Web Application Security

By
Joe McCormack, April 2009

Table of Contents

Introduction 3

Mashups, Gadgets, Widgets and Dashboards 4

Google Gears and Javascript 5

Adobe Flash 6

The Captcha 7

Security Holes Galore 7

Improving Web Application Security 9

Conclusion 13

LIST OF FIGURES 16

REFERENCES 17

Introduction

According to Moscaritolo (2009), web application vulnerabilities comprise eighty percent of web-related flaws. Among web browsers 43 percent of vulnerabilities were attributed to Internet Explorer, 29 percent attributed to Firefox and 10 percent attributed to Safari. What is disturbing is that in the article, Sergey Gordeychik, contributor for WASC (Web Application Security Consortium) suggested that security requirements are not often considered in the system design of web applications. Automated scanners allow attackers to easily detect security vulnerabilities in web applications which may be why so many vulnerabilities are found to be web application based. Help Net Security (2008) wrote an article that demonstrated, based on the WASC Web Application Security Statistics Project 2007, that (1) 41 percent of website vulnerabilities were XSS, (2) 32 percent were due to information leakage, (3) 9 percent were due to SQL injection, (4) 8 percent were due to predictive resource location and the remainder due to other vulnerabilities.

With that information in mind, the purpose of this paper is to cover web applications in further detail by exposing current content usage trends of users and websites using different technologies. After exposing different technologies and what they provide, discussion shifts to how vulnerabilities native to and through those technologies can propagate to web applications, and from those web applications the vulnerabilities can continue down-stream to other organizational assets such as a database or mail server. Typically, a web application's security strength is determined by the knowledge, skill and competence of the team member(s) developing the web application through their knowledge of not only the environment of the web application but also by their knowledge of external influences on the web application. By the end of the paper I expose several programs and resources that can be used to help secure web applications from security vulnerabilities. In today's interconnected world it is critical to recognize and understand the significance of external influences (attack methods), how those influences effect the web application and how the web application can impact other systems by acting as a security vulnerability relay.

Mashups, Gadgets, Widgets and Dashboards

A mashup is a program that takes as input (1) content or functionality from an external or internal website, (2) web service content or functionality from an external or internal website, and, (3) aggregation services such as RSS, in order to combine those inputs to create output that is not available from the inputs by themselves. A mashup program is commonly assumed to be web-based in nature and requires some degree of integration with a web server at the web application level; for instance, you import a API, probably in the form of a DLL, into a .Net web project combining data from the API and data unique to your web project to generate output that is embedded in a webpage or creates output such as XML, thus becoming a mashup. Programmable Web (http://www.programmableweb.com/howto) contains a list of API's for mashups. If you've ever combined functionality or data of an external application/system into your own application/system you've, in essence, created a mashup.

A widget, labeled "gadget" by various companies like Microsoft and Google, are stand-alone applications that can run from the desktop (installed on a computer) which could be an executable or a web-browser plug-in/extension. The other type of "widget" or "gadget" is one which is designed to run in a webpage using web-browser plug-ins such as Adobe Flash in that code is embedded into the webpage using the <object> or <embed> tags, optionally passing parameters, to the location of the SWF file. The SWF file then creates output that is typically viewed in the webpage. Widgets are used extensively by social sites since end-users do not need to have any real programming capability, and, the trend of Internet usage today is less on seeking data to that of having data delivered from multiple sources. Unlike a mashup, a widget typically acts like an RSS aggregator in that they grab data (input) from a single source in order to create output.

A dashboard is a program which organizes content in a way that is easy to read and is usually customized by the end-user. A dashboard is nothing more than a widget functioning as a mashup by combining multiple widgets. A dashboard is commonly desktop based (for example, there are dashboards for Microsoft Excel; Mac OS X Tiger allows you to use a feature known as "Dashboard" to interact with local and remote sources/data from the desktop). There are also digital dashboards, such as .Net Dashboard Suite™ (http://www.perpetuumsoft.com/Product.aspx?lang=en&pid=44) designed to run from within a webpage and are commonly oriented to add visual appeal to reporting data; however, they are not truly a dashboard in the sense of what you expect with desktop dashboards.

