# Mitigating Cross-Site Scripting Attacks with a Content Security Policy

B. SAI JYOTHI[1], RAVIPATI JYOTHI[2], YELLANKI BHAVANI[3], SHAIK MABIBI[4], SANAGAVARAPU PRIYANKA[5]
[1, 2,3,4,5] *Computer Science of Engineering, Vasireddy Venkatadri Institute of Technology, Nambur, Guntur*

***Abstract-*** ***A content security policy (CSP) can help Web application developers and server administrator's better control website content and avoid vulnerabilities to cross-site scripting (XSS). In experiments with a prototype website, the authors' CSP implementation successfully mitigated all XSS attack types in four popular browsers. An XSS attack involves injecting malicious script into a trusted website that executes on a visitor's browser without the visitor's knowledge and thereby enables the attacker to access sensitive user data, such as session tokens and cookies stored on the browser.1 With this data, attackers can execute several malicious acts, including identity theft, key logging, phishing, user impersonation, and webcam activation.***

***Index-Terms: Content Security Policy, Cross Site Scripting, Web Applications, Input Sanitizers, Mitigating, Vulnerabilities.***

## I. INTRODUCTION

A primary goal of CSP is to mitigate and report XSS attacks. XSS attacks exploit the browser's trust of the content received from the server. Malicious scripts are executed by the victim's browser because the browser trusts the source of the content, even when it's not coming from where it seems to be coming from. CSP makes it possible for server administrators to reduce or eliminate the vectors by which XSS can occur by specifying the domains that the browser should consider to be valid sources of executable scripts. A CSP compatible browser will then only execute scripts loaded in source files received from those whitelisted domains, ignoring

all other script (including inline scripts and event-handling HTML attributes). Researchers have proposed a range of mechanisms to prevent XSS attacks, with content sanitizers dominating those approaches. Although sanitizing eliminates potentially harmful content from untrusted input, each Web application must manually implement it—a process prone to error. To avoid this problem, we use a different technique. Instead of sanitizing harmful scripts before they are injected into a website, we block them from loading and executing with a variation of the content security policy (CSP), which provides server administrators with a white list of accepted and approved resources. The Web application or website will block any input not on that list and thus there is no need for sanitizing. The white list also guards against data exfiltration and extrusion—the unauthorized downloading of data from a website visitor's computer.

### 1.1 Objective

A CSP closes this XSS loophole through its white list, which guides the browser to execute only the listed resources. Thus, even if the attacker finds a way to inject a script into the trusted origin, it would not match the resources and content in the white list and would therefore be rejected. Web application developers or server administrators use the default source, or default-src, directive to de ne the white list of resources.

## II. TYPES OF XSS ATTACKS

An XSS attack can be Persistent, Non-persistent or it can be based on a Document Object Model (DOM).

1) Persistent XSS:

Persistent XSS attack also known as a stored XSS or Type-1 XSS attack. It is usually difficult to detect and is more harmful than the other two attack types. Because the malicious script is rendered automatically, there is no need to target individual victims or lure them to a third party website. This type of attack involves injecting malicious script into a trusted website, which stores the script in its

database. If the stored script is malicious and not filtered then it is included as a part of Web application and run within the browser's site. A persistent XSS attack does not need a malicious link for successful exploitation, by simply visiting that web page may compromise the user. In the persistent XSS attack, the malicious input originates from the victims request.

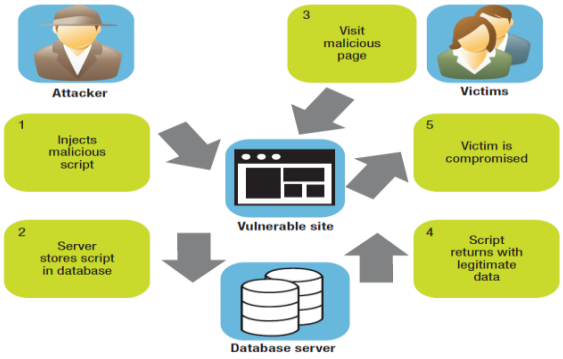



Figure I: - Typical scenario of a persistent cross-site scripting (XSS) attack.

