safety methods for online security. Software developers and security proficient's developing the web applications are the solutions to block this completely preventable attack (Livshits & Erlingsson, 2007).

Reflected XSS Attack

Reflected XSS attack (also termed a *non-persistent XSS attack*) involves a victim user browsing the malicious hyperlink crafted by the attacker. As soon as the victim user clicks on the hyperlink, the malicious injected code will run by the web browser of the victim. *Non-persistent*, as the name indicates, means that the poorly written injected script is not everlasting and merely visible for the short duration, as long as the victim user is browsing the web page. Figure 3 illustrates the exploitation of reflected XSS attack on the end user domain.

Initially the user's web browser will browse the attacker's website and somehow trick the end user to click on the malicious link. As a result, the request for the associated web page is passed to the vulnerable web server of end user domain. Since the vulnerable web server does not possess the requested resource (i.e., malicious script), it returns an HTTP error response message to the

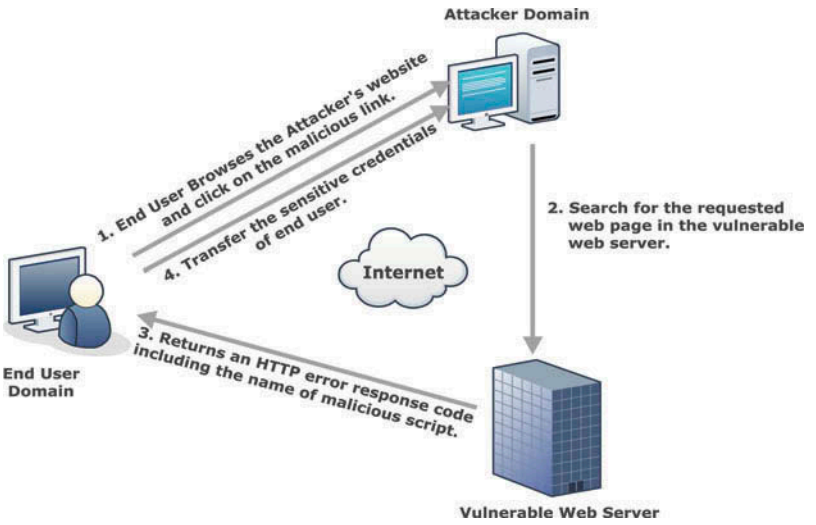

FIGURE 3 Reflected XSS attack exploitation on victim's domain.

web browser of end user, incorporating the name of malicious injected script. As a result, the malicious script will be executed by the web browser of end user and will transfer the sensitive credentials (e.g., cookies, stored passwords, credit card numbers) to the attacker's domain. Later, the attacker will use these credentials to hijack the end user session.

Earlier, in the years 2005 and 2006, non-persistent XSS vulnerabilities in the search engine of Google were discovered. Therefore, it is clearly reflected how a trustworthy Google search engine can be exploited to steal sensitive information through reflected XSS attack. In addition, in 2006, the famous e-commerce website PayPal had perhaps permitted the assailant to steal the sensitive credentials of users (e.g. transaction passwords, credit card numbers) from its members for the duration of almost 2 years.

Stored XSS Attack

In stored XSS attack (also known as *persistent XSS attacks*) malicious scripts are injected permanently in the source code of the website (Hallaraker & Vigna, 2005). Figure 4 illustrates the scenario of exploitation for a stored XSS attack. Here the attacker injects the malicious script permanently on the repository of vulnerable web server. Later, the end user browses the web application of this vulnerable web server and clicks on this malicious link. Finally the malicious script will be executed on the end user's domain and sensitive credentials will be automatically transferred to the attacker's domain.

Persistent XSS vulnerability is an observable weakness that should have been located from the start; however, just similar to XSS on other web applications, it was failed to spotted. Moreover, if not spotted accurately, this class of vulnerability would affect all PHP-based web applications along with several forums and everywhere where the user ID was reflected on the GUI of website. A superior position from which to search for this vulnerability is the crucial forum scripts that

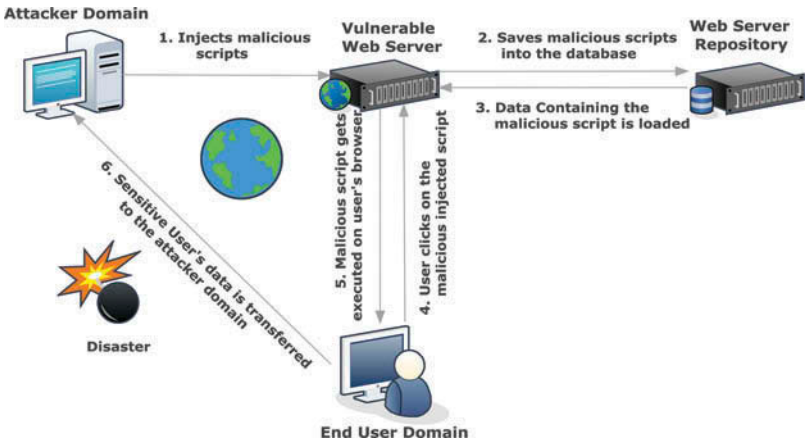

FIGURE 4 Stored XSS exploitation on the end user domain.

web application developers have created for themselves or discovered from websites considered to aid beginners (Forrest et al., 1996).

