SWAP: Mitigating XSS Attacks using a Reverse Proxy

Peter Wurzinger[§], Christian Platzer[§], Christian Ludl[§], Engin Kirda[¶], and Christopher Kruegel[‖]

[§] Secure Systems Lab
Technical University Vienna
{pw,cplatzer,chl}@seclab.tuwien.ac.at

[¶]Institute Eurecom
France
kirda@eurecom.fr

[‖] University of California, Santa Barbara
chris@cs.ucsb.edu

Abstract

*Due to the increasing amount of Web sites offering features to contribute rich content, and the frequent failure of Web developers to properly sanitize user input, cross-site scripting prevails as the most significant security threat to Web applications. Using cross-site scripting techniques, miscreants can hijack Web sessions, and craft credible phishing sites. Previous work towards protecting against cross-site scripting attacks suffers from various drawbacks, such as practical infeasibility of deployment due to the need for client-side modifications, inability to reliably detect all injected scripts, and complex, error-prone parameterization. In this paper, we introduce SWAP (**S**ecure **W**eb **A**pplication **P**roxy), a server-side solution for detecting and preventing cross-site scripting attacks. SWAP comprises a reverse proxy that intercepts all HTML responses, as well as a modified Web browser which is utilized to detect script content. SWAP can be deployed transparently for the client, and requires only a simple automated transformation of the original Web application. Using SWAP, we were able to correctly detect exploits on several authentic vulnerabilities in popular Web applications.*

1 Introduction

Ever since its conception, the World Wide Web has evolved towards an increasingly feature-rich, interactive, and heterogeneous medium. Unlike early Web sites, which were merely meant to deliver text in a practical fashion, nowadays' Web 2.0 sites are not only capable of hosting rich content, such as images, videos, and audio material, but also provide platforms for users to contribute such data and share it with the rest of the world. As long as the input provided by users is benign and the Web applications are used as intended, the challenges are easily met by developers and service providers. However, for various reasons, such as simple curiosity, destructive intentions, or hope for financial profit, there will always be people who aim to exploit Web sites and their users to their advantage. Therefore, even though users expect modern Web services to integrate their content seamlessly and effortlessly into the provided applications, protection of their local computer systems is required, when viewing Web content created and submitted by potentially malicious entities.

Cross-site scripting (XSS) [7, 8] continuously leads the most wide-spread Web application vulnerabilities lists (e.g., *WhiteHat Website Security Statistics Report* [29], *OWASP Top Ten* [19]). Estimates in [29] suggest that 67 percent of all current Web sites are vulnerable to XSS. In fact, not even search giant Google is spared from such attacks. With an XSS vulnerability in Google's online spreadsheet application [5] it was possible to steal a user's cookie (which was valid for all of `google.com`'s subdomains, e.g., `mail.google.com`, `code.google.com`, `spreadsheets.google.com`). Frequently, online banking applications, which make a very attractive target for XSS in order to set up phishing sites, are vulnerable to XSS, as has been demonstrated by *Phishmarkt* [1, 2].

Technically, XSS attacks leverage insufficient input/output validation in the attacked Web application to inject JavaScript code, which is then executed on the victim's machine within the exploited Web site's context, thus bypassing the *same origin policy*. The attacker can craft the injected script such, that it discloses the victim's confidential information, e.g., a session ID. Then, by hijacking the session, the victim can be impersonated. Also, XSS enables the construction of very powerful phishing pages, since the page content is actually delivered by the correct, trusted site.

The first line of defense against XSS is input/output sanitization. Malicious content can be filtered by checking for, and then escaping or disallowing, JavaScript-specific substrings in the user-provided content. However, sanitization can prove to be very difficult. The trend towards more powerful, more interactive, and therefore, more complex Web applications also means an increase in the effort and complexity in avoiding XSS vulnerabilities. From the attacker's point of view, it is enough to discover a single XSS vulnerability to be able to control the content a site serves. Unfortunately, for the developer, finding and patching every single vulnerability may cause significantly more effort. Even worse, not every developer is aware of the threat that XSS poses. As a result, in many cases, no filtering techniques are implemented at all. From a user's perspective, even expertise and caution while surfing the Web do not reliably protect from XSS, since just visiting a Web site can be sufficient for falling prey to an attack.