Google Gears and Javascript

Amit (2008) revealed that a widely-used RIA infrastructure known as Google Gears (used by various services such as Google Docs, MySpace, WordPress and others) is a web browser extension (similar to Adobe Flash in terms of what a web browser extension is) that allows developers to create web applications that run online and offline transparently and can be embedded into HTML pages by embedding javascript calls to the Google Gears API. Google Gears is commonly used because it can simplify the creation of mashups. Although now patched, Google Gears suffered from a vulnerability with cross-origin communication that allowed attackers to circumvent the same-origin policy to launch large-scale user-impersonation attacks. With so many websites using Google Gears that vulnerability (basically allowing non-Google Gears code to execute from a Google Gears worker) could have compromised an unknown amount of data.

Of related interest is that many of the “offline” capabilities of Google Gears are standard DOM web browser features with the HTML 5 specification. Previously it was mentioned that Google Gears followed a RIA Infrastructure. As discussed by Domenig

(2009), RIA (Rich Internet Applications) technology has the goal of combining the advantages of desktop applications with web applications.

The javascript DOM (Document Object Model) is an exposed architecture that allows you, as the creator of a webpage (although you do not have to create a webpage to gain access to the DOM since you can, for instance, create and execute entire javascript functions from within the URL location bar of web browsers) to interact with the web browser in different ways beyond the scope of this paper. XSS (Cross-site scripting) commonly involves javascript. Although it has legitimate use (such as using trusted javascript from Site B in Site A as demonstrated by McCormack (2008)), XSS is commonly associated with javascript as (1) being embedded into a webpage (such as from posting to a blog) enabling the entity who embedded the javascript code to do things like steal cookie data from someone else visiting the webpage to hijacking and impersonating that visitor, (2) phishing wherein someone clicks on a masked link going to a webpage that opens a vulnerable page installed locally and from there (the computer's local zone) the script runs commands with the end-users privileges, and, (3) CSRF (Cross Site Request Forgery) discussed later. As demonstrated by WASC (2006), attacks such as injection is not limited to web applications or databases (such as SQL Injection through the web application), but can also be launched against a mail server when the medium between the end-user and the mail server is a web application.

Adobe Flash

Flash applications have become extremely popular for website developers due to the visual appeal, interaction and portability that is possible across multiple web browsers. These applications have also become popular in creating web-driven content and widgets that are in heavy use with users of social sites. Flash applications, whether it may be a presentation, movie, game or simulation can be viral wherein the SWF may not only be downloaded into an end-user's temporary internet files folder on their computer to enable it to be run, but it could also be placed on other web servers (an

example is by the end-user uploading the SWF into their own web space and then linking to it from a webpage). Adobe Flash itself is a web-browser plug-in, and because it is a plug-in that is installed directly onto an end-user's computer, it has the potential to perform activity beyond the control of the web browser that it is operating within. The common methods of embedding a Flash application, as mentioned previously, is to use the <object> or <embed> tags although you could directly link to an SWF file.

The Captcha

Another method commonly used to verify the entity requesting something from your web application is the real user (and not, for instance, a bot) is through the use of CAPTCHA (Completely Automated Public Turing Test To Tell Computers and Humans Apart). Basically a CAPTCHA is a web application that presents a graphical representation of a word or phrase in distorted form, and with some method of interference behind or in front of the word or phrase to make it "impossible" for a bot, or computer program, to interpret. In conjunction with other security measures, a CAPTCHA may be effective, but in some cases (such as the one involving Amazon.com) a CAPTCHA should not be regarded as the primary means of protecting a website from bots. Ha.ckers (2007) wrote that there are teams of individuals that do nothing but use software to solve CAPTCHAs and do so from overseas and one such team was used to effectively "break" Amazon.com's CAPTCHA system to create thousands of users for a malicious activity. In order to utilize a CAPTCHA effectively, other factors in web application development must be appropriately addressed.

Security Holes Galore

Exposing a security vulnerability in one application typically requires the use of another application and the level of native trust between them. For example, if an end-user clicks on a link to a Flash SWF file Internet Explorer, Firefox and Google Chrome web browsers will automatically create a default set of HTML code (a document) so that

the Flash SWF file will be embedded and then rendered in the web browser window. Unfortunately, in the case of Internet Explorer 6, 7 and 8 the “allowscriptaccess” property defaults to “samedomain” instead of “none” according to Guya.Net (2009). This presents a security vulnerability that can be exploited by the SWF file because the SWF file can use its native “ExternalInterface.call(‘eval’, ‘script’)” capability to reload the web browser-generated HTML code to bypass the security measure employed by Internet Explorer to not allow access to the dynamically generated document; neither Firefox or Google Chrome recognize a plausible security vulnerability in the first place (so it may be possible to gain access to the document source code by simply typing “javascript:alert(document);” in the URL bar after the SWF has loaded in the web browser-generated HTML code page) without you or the SWF reloading the page.