RELATED WORK

In a recent survey, a scheme was proposed that focuses on detecting the vulnerabilities of XSS attacks by performing a static as well as dynamic source code analysis of the web applications (Lucca et al., 2004). Static analysis of the source code of web applications was proposed to identify the possible vulnerabilities in web pages. In another approach, dynamic source code analysis was generally performed to discover the genuine vulnerabilities of web pages derived from static analysis. The authors introduced a technique that is composed of control flow graph (CFG) of information that is utilized by web server page. This graph consists of input and output nodes. Generally input nodes are those nodes related to the statements that collect the input data from a user-submitted HTML forms, extracting the value of query string, cookies, or any other source of input. Output nodes are related to those statements that generally perform write operations on a database field, files, or cookies, or displays the output on the client web page. According to this method, a web server page is likely to be vulnerable if there exits an input message (possibly provided by the user), an input node, and an output node such that all the CFG paths will leave the input node and enter into the output node. In contrast, a web server page including an input message is safe if the output of the web page will not change with respect to the input message.

This technique helps in investigating the only fixed spots of XSS attack vectors in the source code of web applications; however some of the vulnerable spots remain unnoticed by this technique. Moreover, this method also results in false positive warnings. For example, suppose a web server page extracts the input data from the user side and stores into its database. The static analysis of this technique somehow declares that this web page is potentially not safe. But later, on some other web page, this stored input data are read and these data are incorporated in an output web page in encoded form. As a result, now this output web page cannot be declared as vulnerable.

In another recent survey (Cao et al., 2012), several real-world XSS worms on modern and popular web applications are discussed. Most of these XSS worms exploit the vulnerabilities of stored XSS attack on real world web applications (e.g., Facebook, Twitter). The first XSS worm—*the Samy worm* —was found in October, 2005 in MySpace web application (Mook, 2005; Samy 2005). It affected more than 1 million users in just 24 hours. The Orkut *Born Sabado virus* is a seriously disguised XSS worm. During the course of its occurrence, an Internet user account on Orkut was tainted when the user just interprets a scrap transferred by the user's tainted friend. The JavaScript *Yamanner worm* was discovered on Yahoo! Mail. The *Xanga XSS worm* was found on a real world blogs. The *Gaia worm* and the *U-Dominion XSS worm* were discovered on gaming web applications. The *Justin.tv XSS worm* was detected on a websites of video hosting. The *SpaceFlash XSS worm* was also released on MySpace web application.

The Samy worm sets a record in the first 24 hours of propagation as it was spread over the Internet in a fastest way and affected more than 1 million users in just 24 hours. In July 2001, Code Red 1 utilizes the benefit of buffer-overflow vulnerability and tainted more than 359,000 workstations in 24 hours. Meanwhile in the same year, Code Red 2 (Moore, Shannon, & Brown, 2002) comes into picture to exploit the same vulnerability and infected more than 275,000 computers in 24 hours. In January 2003, Slammer replicates itself on UDP ports by taking the benefit of buffer overflow vulnerability and infected 75,000 victims in just 24 hours (Boutin, 2003). In August 2003, Blaster came into existence by exploiting the remote procedure call (RPC) against Windows workstations and infected 336,000 machines worldwide.

Bisht and Venkatakrishnan (2008) matched the parse tree of legitimate JavaScript code block with dynamic-generated JavaScript code to discover XSS attacks. Learning all legitimate JavaScript code needs the program conversion of server side scripts. Also, this technique could be evaded by certain XSS attacks. Trevor, Swamy, and Hicks (2007) produces the hashes for benign source code of JavaScript at the server side and transfer them to web browsers to get authenticated by the web browsers. This technique can be overcome by inserting the benign JavaScript method call in web pages. Shahriar and Zulkernine (2011b) designed an automated server-side XSS detection approach S^2XS^2, which is developed on the concept of "boundary injection" to encapsulate dynamic-generated content and "policy generation" to validate the data. However, this technique consumes more time in policy checks and thus degrades the attack detection capability. In another approach, Shahriar and Zulkernine (2011a) proposed a server-side JavaScript code injection technique that relies on the idea of inserting comment statements consisting of random generated tokens and characteristics of benign JavaScript code. However, it has been observed that their technique discovers only a part of code injection attacks. Some of the code injection attacks are still neglected by their approach. Furthermore because of the drawbacks of JavaScript parser, there is no automated process shown for the removal of various pre-processing techniques of JavaScript such as comment tags and return keyword in event handlers.

In addition, it has also been observed that in the past by several researchers that the initial line of guard for XSS attacks is input filtering. Wurzinger et al. (2009) proposed SWAP (secure web application proxy), a server-side solution for the discovery and mitigation of XSS attacks and vulnerabilities with the help of a mechanism of reverse proxy provided with a web browser, which is utilized for the discovery of malicious JavaScript contents. However, it has been observed that SWAP is not appropriate for high rate web services. Moreover, it is not succeeded in discovering browser-specific XSS because different browsers parse and deliver the content in a different way and as a result it becomes difficult for the web servers to discover all XSS attacks.

Johns, Engelmann, and Posegga (2008) had presented a technique, which combines static and dynamic analysis practices to detect the defective sanitization techniques that can be evaded by a malicious attacker. However this combination of static and dynamic investigation techniques was proven to be effective, but the cost of deployment of such modified framework is very high, as it requires the guarantee of the web developers of third-party web applications, which frequently have other concerns that appears to be more economically sound for them. Moreover confirming the reliability of sanitization routines, for a script-free HTML is naturally a challenging problem due to the continuous updating nature of modern web browsers.

In addition, it has also been surveyed that majority of the XSS defensive mechanism does not shield against DOM-based XSS attacks. Moreover several techniques demand major alterations at the client web browser and as well as server-side. Even though several defense mechanism incorporate the sanitization mechanisms in the code of JavaScript for shielding against the XSS attacks. However, again, majority of these techniques suffer from wide variety of false positives and false negative rates. The necessity to install updates or additional components on each user's web browser or workstation also degrades the performance of client side solutions. The XSS defensive solutions revealed above concerning mitigation of XSS attack cannot thwart the XSS attack totally. A recent statistics from White Hat Security's Website Security Statistics Report showed that nearly 80% of web applications found on the Internet have at least one serious vulnerability. Moreover, XSS vulnerability is considered to be a plague for the modern Internet-based web applications.