Since a conceptual solution for the XSS problem seems infeasible, counter measures currently focus on mitigation techniques to make up for the vulnerabilities still present. Some of the previously proposed mitigation techniques (such as BEEP [26] or Noncespaces [9]) use promising approaches to disable XSS attempts. However, they have to deal with an important problem: They require modifications not only on the server software, but also on the client's Web browser. That is, they need to be installed by users, most of which are oblivious to the damage XSS can cause, or unwilling to deal with the additional effort for properly securing their computer systems. An ideal XSS solution would solve the problem on the server and would not require the user to install any extra components.

In this paper, we present a novel approach to protect users against XSS attacks, that offers the same level of protection as previous work, but without the necessity for client-side modifications. To avoid the disadvantage of involving the end-user, we position a Web browser on a reverse proxy before the server. Our idea is based upon the fact that a Web browser on the client's machine is the ultimate receiver of JavaScript, and therefore a straightedge for script interpretation capabilities. Thus, by utilizing a Web browser, we are able to distinguish between benign (i.e., implemented originally into the Web application) and injected JavaScript code. First, we *encode* all benign JavaScript calls to syntactically invalid identifiers (*script IDs*). Second, we load each requested page in the Web browser attached to the reverse proxy, and watch out for scripts trying to execute. Clearly, all remaining scripts have not been encoded beforehand, and are, therefore, not expected, benign scripts, but injected, malicious ones. Third, after verifying that there is indeed no (malicious) script in the page, we *decode* all previously generated script IDs to restore the original code, and deliver the page to the client.

The main contributions of this paper are summarized as follows:

- We introduce SWAP, a solution for mitigating XSS attacks, by utilizing a reverse proxy equipped with a Web browser in order to detect malicious JavaScript content.
- In contrast to previously proposed solutions, SWAP does not require client-side modifications. Thus, each Web site can be protected from XSS exploits transparently for its visitors.
- We describe our implementation of SWAP, and demonstrate its efficacy in successfully detecting and preventing authentic attacks on three popular Web application's XSS vulnerabilities.

2 Related Work

Server-side mitigation. In order to spot XSS vulnerabilities in Web applications, a number of automatic testing tools have been proposed. Black-box Web application testing tools ([21, 28, 4, 12, 16]), as well as white-box vulnerability scanners ([6, 11, 25, 30, 14]) have been suggested in previous research, and are successfully used in practice. While such tools can greatly help in identifying XSS vulnerabilities, it is likely that some remain undetected, which clearly recommends additional safeguards. Also, for the owner of a Web site running a third party Web application to fix the identified bugs, requires the commitment of the developers of the Web application, which often have other priorities that seem more economically rewarding.

In [23], an application-level firewall is suggested, which is located on a *security gateway* between server and client, and which applies all security relevant checks and transformations (such as character escaping). By separating the security relevant part of the code from the rest of the application, as well as providing a specialized *Security Policy Description Language* to design it, the system helps Web developers to apply measures against XSS in a less error-prone fashion. Comparably to this work, we also use a reverse Web proxy to implement XSS mitigation strategies. However, while the security gateway operates on the incoming requests, our reverse proxy inspects the server's replies. This is preferable because it protects visitors of the page even if an attacker found a way to inject his malicious content in spite of the security gateway's checks. Additionally, by using an actual Web browser in order to identify scripts instead of a complex policy that targets various kinds of sanitization (not all relevant for XSS), our approach asks less from Web masters who wish to deploy it, and leaves less room for mistakes.