Each time a user visits a webpage injected with malicious script, the stored script exploits the user's browser privileges to access sensitive information.

2) Non-Persistent XSS:

A Non-persistent, or reflected, XSS attack, which occurs when a website or Web application passes invalid user inputs. Usually, an attacker hides malicious script in the URL, disguising it as user input, and lures victims by sending emails that prompt users to click on the crafted URL.

When they do, the harmful script executes in the browser, allowing the attacker to steal authenticated cookies or data. In the figure, we assume that victims have authenticated themselves at the vulnerable site.

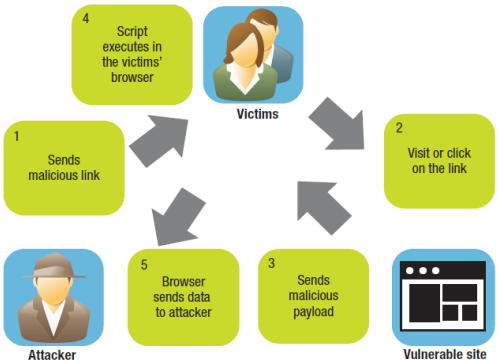


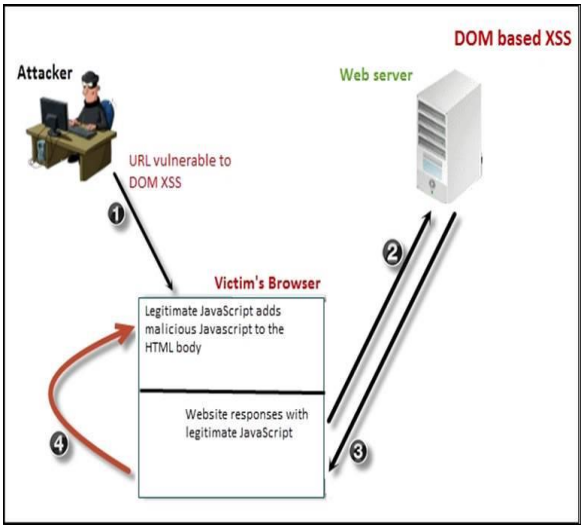

Figure 2. Typical scenario of a no persistent XSS attack.

Victims authenticate themselves at the site and the attacker lures them into loading a malicious link. The link then executes malicious code with the user's credentials.

3) DOM-based XSS:

A webpage is composed of various elements, such as forms, paragraphs, and tables, which are represented in an object hierarchy. To update the structure and style of webpage content dynamically, all Web applications and websites interact with the DOM, a virtual map that enables access to these webpage elements.

The attack occurs when the victim's browser executes the malicious code from the modified DOM. On the client side, the HTTP response does not change but the script executes maliciously. This exploit works only if the browser does not modify the URL characters. A DOM-based XSS attack is the most advanced type and is not well known. Indeed, much of the vulnerability to this attack type stems from the inability of Web application developers to fully understand how it works.




## III. LITERATURE REVIEW

"Defending Against Cross-Site Scripting Attacks", L.K. Shar and H.B.K. Tan, Computer, vol. 45, no. 3, 2012, pp. 55−62. Researchers Have Proposed Multiple Solutions To Cross-site Scripting, But Vulnerabilities Continue To Exist In Many Web Applications Due To

Developers' Lack Of Understanding Of The Problem And Their Unfamiliarity With Current Defenses' Strengths And Limitations.

NOXES: "ACLIENT-SIDE SOLUTION FOR MITIGATING CROSS-SITE SCRIPTING ATTACKS,"E. Kirda et al., Proc. 21st Ann.ACM Symp. Applied Computing (SAC06), 2006, pp. 330−337.

Web applications are becoming the dominant way to provide access to on-line services. At the same time, web application vulnerabilities are being discovered and disclosed at an alarming rate. This paper presents Noxes, which is, to the best of our knowledge, the first client-side solution to mitigate cross-site scripting attacks. Noxes Acts as a Web Proxy and Uses Both Manual and Automatically Generated Rules to Mitigate Possible Cross-site Scripting Attempts. Noxes Effectively Protects Against Information Leakage From The User's Environment While Requiring Minimal User Interaction And Customization Effort.