EXPLOITATION OF REFLECTED AND STORED XSS ATTACKS

Exploitation of Reflected XSS Attack

This section shows some of the patterns of scripts for the exploitation of reflected XSS vulnerability (Figure 5). Reflected XSS attack can be exploited on the victim's domain by simply injecting the scripting S1 (i.e., index.php). In addition to this, the attacker will craft a URL address as specified in the script S2 via email or embedded in another web page. When the victim loads the URL address as specified in the script S2 into the web browser, the victim will perceive an alert message, which displays the message *attacked*—this script does not create any harm, other than the infuriating pop-up "attacked" message.

At this time, the malicious attacker can make an effort to alter the "target URL" of the malicious link "Click to download". Instead of the malicious link directing to "xssattackexamples.com" URL address, the attacker can forward it to "not-real-xssattackexamples.com" by creating the URL address as highlighted in the script S3. In this script, the function to run on "*window.onload*" event was invoked. Since the website named *index.php* first echoes the specified name, secondly it represents the tag is echoed. The Script S4 will not execute properly, as it will execute only before the <a> tag is echoed.

Moreover, an attacker can also craft a URL address that a user cannot straightforwardly interpret. So the attacker will perform the encoding schemes to convert the ASCII characters to hexadecimal format as shown in the script S5. Now the victim cannot interpret the URL, which is crafted in hexadecimal format, and it is more likely that the attacker can visit this URL.

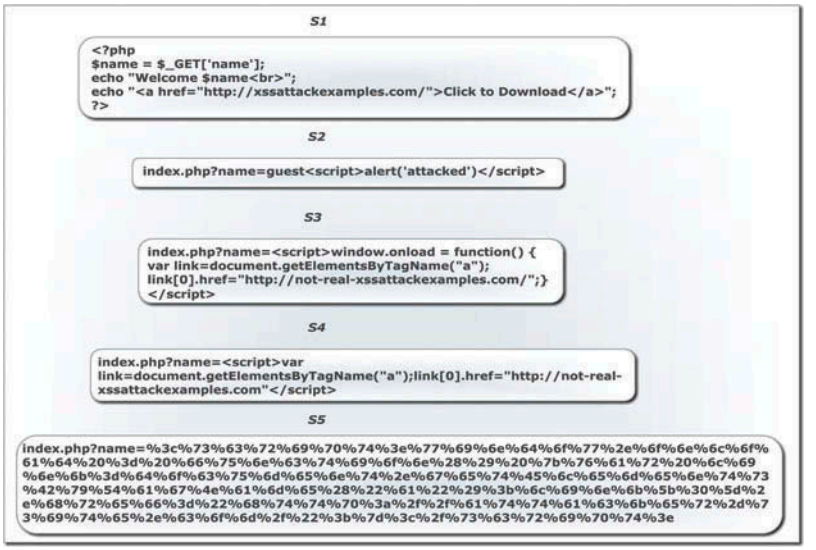

FIGURE 5 Patterns of scripts for the exploitation of reflected XSS vulnerability.

Exploitation of Stored XSS Attack

This section discusses the exploitation of stored XSS attack with the help of some real-world pattern examples of scripts. Figure 6 highlights some of the patterns of scripts for the exploitation of stored XSS vulnerability. The script S1 is a very general type that will display the pop up alert messages reflecting that "XSS" if it is susceptible. However, what happens if a website developer has an earlier idea of XSS vulnerability, but can offer very little safety against it? Here is the requirement of CharCode comes into picture, which is mainly just a straightforward structure of character encoding that has the ability to perform the encoding techniques on characters so that they achieve suitable protection but still get visible usually on the web page. The script S2 is a general one that will display the pop up alert message displaying "XSS" if it is susceptible. The script S2 is a valuable script to know, as it offers more than one type of attack at the same time. If a user receive only one or more alert message, we come to know that only one of two of them work, so we need to try to eliminate some of them to text which one is affecting the site. The CharCode for "X" is 88 and "S" is 83. As one can see, each provides a slight variation to try to beat character blocking.

The XSS attack vectors could also be concealed in a non-existent image. The code shown in the script S2 will execute malicious script concealed as an image. The subsequent script S3 is very liable to execute if a website is susceptible to XSS attack. The XSS is concealed in an image shape and CharCode is being utilized to exhibit the XSS vulnerability. In addition, the issue becomes somewhat further problematic as the text is input in Unicode and ASCII formats. Unicode is simply an essential code that was utilized to permit all characters to be accessible to everybody. Similarly, ASCII format has a same related principle. The next script S4 highlights

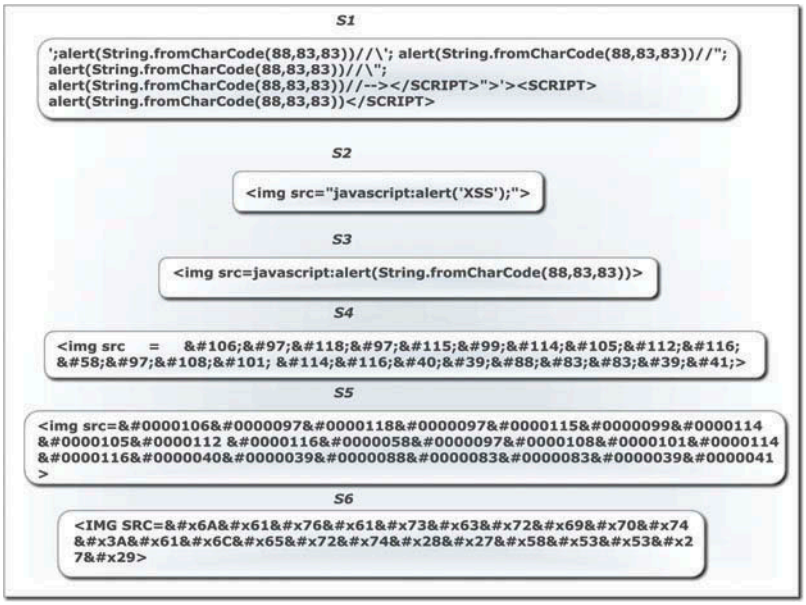

FIGURE 6 Real-world pattern example of scripts for the exploitation of stored XSS vulnerability.