Client-side mitigation. Complementary to mitigating XSS on the server-side, there are several client-side solutions. In [10], a strictly client-side mechanism for detecting malicious JavaScripts is proposed. The system consists of a browser-embedded script auditing component, and an IDS that processes the audit logs and compares them to signatures of known malicious behavior or attacks. With this system, it is possible to detect various kinds of malicious scripts, not only XSS attacks. However, for each type of attack a signature must be crafted, meaning that the system is defeated by original attacks not anticipated by the signature authors.

Noxes [13] is a client-side Web-proxy that relays all Web traffic and serves as an application-level firewall. The approach works without attack-specific signatures. However, as opposed to SWAP, Noxes requires user-specific configuration (firewall rules), as well as user interaction when a suspicious event occurs.

Another client-side approach is presented in [27], which aims to identify information leakage using tainting of input data in the browser.

All client-side solutions share one drawback: The necessity to install updates or additional components on each user's workstation. While this might be a realistic precondition for skilled, security-aware computer users, it is perceived as an obstacle or is not even considered by the vast majority of users. Thus, the level of protection such a system can offer is severely limited in practice.

Hybrid mitigation approaches. Some solutions apply hybrid approaches, which also involve the Web browser. The server annotates the delivered content and provides information on the legitimacy or level of privileges of scripts. The Web browser is then responsible for checking and enforcing these annotations.

BEEP (*Browser-Enforced Embedded Policies*) [26] proposes to use a modified browser that hooks all script execution attempts, and checks them against a policy, which must be provided by the server. Two kinds of policies are suggested. First, using a white list of the hashes of all allowed scripts, which the browser can check against. Second, labeling those nodes in the HTML source, which are supposed to contain user-provided content, so the browser can determine whether a script's position in the DOM tree is within user-provided content. The modified browser verifies each script with respect to the policy and prohibits scripts from execution that do not comply.

In Noncespaces [9], the authors propose to use randomized XML namespaces in order to partition the content into different trust classes. The client is responsible for interpreting the namespaces and restricting the content's rights according to a policy that is provided alongside the Web site. The owner of the site can assign the desired trust levels via XPath expressions, and thus, disallow JavaScript code in HTML subtrees that are supposed to contain user-contributed content.

The mentioned hybrid mitigation techniques offer the most powerful features and the best ratio between parameterization costs and level of protection. However, they share the same drawback as the strictly client-based solutions: The requirement to being deployed on user's machines. Our solution is similar to BEEP and Noncespaces in that we use a server-provided specification of legitimate JavaScript content and detect when a script has been injected. However, our solution performs *a*ll XSS mitigation functionality on the server-side. It therefore does not require any client-side modifications, and can be applied transparently, without the user even being aware of it.

3 SWAP Overview

SWAP operates on a reverse proxy, which relays all traffic between the Web server that should be protected and its visitors (as depicted in Figure 1). The proxy forwards each Web response, before sending it back to the client browser, to a *JavaScript detection component*, in order to identify embedded JavaScript content. In the JavaScript detection component, SWAP puts to work a fully functional, modified Web browser, that notifies the proxy of whether any scripts are contained in the inspected content.

In order to differentiate between benign and malicious JavaScript, previously to enabling the proxy with the JavaScript detection component, the hosted Web application is modified. All legitimate script calls in the original Web application are encoded into unparsable identifiers, so called script IDs, and thus, hidden from the JavaScript detection component. Consequently, it is safe to assume that each script that is still found must have been injected, either via the preceding Web request (reflected XSS), or via the Web application's database (stored XSS).

If no scripts are found, the proxy decodes all script IDs, effectively restoring all legitimate scripts, and delivers the response to the client. If the JavaScript detection component, on the other hand, detects a script, SWAP refrains from delivering the response, but instead notifies the client of the attempted XSS attack.

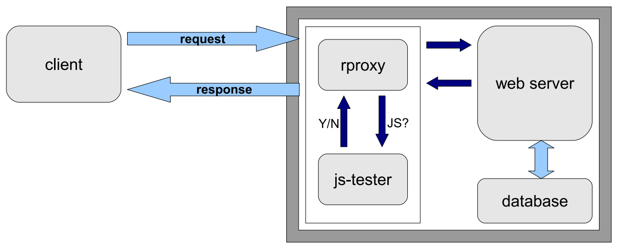

Figure 1. Scheme of SWAP setup

To summarize, the main components of SWAP are:

1. A JavaScript detection component, which, given the Web server's response, is capable of determining whether script content is present or not.