A new, emerging security vulnerability involves what is known as CSRF (Cross-Site Request Forgery) that can be achieved with Adobe Flash by using Flash’s native ability to inject javascript into the webpage it is embedded within, noted by Guya.Net (2009). With this ability, the crafter of the Adobe Flash SWF file can embed a <script> that could perform a variety of functions to extract data from a target website; and using Adobe Flash’s cross-domain post capability allows you to handle larger volumes of data that a get method prohibits. For example, you login to Site A. Site A authenticates you and you are (most likely) assigned a session. You don’t log out of Site A. You then travel to Site B which has a malicious SWF file that you access. That SWF file uses code generated with a <script> injection and can then access data from Site A using your assigned session. The responses from Site A are captured and the SWF file can store those responses into a database or another location. What is interesting is with a CSRF attack, an SWF file does not have to be used (although the SWF file does obfuscate, or hide, the code) as was revealed by Techreads.Com (2007) where common DOM techniques were used that worked in multiple web browsers. But, as revealed by Heiko (2008), the SRC attribute of a standard <img> tag can have the same effect by using a SRC path such as targetsite.com/dataRequest.aspx?sourceaccount=1443&action=transferfunds&destinationaccount=1000. The visible effect (assuming the image area is visible to the end user)

would be nothing more than a broken image icon since the response from the target site would be in a format that was not an image type.

Another exploitation method is browser-based cross-zone scripting where a script gets executed in a trusted (privileged zone) zone, as configured in the web browser, and continues with the use of an insecure ActiveX component on the end-users computer. HTTP response splitting, on the other hand, is a web application vulnerability that surfaces when the web application does not sanitize input values that are sent to it (for instance, from a web form).

It has long been thought that you can verify that the entity requesting something from your web application can be validated by checking the IP address of the entity against what was originally recorded; however this is only useful if the attacker is not behind the same NAT'ed (Network Address Translation) IP address or web proxy as the real end-user. However, newer versions of Internet Explorer, Firefox, Safari and Google Chrome use the "HttpOnly" flag which allows your web application to set a cookie that is unavailable to client-side scripts.

Improving Web Application Security

When you develop a web application or use one that was already built, you have to keep in mind that it could be attacked or used by someone with considerable knowledge of not only the product used to create the application (such as intimate knowledge of Adobe Flash), but also technologies related or used in conjunction with the application (such as, with the case of Adobe Flash, knowledge of web browser behavior, DOM, low-level protocols such as GET/POST, and other applications). Before you roll out any web application it should be tested for security and exercise good security practices such as not placing session or authentication data into public-facing

(client-script accessible) cookies or session data as shown in Figure 1. Some applications that could be used for security testing include:

- Acunetix WVS - http://www.acunetix.com/
- Automated Adobe Flash/Flex Crawling and Scanning - http://www.authorstream.com/Presentation/orysegal-88813-automated-flash-flex-crawling-scanning-web-application-security-testing-owasp-il-2008-ronen-bachar-ria-science-technology-ppt-powerpoint/
- Core Impact Pro - http://www.coresecurity.com/content/core-impact-overview
- Fortify 360 - http://www.fortify.com/products/fortify-360/vulnerability-detection.jsp?location=location1
- IBM Rational AppScan Standard Edition - http://www-01.ibm.com/software/awdtools/appscan/standard/
- Retina Web Security Scanner - http://www.eeye.com/html/products/RetinaWebScanner/index.html
- WebInspect 8.0 - https://download.spidynamics.com/webinspect/default.htm

Additionally there are many online resources that you can visit that list current and past vulnerabilities which may be useful in security testing web applications (in addition to other software and systems) such as:

- Computer Security Vulnerabilities - http://www.security.nnov.ru/
- National Vulnerability Database - http://nvd.nist.gov/
- US-CERT Cyber Security Bulletins - http://www.us-cert.gov/cas/bulletins/