the script in ASCII form. As clearly reflected, this step will thrash numerous filters since the code is unrecognizable. In contrast, interpreting the code can exhibit what it was intended to perform. The subsequent script S5 highlights the script code in Unicode representation. Again this step makes the text unrecognizable but it works in the same way. If the website has a restricted quantity of characters permissible, then this approach most likely would not be valuable. The script S6 highlights the same malicious code in hexadecimal form. Figure 7 shows some more real-world patterns of scripts for the exploitation of XSS vulnerability.

The record of probable XSS attacks is continuous. There are so many ways to evade safety checks; therefore website developers have to execute some of the preventive measures to shield their websites. Moreover web forms utilized on the majority of websites permit the users to input the code thst will reside someplace and inescapably viewed by some victim. In addition, XSS attacks can also be utilized to execute a concealed cookie stealer file, which a user will observe and permit the attacker to steal their login credentials or carry out session hijacking (Avancini & Ceccato, 2010; Gundy & Chen, 2012).

TESTING AND EXPLOITATION OF XSS ATTACKS

We have tested and exploit the vulnerabilities of XSS attack using the Open Web Application Security Project (OWASP) Web Goat (June 2, 2015) and Net Beans (June 2, 2015) on Apache Tomcat server. In this paper, certain experiments have been performed and executed for evaluating the vulnerabilities of XSS attack by using the capabilities of Net Beans and Web Goat.

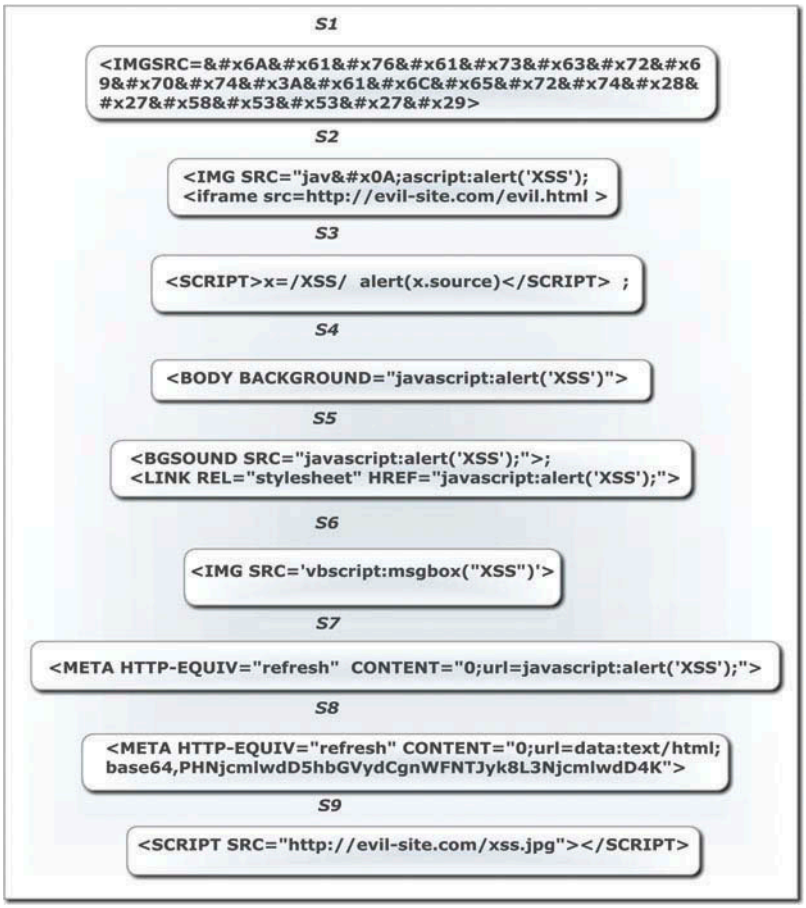

FIGURE 7 Real-world pattern examples of scripts.

Exploitation of XSS Attack Using Net Beans

Figure 8 explains the sequence of steps for the exploitation of XSS attacks using Netbeans. Here are some of the sequence of steps which illustrates the exploitation of XSS vulnerability using Net Beans.

Step 1

Step 1: To create a web form using Java and HTML, input the user ID and password, which is shown in Figure 9 in the form of a snapshot.

Step 2

Step 2: On entering username and password it redirects to a web page as shown in Figure 10 containing a malicious script.