"Defeating Script Injection Attacks With Browser-Enforced Embedded Policies",T. Jim, N. Swamy, And M. Hicks, Proc. 16th Int'l Acm Conf. World Wide Web (Www07), 2007, Pp. 601−610.

This paper proposes a simple alternative mechanism for preventing script injection called Browser-Enforced Embedded Policies (BEEP). The idea is that a web site can embed a policy in its pages that specifies which scripts are allowed to run. The browser, which knows exactly when it will run a script, can enforce this policy perfectly. We have added BEEP support to several browsers, and built tools to simplify adding policies to web applications.

"DOCUMENT STRUCTURE INTEGRITY: A ROBUST BASIS FOR CROSS-SITE SCRIPTING DEFENSE"

Y. Nadji, P. Saxena, and D. Song, "Document Structure Integrity: A Robust Basis for Cross-Site Scripting Defense,"Proc. 6th Ann. Network & Distributed System Security Symp. (NDSS09), 2009;www.cs.berkeley.edu/~dawnsong /papers/2009%20dsi ndss09.pdf.

Cross-site scripting (or XSS) has been the most dominant class of web vulnerabilities in 2007. In this paper, we develop a new approach that combines randomization of web application code and runtime tracking of untrusted data both on the server and the browser to combat XSS attacks. We call this property DOCUMENT STRUCTURE INTEGRITY (or DSI). Similar to prepared statements in SQL, DSI enforcement ensures automatic syntactic isolation of inline user generated data at the parser-level.

3.1 Existing System

Identity request will be send by the system. The collected information will be send to the collected database server. The server not only instructs the clients about the XSS attacks but also informs the vulnerable sites for preventing. So this mechanism requires minimal effort and low performance. The pattern filtering and code filtering approaches are only to prevent the persistent attack and those rules don't work for non-persistent XSS attack. If the filtering approach fails to work, then the malicious script will be stored and executed in the database.

3.1.1 Disadvantages

- Approved scripts have to be identified by the website.
- It provides low performance.
- How to use the collected information in database is not addressed.
- There are multiple policies for the documents. No single policy for all the documents.
- It requires user-defined security policies which can be labor-intensive.
- How to make system deployed universally has also not been mentioned.

3.2 Proposed System

Researchers have proposed a range of mechanisms to prevent XSS attacks, with content sanitizers dominating those approaches. Although sanitizing eliminates potentially harmful content from untrusted input, each Web application must manually implement it—a process prone to error. To avoid this problem, we use a different technique.

Instead of sanitizing harmful scripts before they are injected into a website, we block them from loading

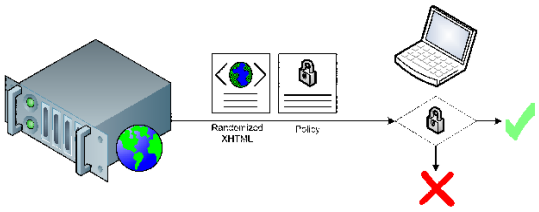
and executing with a variation of the content security policy (CSP), which provides server administrators with a white list of accepted and approved resources. The Web application or website will block any input not on that list and thus there is no need for sanitizing.

3.2.1 Advantages of Proposed System

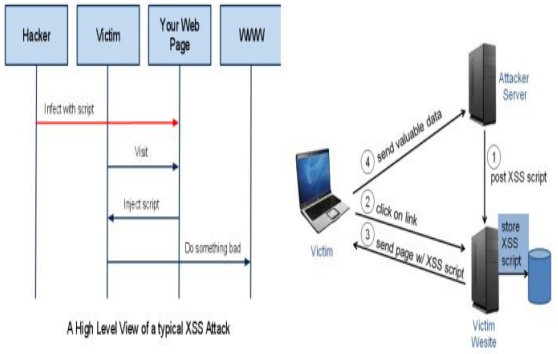
- It improves accuracy.
- Increases Computational Efficiency.
- Scalability and Reliability.
- The Proposed approach is modeled in such a way that it validates the input at client side. This technique works for both Persistent and Non-Persistent attacks. The server side approach provides validate output.
- Web applications provide security critical services for preventing web related vulnerabilities.
- Automatic rewriting of .NET applications better support CSP.