2. A reverse proxy installed in front of the Web server, which intercepts all HTML responses from the server and subjects them to analysis by the JavaScript detection component.

3. A set of scripts to automatically encode/decode scripts/script IDs.

4 Implementation

4.1 Web Application Modification

SWAP is based on the idea of rendering all legitimate JavaScripts syntactically incorrect, so that every JavaScript that is eventually executed by a browser can be concluded to be malicious. Therefore, the first step for deploying SWAP is to identify all legitimate script calls in the original Web application, and to replace each one by a unique identifier, a *script ID*. This effort has to be repeated every time a change is made to the application that alters or adds JavaScript code. Fortunately, it is easily possible to automate this step.

Generally, in order to locate legitimate scripts in the original Web application, it is advisable to utilize a similar mechanism as the JavaScript detection component later used to identify malicious scripts (as described in section 4.2). This ensures that no legitimate scripts are overseen and later erroneously reported as malicious. Since we assume that all legitimate scripts are shipped with the software and not user-contributed, obviously, this step should be performed on a fresh installation of the application, without any user-provided content in the application database. Note, that in the case where legitimate scripts are stored in the database, also these scripts must be encoded into script IDs. For the applications we used for testing, applying simple `bash` scripts using `grep` and `sed` on the source code was sufficient to accomplish the task.

There are three requirements for a script ID: First, it must not contain any valid HTML tags, so that except of removing the script, the structure of the Web page is preserved. Second, it must not contain what would be interpreted as JavaScript by a browser, so that when rendering a page it is safe to conclude that all script executions stem from illegitimately injected scripts. Third, the mapping must be reversible, so that after probing a page for scripts, the original condition with functional JavaScript code can be reestablished. For our prototype implementation, we defined a set of strings that directly indicate the presence of JavaScript code, such as the script tag (`<script>`), names of event handlers (e.g., `onclick`, `onload`), or the `javascript:` directive. Under consideration of the mentioned three requirements, we chose to replace single characters of each of these strings, such that some readability of the obfuscated content is preserved. E.g., `<script>` turns to `<scrip1>`, `onclick` turns to `onc1ick`, `javascript:` turns to `javascrip1:`,...

Note, that even though we chose to replace only certain JavaScript keywords, SWAP remembers the complete script code as a script ID. Only a character sequence which matches the complete script code, with changed keywords, will be decoded properly. It is therefore not possible for an attacker who gains knowledge of the encoding scheme, to inject an encoded script previously unknown to SWAP that will be decoded properly and subsequently executed by the client's browser. If readability of the encoded scripts is not of concern, the script IDs could as well consist of hashes of the script code.

4.2 JavaScript Detection

After having encoded all JavaScript code in the Web application into script IDs, we now expect all HTML pages returned by the server to be free of parsable script content, unless it has been injected maliciously. In order to verify that, we require a JavaScript detection component, that can determine whether a page contains JavaScript content or not.

Even though there are exact specifications on how an HTML parser is supposed to identify and interpret JavaScript code, browsers often attempt to compensate for Web developers' mistakes and also process and execute scripts that do not match the specification. Not only does this lead to incompatibilities between different browsers and their according parser implementations, but it also opens unforeseeable possibilities for a Web developer to initiate a script execution. For this reason, crafting a custom parser and basing the decision on whether it contains a script or not on its output, is likely to produce unsatisfactory results. More precisely, a parser that strictly follows the specifications would miss certain malformed scripts that a browser would execute.

For this reason, we chose to put a modified version of an actual Web browser to work in the JavaScript detection component, in order to render the page and decide whether there is script code included. We decided for Mozilla Firefox, since it is the most widely used open-source Web browser.