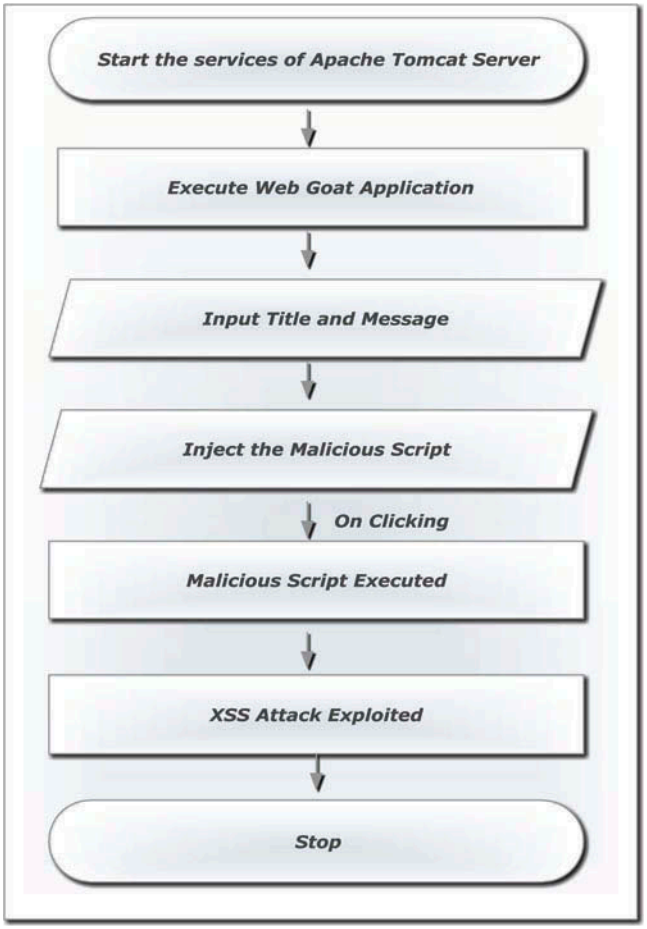

FIGURE 8 Steps for exploiting the vulnerabilities of XSS attack.

Step 3

Step 3: The link "click here" is injected with a malicious Java script, which is as shown in Figure 11.

This malicious link redirects the user to attacker's desired place "i.e. w3schools.com" according to this script, fetches the username and password from the stolen cookie, and reflects it in the URL address as shown in Figure 12.

Exploitation and Execution of XSS Attack Using Web Goat

Here these authors have performed the experiments for assessing evaluating the testing and exploitation of XSS vulnerabilities on open source OWASP Web Goat application. Following

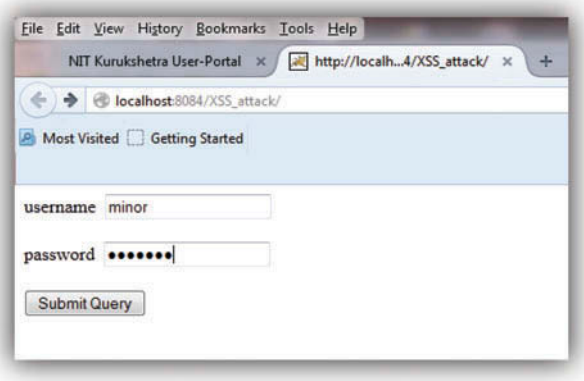

FIGURE 9 Web application form supplying the user-ID and password.

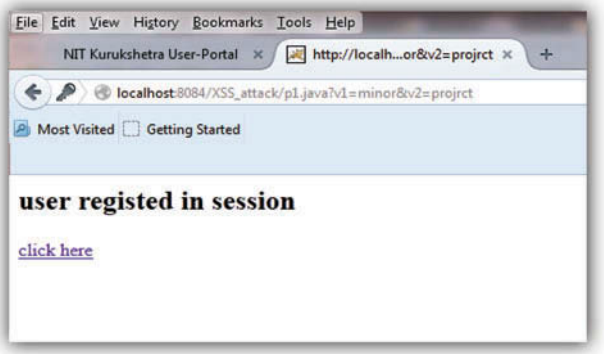

FIGURE 10 Injected malicious script.

```
<script>
document.location=\"http://w3schools.com?c=\"+(document.cookie);
</script>
```

FIGURE 11 Injected malicious link.

are some of the sequences that explains step-by-step exploitation of XSS attack on Web Goat web application:

Step 1

Step 1: Start the services of Apache Tomcat server and then run the home web page of web goat web application as shown in Figure 13.

FIGURE 12 URL reflecting the user ID and Password.

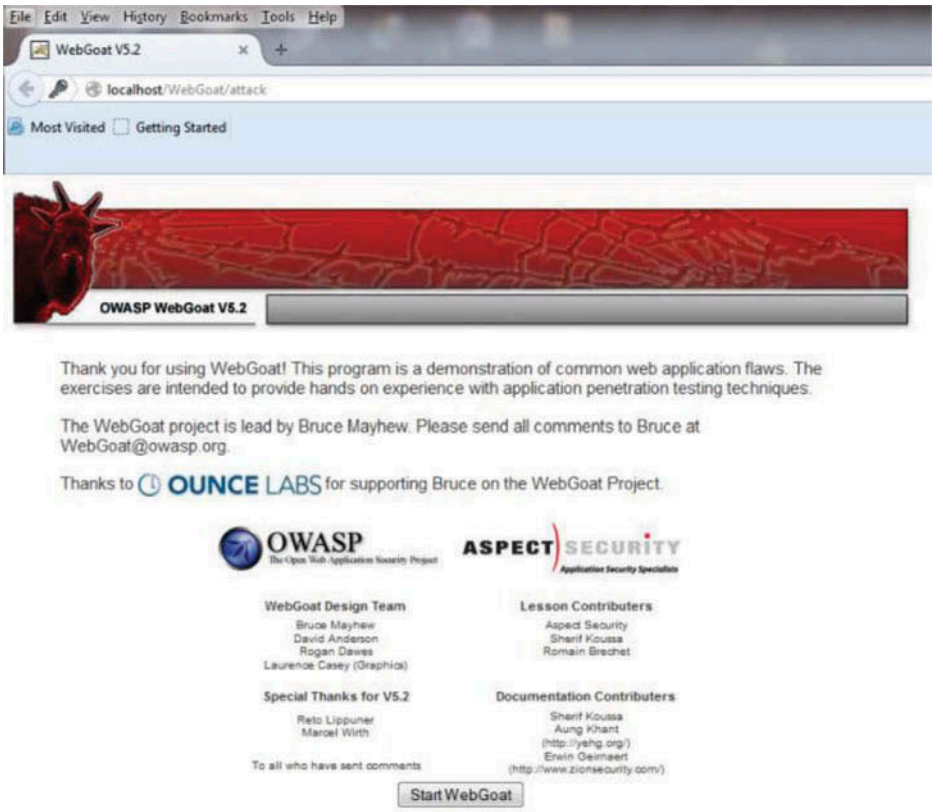

FIGURE 13 Web goat home web page.