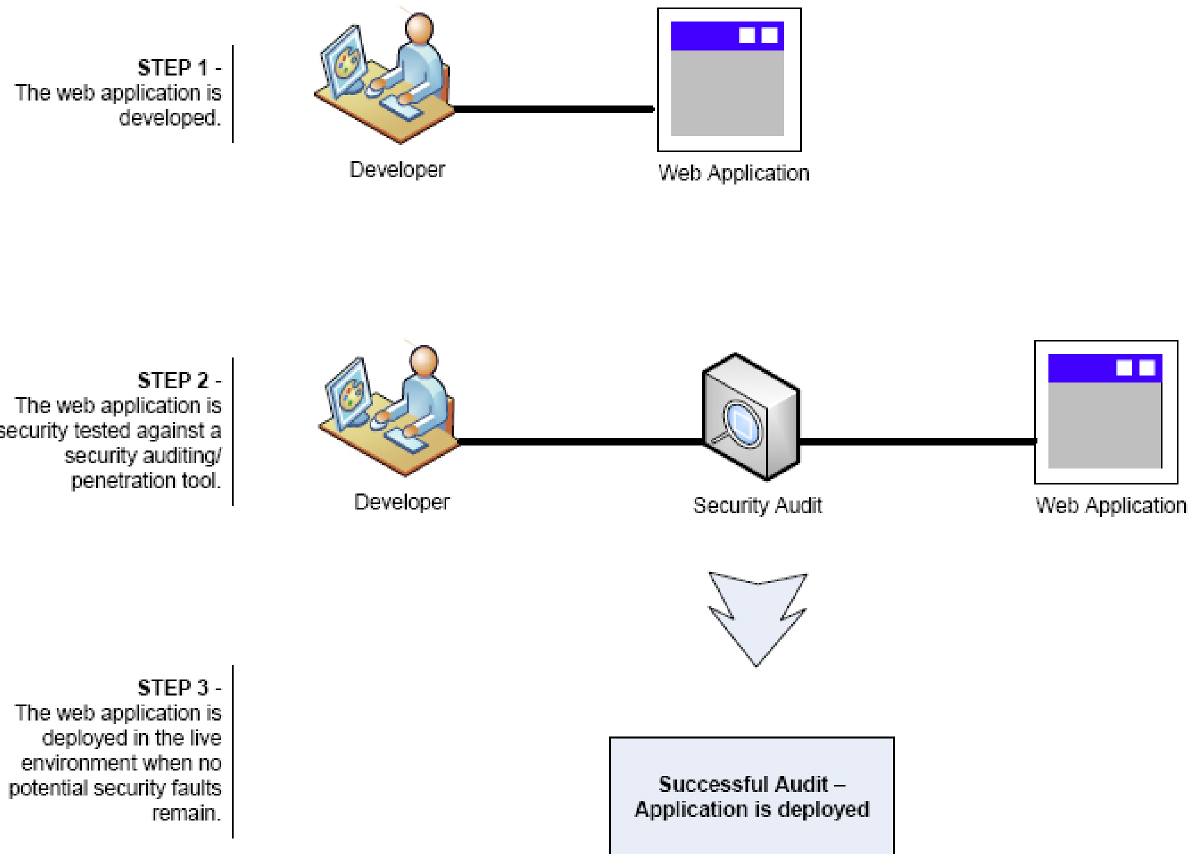

Figure 1: Web Application Auditing

With web application development it is quite conceivable that a web application may accept a file to upload and then have that file immediately exposed to the web environment so that website users may interact with it (for example, someone updates their forum avatar image or uploads a zip file). There are a few businesses that make antivirus engines available, via SDK/API (Software Development Kit/Application Programming Interface) at the application level. This means it is possible for the application to scan the uploaded file immediately and act upon the file, as needed, instead of relying on an external service as shown in Figure 2. This type of ability allows the application to be more proactive, responsive and event logging capable where data relating to a user may be recorded with greater detail than a system not directly tied to the application. Some antivirus SDK/API's are available from:

- AhnLab Engine SDK - http://global.ahnlab.com/global/products/business_engine.html
- AntiVir SAVAPI - http://www.avira.com/en/solutions/system_integration.html
- Frisk F-PROT - http://www.f-prot.com/partners/oem/
- VIPRE SDK - http://www.sunbeltsoftware.com/Developer/VIPRE-SDK/

