

University of Toronto
Faculty of Engineering

ECE1776F: Project Proposal:
A Client-Side Browser-Integrated Solution for
Detecting and Preventing Cross Site Scripting (XSS) Attacks

Gordon Chiu
Bernice Chan

September 25, 2006

1.0 Introduction

Web applications have become the dominant method for implementing and providing access to on-line services. As the use of web applications increases, a greater focus on web security is needed. In recent years, there has been a significant increase in the number of web-based attacks [1]. One key technology used in interactive web applications is JavaScript [2]. Embedded into the HTML of a web page, it is dynamically executed at the client, allowing for enhanced webpage display and greater interactivity. However, the automatic execution of JavaScript code provided by the remote server may represent a possible vector for attack on the end-user's computing environment.

1.1 Motivation

There exists a potential vulnerability that can permit malicious code to access resources of another domain ("the vulnerable domain"). If an attacker were able to inject the code such that it appears to come from the vulnerable domain, it would run with full privileges and access to state information of the vulnerable domain. Such an attack is known as a cross-site scripting (XSS) Attack. [3]

Recently, MITRE reported that cross-site scripting attacks accounted for 21.5% of all common vulnerability and exposures (CVEs) reported in 2006. Two main classes of cross-scripting attacks exist. The first is non-persistent or reflected attack, and is by far the most common. In a reflected attack, a URL containing the malicious code, either as a HTTP GET query parameter or part of the locator string itself, is constructed by the attacker. The vulnerable page or server "reflects" the malicious code back to the user (for example, the vulnerable page echoes, without checking, one of the input variables to the page). The second type of attack is a persistent or stored attack. In a stored attack, the malicious code is stored server-side (in a database, or file-system) and displayed to the user. One example of this is online message boards, where an attacker could post messages containing the malicious code. A related vulnerability is HTTP response splitting whereby an attacker can insert code into an HTTP response or form an additional controlled HTTP response by inserting extra CR and LF characters into URL or body parameters. This vulnerability enables the attacker to mount various kinds of XSS and web caching attacks.

2.0 Proposed Solution

Our proposed work aims to create a client-side in-browser solution for mitigating cross-site scripting, which addresses concerns and shortcomings present in previous work. The solution will be implemented as a plugin for the popular web browser FireFox; this plugin will automatically detect possible instances of a cross-site scripting attack and prevent execution of malicious JavaScript code. We also propose several additional features to help combat phishing, IDN domain name spoofing, and identify possibly suspicious links. The following are ideas for how to implement detection and prevention cross-site scripting attacks at the browser level. This list is subject to change as our proposed work may reveal further avenues for exploration.

- Detection and prevention of all XSS local (Type 0) and Reflected Attacks (Type 1). In both types of attacks, the malicious JavaScript code is included as part of the URL (parameter passing). This JavaScript code is then embedded in the resulting HTML document by the client side scripts (Type 0) or by the vulnerable server (Type 1). It should be possible to perform a comparison of the executable portions of JavaScript code embedded in the page with the original requested URL. If portions of the requested URL are present in executable JavaScript code, this is an indication that cross-site scripting could occur, and execution of the JavaScript code should be prevented.

Several considerations need to be made. First, only executable JavaScript code should be considered. String constants or other "data" should be exempted from the URL-matching check. Second, the string matching should be robust enough to account for hexadecimal encoding of parameters, and other character encoding schemes. Also, the string matching should be done in an "approximate matching" fashion to circumvent advanced attack techniques which fragment the malicious code payload in multiple parameters. Finally, the browser is a natural place to perform the comparison, after JavaScript has been parsed but prior to execution: this reduces the effectiveness of attacks that use comments or other irrelevant markup (such as extra whitespace, redundant or irrelevant parameters, uncommon character encodings) to disguise the malicious code.

- Detection and prevention of HTTP response splitting. The attacker modifies the HTTP response header and body by inserting extra CR and LF characters into a URL or body parameters. In doing so,

the attacker is able to insert code into the HTTP response or additionally split the response into two and have complete control of the second response. To guard against possible URL and body parameter injections, all CR and LF characters must be disallowed in HTTP response headers.

Possible Additional Features:

- **Anti-Phishing Feature.** This feature will leverage Google's reputation system. We will rely directly on Google's PageRank system to determine the relevancy and legitimacy of a site and display this information in the toolbar. We may additionally identify domain names using international character sets by color coding different character sets to combat IDN domain spoofing. These two features will help to warn a user that the site is potentially a phishing website.
- **Suspicious Link Feature.** This feature will attempt to identify links where the displayed text of a hyperlink (that is formatted as a URL string), the linked location, and/or text displayed during a mouseover event differ. The plugin will not prevent the user from accessing the link, but will simply add a warning indicator flag next to the link.