We have adapted the source code of the Firefox browser in only three different positions, in order to handle one type of script-embedding each: First, to get notified of scripts which are executed automatically on loading of the page, we directly hook into the code responsible for script execution. Second, to get notified of event handlers, most of which are only executed on user interaction, we hook into the code responsible for keeping track of registered event handlers. Third, to get notified of JavaScript URL link scripts, which are only executed when clicked upon, we hooked into and modified the code responsible for displaying a link in the correct color, depending on whether it had been visited before. To the best of our knowledge, all different ways of embedding valid JavaScript into an HTML document can be discovered with these three modifications.

4.3 Reverse-Proxy

To intercept, modify, and check all traffic transparently to the user, we utilize a simple, custom, Python-based reverse proxy. All (inbound) HTTP requests are forwarded unchanged to the Web server. Only the (outbound) HTTP replies are inspected more closely. Each reply is first checked on whether it consists of HTML code (as opposed to, e.g., a file download), and can therefore contain scripts. All HTML pages are forwarded to the JavaScript detection component, which determines whether the page contains any JavaScript code, and reports its findings back to the proxy. If, expectedly, no code is found, the page is deemed clean and the proxy delivers it to the client, after decoding all script IDs and restoring the original legitimate script content. If, however, JavaScript content is identified in the page the proxy obtained from the server, it is most likely injected, and the proxy returns a warning message to the client, instead of the actual content. Alternatively, a less radical solution than dropping the complete response can be implemented, e.g., to omit the malicious script content.

5 Evaluation

5.1 Detecting Script Content

The ability of SWAP to correctly detect XSS attacks strongly depends on how precisely the JavaScript detection component works in locating JavaScript content within HTML code. In order to verify that our implementation works satisfactorily also in non-traditional ways of embedding script code, we evaluated it on the XSS Cheat Sheet [22], a collection of various XSS attack code snippets, that cover a broad range of nuances regarding filter evasion. All tested examples that work in an unmodified Firefox browser have been successfully detected by our JavaScript detection component.

5.2 Detecting Authentic Attacks

In order to evaluate the quality of our prototype implementation, first, we aimed to assess its ability to correctly identify injected scripts. For that purpose, we deployed three well-known Web applications in a test environment, all of which are vulnerable to XSS, and applied SWAP as their protection. That is, we encoded all JavaScript code into script IDs in the applications' source code, and installed the reverse proxy and JavaScript detection component in front of the application server.

The applications we chose are phpBB [20], a well-known and widely used bulletin board application, phpstats [15], which enables the user to view statistics on the contents of a file system, and *Alexguestbook* [3], a guest book script for Internet Web sites.

All exploits we applied to all three testing applications have been verified to work correctly, by first trying them on a setup without SWAP protection. In all cases, we could successfully conduct a XSS attack.

First, We evaluated phpBB version 2.0.0, which is vulnerable to a stored XSS exploit [17]. After enabling SWAP, the application still performed as expected, and the usability of the Web site was not diminished. The performance detriment introduced by the additional security safeguard was unnoticeable. When attempting to inject our attack string (Figure 2), SWAP correctly recognized the attack, and returned a warning message instead of the actual content to the client.

```
[img]http://a.a/a"onerror="javascript:alert
     (document.cookie)[/img]
```

Figure 2. phpBB Exploit.

phpstats suffers from a reflected XSS vulnerability [18]. Again, no noticeable changes in performance or usability were observed after enabling SWAP. After injecting the exploit (Figure 3) into the SWAP protected application, expectedly, the XSS attack was recognized and delivery of the malicious content to the client was prevented.

```
http://mysite.org/phpstats/phpstats.php?
     baseDir=<script>alert(1)</script>&mode=
     run
```

Figure 3. phpstats Exploit.