## IV. SYSTEM MODEL

Cross-site scripting is a type of computer security vulnerability found in web-based applications which allows code injection by malicious web users into any web page that is viewed by the other users. The term "Cross-site scripting", originated when a malicious website could potentially load a website onto other window and then use JavaScript to read or write information on the other website, which was later redefined as injection.




At the time of attack, everything seems to be fine to the end users, but they are subjected to a wide variety of threats. This XSS attack is potentially a dangerous vulnerability that is easy to execute and arduous to repair. The above figure shows that the vulnerable site sends the documents requested by the user only if it is secured by the white listed policy. If the requested content from the web application satisfies the Content Security Policy, then it is send to the user. Otherwise the request is rejected and secure messages are sending as the requested website is malicious.




To avoid the web application attacks the web browser security model is built on the same origin policy that isolates one origin from the other thus providing the developers a safe sandbox environment to build these applications in which the code from one origin (http://self.com) has access to only https://self.com data and the code from other origin (https://other.com) is not permitted to access https://self.com data. But the attackers by pass this policy by exploiting cross-site scripting vulnerabilities in the web application. He injects his own script into the web application and later this injected script will get embedded along with the actual intended response from the website whenever any user visits that particular webpage.

Working of CSP

If a browser is embedded with CSP, it simply follows the CSP's directives-language constructs that specify how a compiler should process its input. CSP blocks the execution of inline JavaScript. CSP allows developers or administrators to explicitly define, using a declarative policy language, the origin from which different classes of content can be included into a document. Policies are sent by the server in a special security header, and a browser supporting the standard is then responsible for enforcing the policy on the client. CSP provides a principled and robust mechanism for preventing the inclusion of malicious content in security-sensitive web applications.

The types of directives supported in the current W3C standard CSP 1.0.

| Directive | Content Sources |
|---|---|
| default-src | All types, if not otherwise explicitly specified |
| script-src | JavaScript, XSLT |
| object-src | Plugins, such as Flash players |
| style-src | Styles, such as CSS |
| img-src | Images |
| media-src | Video and audio (HTML5) |
| frame-src | Pages displayed inside frames |
| font-src | Font files |
| connect-src | Targets of XMLHttpRequest, WebSockets |

Source directives

CSP source directives control how a client-side browser should behave when it comes across various types of protected website content-from JavaScript to connection locations. Of these source directives the most common are default, script and style.

Default source:

Web application developers or server administrators use the default source, or default-src, directive to define the white list of resources. Sample policies using this directive are

Content-Security-Policy: default src 'self'

Which permits client browsers to load all resources only from the Web application's own origin (protocol, hostname, and port number), and

Content-Security-Policy: default src 'none'

Which specifies with the keyword none that no resource is allowed to load?

Script source:

The script-src directive controls the loading of JavaScript on the website. The first part of the sample policy

Content-Security-Policy: default src 'none'; script-src script .example.com javascript.example.com

Specifies a default-src of 'none'. The second part permits the client browser to load script from script.example.com and javascript.example.com. The second part overwrites the default-src policy— that is, no resource (script) is permitted to load except from script.example.com and javascript.example.com.

Style source:

The style-src directive controls the use of Cascading Style Sheets (CSS) and other styles on a webpage. The policy

Content-Security-Policy: default src 'none'; style-src 'unsafe inline' maxcdn.bootstrapcdn.com

Allows the use of inline style and the style sheets from bootstrapcdn.com only. It disallows the loading of any other sources, such as the connect, frame, and media sources.