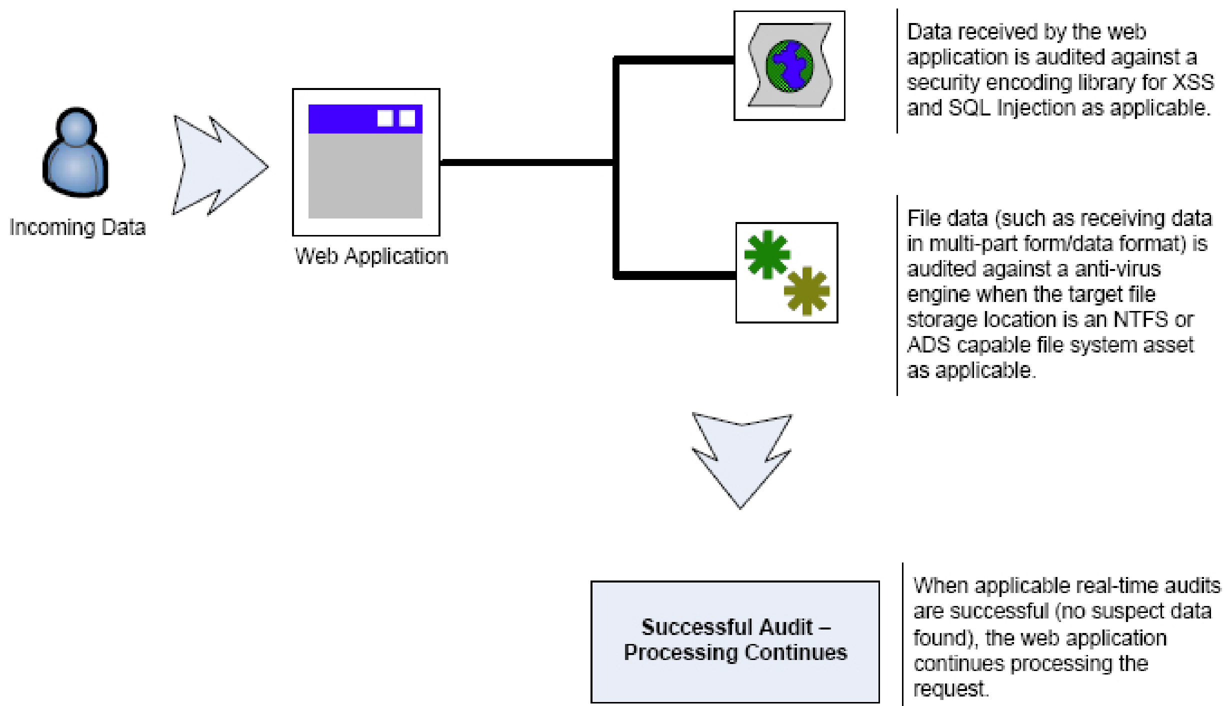

Figure 2: Web Application Real-time Audit of Incoming Data

Where you have web applications which accept data, that data should be sanitized so that it cannot be sent to an end-user and interpreted and executed by the web browser (such as providing an end-user with the capability of submitting a web form where they may type in client-side script, or use a RTE (Rich Text Editor)), and, the data should be protected with a valid and recognized SSL certificate. Optimally, the data should be sanitized against a security encoding library, as mentioned by OWASP (2009) and not just escaped (double-encoding such as %253C for < and %253E for > can be

used to get around some sanitization attempts). One such security encoding library, called AntiXSS is available from http://www.codeplex.com/AntiXSS. Your web application should only accept data based on context. That is, you do not want a web application which accepts data from a large web form to allow GET submissions in addition to POST submissions (remember, simple XSS vulnerabilities will use query strings which use the GET method). This is not a fail-safe method to help protect your web application (since, using client-script, a request can be made to mimic a POST submission in a link or in other scenarios), but it is a good first step. As well, do not forget that a web service does not always need to have GET and POST methods enabled, as indicated by Meier et al (2003).