Various versions of *@lex Guestbook* are vulnerable to multiple XSS exploits [24]. Also in this case, SWAP did not influence the behavior or performance of the applications noticeably. Two exploits (Figure 4) were executed and successfully blocked from being delivered to the user when SWAP was enabled.

```
http://mysite.org/alexguestbook4/setup.php?
     language_setup="><script>alert(document
     .cookie)</script>

http://mysite.org/alexguestbook4/index.php?
     mots_search=&rechercher=Ok&debut=0&lang
     =&skin=&test="><script>alert(document.
     cookie)</script>
```

Figure 4. @lex Guestbook Exploits.

Table 1. Page load times (ms) with and without SWAP deployment.

Size(kB)	w/o SWAP	w/ SWAP	SWAP Δ	Factor
1	27.31	196.11	168.80	7.18
10	53.84	200.50	146.66	3.72
50	120.50	331.80	211.30	2.75
100	166.23	427.66	261.43	2.57

5.3 Performance

Due to the additional requirements for processing power introduced by SWAP, clearly, a performance detriment is introduced, meaning that the client will experience higher latency when requesting content from a SWAP protected Web server, as compared to a server that does not feature SWAP protection. SWAP adds to the latency two-fold: First, by putting an additional stepping stone between client and server, namely the reverse proxy, all traffic is relayed instead of a direct transmission, and thus, takes longer to arrive at its target. Second, and more importantly, the JavaScript detection component effectively has to render each page before it can be delivered to the client. We have conducted experiments to measure the magnitude of the performance penalty inflicted by our SWAP prototype implementation.

Our test environment consisted of two machines, one to host a Firefox browser, and the other to host an Apache2 Web server. Each test run consisted of 110 reloads of a test page on the client machine. After each load, we let a Firefox extension calculate and log the load time, delete the cache and subsequently reload the page after a 500ms delay. From the 110 obtained timings, the lowest and highest five were discarded. From the remaining 100 timings, the average was taken. The tests have been conducted on test pages of 1 kB, 10 kB, 50 kB and 100 kB size respectively, both with and without SWAP protection.

The factor by which a deployment without SWAP outperforms a SWAP protected setup decreases steadily with increasing file size. This can be attributed to the constant effort for proxy relaying as well as initializing the Firefox browser used in the JavaScript detection component. However, with a higher amount of data, the JavaScript detection component's effort to render the page becomes prevalent, and the ratio apparently converges towards a value slightly greater than 2. From our set of test pages, SWAP causes the lowest additionally introduced latency at a size of 10 kB. For lower sizes, the constant effort for proxying and initialization is dominant, which leads to a drastic slowdown in comparison to a non-protected setup, whereas for higher sizes, the time SWAP requires to render the page becomes dominant.

Note, that during our tests the client and server machines were connected locally, and no Internet connection was used. In a realistic scenario, the additional latency inevitably introduced for communication between two remote hosts over the Internet would significantly add to the reported latencies, equally with and without SWAP.

6 Limitations

As already mentioned in Section 5.3, SWAP introduces a performance overhead. Even though we experienced the delay to be acceptable during our experiments, SWAP might not be suitable for a high-performance Web service. For our implementation prototype, no attempts were made to enhance processing speed, and there surely is potential for speed-ups. Most importantly, the utilization of a full fledged Web browser as the JavaScript detection component, even though it offers important benefits, could be given up in favor of a more light-weight solution, using Web scraping tools, such as Crowbar, or HtmlUnit.

Since not all implementations of the HTML protocol follow precisely the defined standards, different Web browsers, in some cases, have a different notion on what is, and what is not, valid JavaScript. While this makes a Web browser the perfect tool to identify malicious scripts for users surfing the Web with the same browser (in our case, Firefox), users employing different browsers (e.g. Internet Explorer, Opera) are not perfectly protected, since their browser might parse and execute certain content as a script, which Firefox merely treats as text, and therefore, fails to recognize as a malicious script. Nevertheless, a high level of protection for users of other browsers is still provided, because the behaviors of different browsers match for the majority of cases. Furthermore, future versions of SWAP could feature different versions of the JavaScript detection component, which are alternatively activated, according to the user agent string.

Finally, SWAP's capability to detect maliciously injected content is limited to JavaScript. It cannot defend against other types of undesirable content, such as static links pointing to sites including malicious scripts.