Step 2

Step 2: Click on *Start WebGoat* button and choose cross-site scripting option. Then click on stored XSS and enter the message along with the malicious script, as shown in Figure 14.

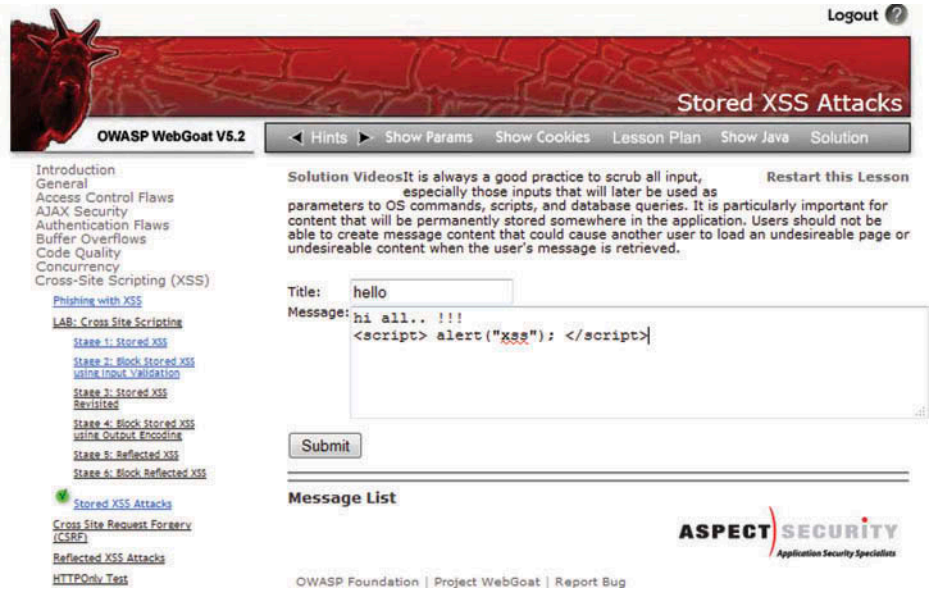

FIGURE 14 Web page embedded with the malicious script.

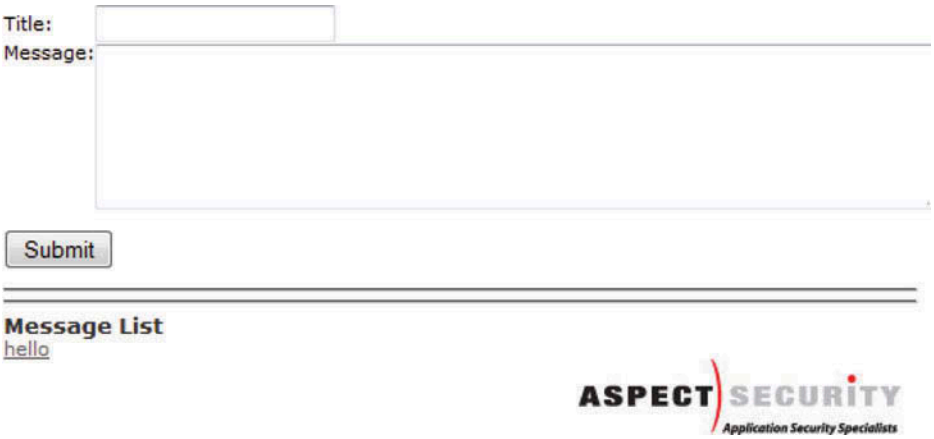

FIGURE 15 Reflected malicious script in the message list.

Step 3

Step 3: Click on the submit button and the injected message will reflect on the website as shown in Figure 15.

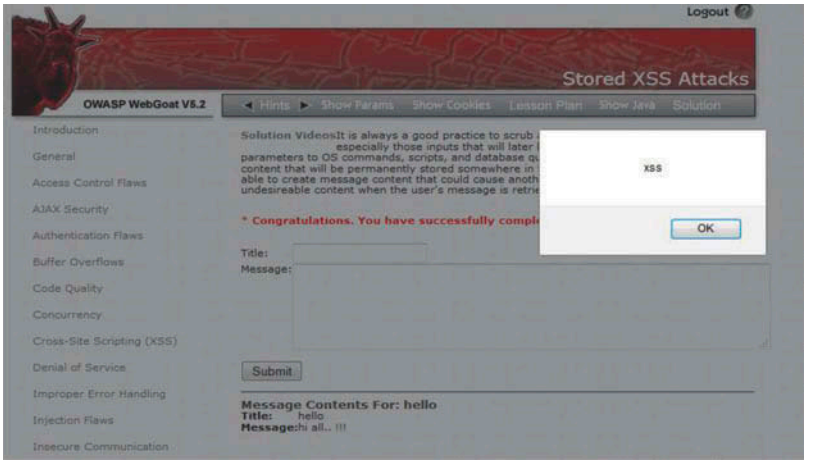

FIGURE 16 Malicious script is executed and the XSS attack exploited.

Step 4

Step 4: When any other user will try to see this message, the malicious injected script gets executed and finally XSS attack gets exploit on the web application goat, which is as shown in the Figure 16.