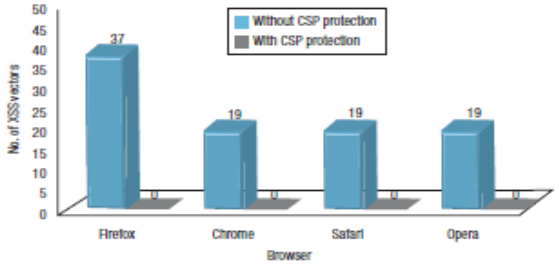
## V. RESEARCH FEATURE

**TABLE 1.** Browser versions that support content security policies (CSPs).

| Header | Firefox | Chrome | Safari | Internet Explorer | IOS Safari | Android | Chrome Android |
|---|---|---|---|---|---|---|---|
| Content-Security-Policy | 23+ | 25+ | 7+ | – | 7.1+ | 4.4+ | 47 |
| X-Content-Security-Policy | 4.0+ | – | – | 10+ (limited)* | – | – | – |
| X-WebKit-CSP | – | 14+ | 6+ | – | 5.1+ (limited)* | – | – |

*Supports only partial CSP

Based on Comparison chart test results are evolved that is, Without CSP protection, as many as 37 XSS vectors were successful (Firefox). Even the XSS auditor in Chrome, Safari, and Opera could not eliminate all XSS vectors. However, applying CSP protection eliminated all XSS vectors for each browser. This result is shown in following figure.




## VI. RELATE WORK

CSP was proposed by Stamm et al, who provided the first implementation in the Firefox browser. Subsequently, CSP became a W3C standard and was adopted by most major browsers. CSP was the first widely deployed browser policy framework to mitigate content injection attacks. However, it was not the first one to be suggested. SOMA (Same Origin Mutual Approval) reduces the impact of XSS by controlling information flows. Website operators need to approve content sources in a manifest file, as well

as content providers need to approve websites to include their content. BEEP (Browser Enforced Embedded Policies) can prevent XSS attacks with a whitelist approach for JavaScript and a DOM (Document Object Model) sandbox for possibly malicious user content.

## VII. CONCLUSION AND FUTURE SCOPE

Although our CSP has many benefits, it is not intended as a primary defence mechanism against XSS attacks. Document structure integrity (dsi) is a client-server architecture that restricts the interpretation of untrusted content.7 DSI uses parser-level isolation to isolate inline untrusted data and separates dynamic content from static content. However, this approach requires both servers and clients to cooperatively upgrade to enable protection.

Lot of work has been done to handle XSS attacks which include:

- Client side approaches
- Server side approaches
- Testing based approaches
- Static and dynamic analysis based approaches

## VIII. FUTURE SCOPE

In this work, it has been restricted the XSS attacks with the help of content filtering algorithm. This algorithm works fine because it allows no script to store in the database and thus no script is made to be executed. But, it made the efforts to reduce the XSS attacks by means of cookie stealing which is not only the way of performing XSS attacks. In future, the same algorithm will be implemented to restrict attacks done through key logging etc. The scope may be extended to implement CSP to execute the inline JavaScript.

## REFERENCES

[1] M. Johns, "Code Injection Vulnerabilities in Web Applications—Exemplified at Cross-Site Scripting," PhD dissertation, Univ. of Passau, 2009; https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index /docId/144.

[2] Open Web Application Security Project, "OWASP Top 10 – 2013: The Ten Most Critical Web Application Security Risks," 2013;www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013.

[3] I. Yusof and A.-S.K. Pathan, "Preventing Persistent Cross-Site Scripting (XSS) Attack by Applying Pattern Filtering Approach," Proc. 5th IEEE Conf. Information and Communication Technology for the Muslim World (ICT4M14), 2014, pp. 1−6.

[4] L.K. Shar and H.B.K. Tan, "Defending against Cross-Site Scripting Attacks," Computer, vol. 45, no. 3, 2012, pp. 55−62.

[5] E. Kirda et al., "Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks," Proc. 21st Ann.ACM Symp. Applied Computing (SAC06), 2006, pp. 330−337.

[6] T. Jim, N. Swamy, and M. Hicks, "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies," Proc. 16th Int'l ACM Conf. Worldwide Web (WWW07), 2007, pp. 601−610.