7 Conclusion

We presented SWAP, a server-side solution for protecting users of a Web application from cross-site scripting attacks. SWAP operates on a reverse proxy, intercepting all HTML responses, and forwarding them to a JavaScript detection component, consisting of a full fledged Web browser. Due to previous, automated modifications to the Web application, this component is able to distinguish between benign, and malicious scripts. The proxy prevents each malicious response from being delivered to the client, and thus effectively inhibits the attack to be carried out on the client's browser. We have implemented a prototype, and conducted experiments, showing the efficacy of SWAP to successfully detect and defeat cross-site scripting attacks.

Acknowledgments

This work has been supported by the Austrian Science Foundation (FWF) under grant P18764, Secure Business Austria (SBA), and the WOMBAT and FORWARD projects funded by the European Commission in the 7th Framework.

References

[1] Phishmarkt :: de. http://baseportal.com/baseportal/phishmarkt/de, 2006.

[2] Phishmarkt :: at. http://baseportal.com/baseportal/phishmarkt/at, 2007.

[3] A. Soulard, P. Gieling, M. Hercelin and J. Boulmont. @lex Guestbook. http://www.alexguestbook.net, 2008.

[4] Acunetix. Acunetix Web Vulnerability Scanner. http://www.acunetix.com/, 2008.

[5] B. (BK) Rios. Google XSS. http://xs-sniper.com/blog/2008/04/14/google-xss/, 2008.

[6] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Security and Privacy Symposium*, 2008.

[7] CERT. Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests. http://www.cert.org/advisories/CA-2000-02.html, 2000.

[8] D. Endler. The Evolution of Cross Site Scripting Attacks. Technical report, iDEFENSE Labs, 2002.

[9] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[10] O. Hallaraker and G. Vigna. Detecting Malicious JavaScript Code in Mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2005.

[11] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[12] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web Vulnerability Scanner. In *World Wide Web Conference*, 2006.

[13] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *21st ACM Symposium on Applied Computing (SAC)*, 2006.

[14] G. D. Lucca, A. Fasolino, M. Mastoianni, and P. Tramontana. Identifying cross site scripting vulnerabilities in web applications. In *Sixth IEEE International Workshop on Web Site Evolution (WSE)*, 2004.

[15] M. Wagner. phpstats 0.1 alpha. http://www.michael-wagner.de/software/phpstats/, 2008.

[16] S. McAllister, E. Kirda, and C. Kruegel. Expanding human interactions for in-depth testing of web applications. In *11th Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[17] NIST National Vulnerability Database. CVE-2002-0902: Cross-site scripting vulnerability in phpBB 2.0.0. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2002-0902, 2002.

[18] NIST National Vulnerability Database. CVE-2008-0125: Cross-site scripting (XSS) vulnerability in phpstats.php. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2008-0125, 2008.

[19] OWASP. OWASP Top Ten. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2007.

[20] phpBB. phpBB web forum software. http://www.phpbb.com, 2008.

[21] PortSwigger. Burp Suite. http://portswigger.net/suite/, 2008.

[22] RSnake. XSS Cheat Sheet. http://ha.ckers.org/xss.html, 2008.

[23] D. Scott and R. Sharp. Abstracting Application-level Web Security. In *11th World Wide Web Conference*, 2002.

[24] SecurityFocus. @lex Guestbook Multiple Cross-Site Scripting Vulnerabilities. http://www.securityfocus.com/bid/28519/, 2008.

[25] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Symposium on Principles of Programming Languages*, 2006.

[26] T. Jim and N. Swamy and M. Hicks. BEEP: Browser-Enforced Embedded Policies. In *16th International World Wide Web Conference (WWW2007), Banff*, 2007.

[27] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[28] Web Application Attack and Audit Framework. http://w3af.sourceforge.net/.

[29] WhiteHat Security. Website Security Statistics Report. http://www.whitehatsec.com/home/resource/stats.html, 2008.

[30] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium*, 2006.