2.1 Related Work

The problem of detecting and mitigating cross-site scripting attacks has been previously studied in literature. Many different solutions have been proposed. Server-side solutions include static analysis of web application code and firewall packet-filtering to avoid code injection. Client-side solutions include using browser-based solutions to detect malicious JavaScript code as well as HTTP proxy servers configured to detect cross-site scripting. Solutions to prevent HTTP response splitting include detection of CRLF characters in URL parameters.

Huang et. al propose methods for using both static type-checking analysis and data-checking guardbands for runtime protection of web application code [4]. However, they are of limited use detecting or preventing cross-site scripting because of the data-type of JavaScript: simple strings.

Hallaraker and Vigna propose detection of malicious code by auditing the execution of JavaScript code. The execution logs are then compared to known classes of attacks [5]. These methods can be easily circumvented, however, by modifying the mechanisms in which the state data is transmitted back to the attacker.

Mookhey and Burghate propose a scheme for detecting SQL injection and cross-site scripting attacks by leveraging existing network firewall-based intrusion detection systems at the web-application server site to detect the JavaScript "<script>" tags. [6] However, this approach generates many false positives, and also fails to prevent more carefully crafted attacks (such as different character encodings of the tag or fragmentation of the tag and script).

To prevent a HTTP response splitting attack, Klein [7] recommends that all CR and LF characters should be disallowed in HTTP response headers. However, many implementations have relied on double CRLF patterns or have simply blocked potentially harmful characters in the URL. As described by Klein [8] in a recent article, HTTP response splitting may be injected not only in the URL, but also in body parameters and possibly headers and path.

In recent work, Kirda et al. utilize as a web proxy as a "personal firewall" to mitigate cross-site scripting attacks. When the user requests a page, the web proxy processes the request and produces firewall rules based on the list of the static URLs (images, hyperlinks) present in the document. Using these rules, subsequent browser requests can be filtered to prevent access to a dynamically generated URL. [9]

Although this approach can be powerful, there are a few shortcomings associated with this method. First, this mechanism does not block the cross-site scripting attack, but only its mechanism for transmitting stolen user information. Since malicious code is still being executed, it is still possible to perform denial-of-service (DOS) XSS attacks or manipulate the user's web application session. The approach eliminates the mechanism through which most cross-site scripting attacks return the stolen user information back to the attacker, but transmission is still possible. One possible other technique for transmitting the stolen user information is to post the information back to vulnerable domain, in the form of a public-readable comment or similar post. Finally, with the advent of Asynchronous JavaScript and XML (AJAX), dynamically generated URLs have become common-place in many web applications.

An additional major shortcoming of the use of proxies or firewalls (either client-side or server-side) for the mitigation of cross-site scripting attacks is their incompatibility with the Secure Socket Layer (SSL). As they cannot decrypt the encrypted SSL records, cross-site scripting attacks on these secured websites cannot be detected. Because of their encrypted nature, these websites are likely to be those of major importance (e.g. banking and financial information).

References

- [1] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: A web vulnerability scanner."
- [2] D. Flanagan, JavaScript (2nd ed.): the definitive guide. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1997.
- [3] D. Endler, "The evolution of cross site scripting attacks," tech. rep., iDEFENSE Labs, 2002.
- [4] Y.W. Huang, F. Yu, C. Hang, C.H. Tsai, D.T. Lee, and S.Y. Kuo, "Securing web application code by static analysis and runtime protection," in WWW '04: Proceedings of the 13th international conference on World Wide Web, (New York, NY, USA), pp. 40-52, ACM Press, 2004.
- [5] O. Hallaraker and G. Vigna, "Detecting malicious javascript code in mozilla," in ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05), (Washington, DC, USA), pp. 85-94, IEEE Computer Society, 2005.
- [6] K. K. Mookhey and N. Burghate, "Detection of sql injection and cross-site scripting attacks," SecurityFocus: InFocus, 2004.
- [7] A. Klein "Divide and Conquer: HTTP Response Splitting, Web Cache, Poisoning Attacks, and Related Topics White Paper", 2004.
- [8] A. Klein "HTTP Response Smuggling", 2006.
- [9] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A client-side solution for mitigating cross-site scripting attacks," 2006.