PREVENTION OF XSS ATTACKS OVER NET BEANS AND WEB GOAT

However, previous researchers proposed various defensive solutions for mitigating the XSS vulnerabilities from modern web applications. This study, however, has prevented the execution of malicious scripts on the victim's web browser by replacing the injected scripts with the encoded scripts. This encoded version of script can be generated from Rsnake Cheat Sheet Calculator (Hansen, 2015). The following Table 1 highlights some of the encoded versions of the scripts.

TABLE 1
Encoded Version of Scripts

Normal Script	*Encoded Script*
xxs link	xxs link
<script>alert(document cookie);</script>	<script> alert(document.cookie); </sript>
<IERAME SRC="javascript:alert ('XSS');"></IERAME>	<IRAME SRC="javascript:alert('XSS')& #x3C;/IRAME>

This, however, is not the final defensive mechanism to bypass the XSS attacks. The encoded version of the scripts can be evaded by following some different paths in the source code of web applications. In addition, several server-side countermeasures do exist, but such techniques have not been universally applied because of their exploitation overhead. Another point is that existing client side solutions degrade the performance of client's system resulting in poor web surfing experience. The requisite to install updates or additional components on each user's web browser or workstation also degrade the performance of client side solutions.

CONCLUSION AND SCOPE FOR FUTURE RESEARCH

In cyber physical systems, physical world processes are merged with global virtual networks to provide fast Internet. Cross-site scripting vulnerabilities are persistent in web applications. Mischievous users frequently exploit these vulnerabilities to steal personal information. One can prevent a malicious script from being executed by encoding it and then it can be easily checked whether a website is vulnerable to XSS or not. However, this is not the final defense mechanism for mitigation the XSS vulnerability from modern web applications.

In this article, the authors have tested the vulnerabilities of XSS attack on the local host server Apache Tomcat by utilizing the malicious scripts from XSS cheat sheet website. In addition, this research has also exploited the same vulnerabilities on Web Goat. Lastly this study has also exploited the encoded versions of the injected scripts for testing the level of XSS attack prevention capability. Based on the observed results, we have discussed some further work.

It can also be concluded that website developers should revise the exploitation of supportive tools to assure an organization without chance of vulnerabilities, although the attacker can go on dealing with novel policies to take advantage of vulnerable web applications. The consequences of such vulnerabilities can be perceived by the existence of the online web applications in main stern schemes in organizations, such as health care institutions, e-commerce organizations, and banking. This article has explained how the survival of XSS vulnerabilities on a website can include a High level of risk for the users of web application. There are currently very stimulating defensive solutions that offer appealing techniques to resolve the XSS problem. But again, these solutions suffer from several loopholes as previously discussed. Some do not offer adequate security measures and can be simply evaded, while others are so obscure that they would be difficult to implement in real-world scenarios.

Future Scope

The most essential technique of mitigating XSS attacks is to carry out safe input handling. Encoding mechanisms should be exploited each time user input is incorporated in a web page. In several other cases, encoding mechanisms have to be substituted with input validation mechanisms. Safe handling of user-supplied input has to take into consideration which perspective of a web page the user-supplied input is injected into. Therefore, to thwart all types of XSS attacks, safe handling of user-supplied input has to be exploited in both client-side and server-side source code of web applications. Taking into consideration all the research gaps discovered in the existing defensive techniques of XSS attacks, a novel XSS defensive technique is needed to fulfill all the following requirements:

- Automated process are needed to distinguish between legitimate JavaScript code from a malicious injected code.
- Efficient and fast policy checks are needed to optimize the speed of XSS attack detection capability.
- Placement of context-sensitive sanitization routines is required to be introduced in the source code of web applications.
- XSS defensive solution should be capable enough to discover all browser-specific XSS attacks.
- The method must possess the updated white-list and blacklist of scripts to avoid the rate of false positives and false negatives.
- Powerful web crawler components must be utilized to execute the deep scanning of web pages to discover possible target links and user input forms.

REFERENCES

Alcorna, W. (2006). Cross-site scripting viruses and worms—A new attack vector. *Journal of Network Security*, *7*, 7–8.

Arulsuju, D. (2011). Hunting malicious attacks in social networks. In *Advanced Computing's Third International Conference* (pp. 13–17), IEEE, December 14–16, Chennai, India.

Athanasopoulos, E., Krithinakis, A., & Markatos, E. P. (2010). Hunting cross-site scripting attacks in the network. In *W2SP 2010: Web 2.0 Security and Privacy Workshop* (pp. 89–92), May 20, Oakland, California, USA.

Avancini, A., & Ceccato, M. (2010). Towards security testing with taint analysis and genetic algorithms. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems* (Section 5, pp. 65–71), ACM, May 2, Capetown, South Africa.

Barhoom, T. S., & Kohail, S. N. (2011). A new server-side solution for detecting cross-site scripting attack. *International Journal of Computer Informatics System*, *3*(2), 19–23.

Bisht, P., & Venkatakrishnan, V. N. (2008). XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment* (pp. 23–43), Springer-Verlag, Berlin, Heidelberg, 10–11 July, Paris, France.

Boutin, P. (2003, July). Slammed! An inside view of the worm that crashed the Internet in 15 minutes. *Wired, 11*(7)1–2. Retrieved from http://www.wired.com/wired/archive/11.07/slammer.html

Cao, Y., Yegneswaran, V., Possas, P., & Chen, Y. (2012). Pathcutter: Severing the self-propagation path of XSS JavaScript worms in social web networks. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, Internet Society, 5–8 February, San Diego, CA.

CVE (Common Vulnerabilities and Exposures). MITRE Corporation. 3 July 2007. Retrieved June 2, 2015 from https://cve.mitre.org/.

Faghani, M. R., & Saidi, H. (2009). Social networks' XSS worms. In *International Conference on Computational Science and Engineering* (pp. 1137–1141), IEEE, 29–31 August, Vancouver, BC, Canada.

Flanagan, D. (2001, November). *JavaScript: The definitive guide* (4th ed.). Sebastopol, CA: O'Reilly Media, Inc.

Fogie, S., Grossman, J., Hansen, A. Rager, & Petkov, P. D. (2006). *XSS attacks: Cross-site scripting exploits and defense*. Burlington, MA: Syngress Publishing, Inc./Elsevier.

Forrest, S., Hofmeyr, S. A., Somayaji, A., & Longstaff, T. A. (1996). A sense of self for UNIX processes. In *IEEE Symposium on Security and Privacy* (pp. 120–129), 6–8 May, Oakland, CA, USA.

Grossman, J., Hansen, R., & Fogie, S. (2007). *Cross-site scripting attacks*. Amsterdam, The Netherlands: Syngress/Elsevier.

Gupta, M., & Gupta, B. B. (2014). *BDS: Browser dependent XSS sanitizer. Book on cloud-based databases with biometric applications*. Hershey, PA: IGI–Global. Advances in Information Security, Privacy, and Ethics (AISPE) Series.

Gupta, S. & Gupta, B. B. (2015). PHP-sensor: A prototype method to discover workflow violation and XSS vulnerabilities in PHP web application. In *12th ACM International Conference on Computing Frontiers (CF '15)*, 18–21 May, Ischia, Italy.

Gundy, M. V., & Chen, H. (2012). Noncespaces: Using randomization to defeat cross-site scripting attacks. *Computers & Security*, *31*(4), 612–628.

Gupta, S., & Sharma, L. (2012). Exploitation of cross-site scripting (XSS) vulnerability on real world web applications and its defense. *International Journal of Computer Applications (IJCA)*, *60*(14), 28–33.

Gupta, S., Sharma, L., Gupta, M., & Gupta, S. (2012, September). Prevention of cross-site scripting vulnerabilities using dynamic hash generation technique on the server side. *International Journal of Advanced Computer Research (IJACR)*, *2*(5), 49–54.

Hallaraker, G., & Vigna. (2005). Detecting malicious JavaScript code in Mozilla. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)* (pp. 85–94), 16–20 June, Shanghai, China.

Hansen, R. *XSS cheat sheet for filter evasion*. Retrieved from https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Howard, M., & LeBlanc, D. E. (2005). *Writing secure code* (2nd ed.). Redmond, WA: Microsoft Press.

Johns, M., Engelmann, B., & Posegga, J. (2008). XSSDS: Server-side detection of cross-site scripting attacks. In *Proceedings of the ACSAC, California* (pp. 335–344), IEEE, 8–12 December, Anaheim, CA, USA.

Livshits, B., & Erlingsson. (2007). Using web application construction frameworks to protect against code injection attacks. In *Workshop on Programming Languages and Analysis for Security* (pp. 95–104), ACM, June 14, San Diego, CA, USA.

Lucca, G. A. D., Fasolino, A. R., Mastoianni, M., & Tramontana, P. (2004). Identifying cross-site scripting vulnerabilities in web applications. In *Proceedings of the 6th IEEE International Workshop on Web Site Evolution* (pp. 71–80). Washington, DC: IEEE.

MacDonald, M., & Szpuszta, M. (2005). *Pro ASP.NET 2.0 in C# 2005*. New York, NY: Apress.

Mook, N. (2005, October 13). Cross-Site scripting worm hits MySpace. *BetaNews*. Retrieved from http://www.betanews.com/article/CrossSite_Scripting_Worm_Hits_MySpace/1129232391

Moore, D., Shannon, C., & Brown, J. (2002). Code-Red: A case study on the spread and victims of an Internet worm. In *Internet measurement workshop (IMW) 2002—ACM SIGCOMM/USENIX* (pp. 273–284), November 2002, San Diego, CA. San Diego, CA: Center for Applied Internet Data Analysis (CAIDA). Retrieved from http://www.caida.org/outreach/papers/(2002)./codered/codered.pdf

NetBeans. *NetBeans IDE: The smarter and faster way to code*. NetBeans homepage. Retrieved June 2, 2015 from https://netbeans.org/

Open Web Application Security Project (OWASP). *OWASP Web Goat project*. Bel Air, MD: OWASP. Retrieved June 2, 2015 from https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

Shahriar, H., & Zulkernine, M. (2011a). Injecting comments to detect JavaScript code injection attacks. In *Proceedings of the 6th IEEE Workshop on Security, Trust, and Privacy for Software Applications* (pp. 104–109), 18–22 July, Munich, Germany.

Shaihriar, H., & Zulkernine, M. (2011b). S2XS2: A server side approach to automatically detect XSS attacks. In *Ninth International Conference on Dependable, Automatic Secure Computing* (pp. 7–17), IEEE CS Press, 12–14th December, Sydney, Australia.

Samy. (2005, October 4). *Technical explanation of the MySpace worm*. Retrieved from http://namb.la/popular/tech.html

Samy's cancelled MySpace profile. Retrieved from http://www.myspace.com/33934660

Trevor, J., Swamy, N., & Hicks, M. (2007). Defeating script injection attacks with browser–enforced embedded policies. In *WWW'07: Proceedings of the 16th International Conference on World Wide Web* (pp. 601–610), May, 8–12, Banff, AB, Canada.

Wurzinger, P., Platzer, C., Ludl, C., Kirda, E., & Kruegel, C. (2009). SWAP: Mitigating XSS attacks using a reverse proxy. In *ICSE Workshop on Software Engineering for Secure Systems* (pp. 33–39), IEEE Computer Society, May 14–16, Vancouver, Canada.