Conclusion

When you develop a web application, ensure that incoming data is audited by using a security encoding library. Try not to place sensitive data into cookies (such as an account login or password), if cookies have to be used for something, try “HttpOnly” cookies that are stored on the client which are not accessible by client-side script; use session data sparingly. If binary data is accepted (such as the upload of a file), that file should be sanitized of any virus, worm or malware which may be embedded in it by a feature of Windows NTFS, and other file systems, known as ADS (Alternate Data Stream). The simplest way to remove what was attached to the original file (by using ADS) is to place the file on a FAT or FAT32 partition (FAT does not support ADS) and then move the file to the destination location; but in most instances real-time anti-virus applications that can be integrated into the web application are preferred. With regard to handling incoming data, ensure that the web application only accepts the data method expected. For example, if you have a web form set up which is sending data using the POST method, the web application should only accept incoming data through the POST method. Don’t forget to test the web application against an effective security auditing tool so that known vulnerabilities can be detected and they can be resolved before deploying the web application. Finally, secure the transport of data between the client and your web application with an SSL (Secure Sockets Layer) certificate which will

encrypt data being passed back and forth instead of being passed as clear text where anyone could intercept and read the data with minimal effort.

Consider reading “Active Man in the Middle Attacks” available online, written by Saltzman and Sharabani (2009) which explains in fine detail different vulnerabilities that can be relevant to web applications. As well OWASP (2005), the Open Web Application Security Project organization has a variety of educational presentations on several aspects of web application security that are worth reviewing. SANS Institute (2008) provides a list of tools, organized by scope to assist in protecting web applications from attacks in addition to other assets, such as application IDS/IPS (Intrusion Detection System/Intrusion Prevention System). In conjunction with an application IDS/IPS, a web application should log all requests that are made to it for auditing purposes (auditing of a SQL database has finally been made a trivial process with SQL Server 2008 because full auditing is built-in and managed natively from SQL Server 2008 so you don’t need to write a limited application to attempt the same thing such as what you can, to a limited degree, observe with SQL Profiler). Although not directly related to a web application, it would be prudent to have a measure of additional control over services running on the web server that may be exploited through a web application. Hameed (2008) demonstrates how services can be locked-down on Windows Server 2008. As well, Microsoft TechNet (2009) has created a security compliance management toolkit that could aid in improving security of the Windows 2008 Server.

If you plan on supporting mashups, widgets, gadgets or dashboards on your website when either those applications or the data those applications import come from an external source, beyond your control, you should consider the ramifications. Some of which are outlined below as specified by Safe Mashups (2009):

- Unanticipated entry point into internal applications from a desktop
- Unanticipated exit point for sensitive corporate information

- A new way to compromise a desktop that may be authenticated to resources that divulge sensitive private information

It is important to also point out that in losing a degree of control and security by using one of the aforementioned tools, you could open yourself up to classic vulnerabilities that can be exploited through the tools themselves (if those tools are not written and updated to current security standards and practices) such as cross-zone scripting, HTTP response splitting, CSRF attacks, phishing and XSS. From the perspective of web application development, those may not seem like significant concerns since the tools being adopted or allowed to be used are 3rd party tools primarily affecting your website users' web browsers…it is important to realize that getting into your website (your website applications or assets such as sensitive data) starts by being able to see through the window you've provided. In the context of this paper that window is the user of your website.

LIST OF FIGURES

Figure	Page
Figure 1: Web Application Auditing..	11
Figure 2: Web Application Real-time Audit of Incoming Data..	12

REFERENCES

Amit, Y. (2008). *AJAX Secuity: Breaking Google Gears' Cross-Origin Communication Model.* Retrieved from http://blog.watchfire.com/wfblog/ajax_security/ on March 26, 2009.

Carnegie Mellon University (2000). *CAPTCHA: Telling Humans and Computers Apart Automatically.* Retrieved from http://www.captcha.net/ on April 5, 2009.

Domenig, M. (2009). *Rich Internet Appliations and AJAX - Selecting the Best Product.* Retrieved from http://www.javalobby.org/articles/ajax-ria-overview/ on March 13, 2009.

Guya.Net (2009). *Bug in Internet Explorer Security Model When Embedding Flash*.
Retrieved from http://blog.guya.net/2008/09/10/bug-in-internet-explorer-security-model-when-embedding-flash/ on March 4, 2009.

Guya.Net (2009). *Encapsulating CSRF Attacks Inside Massively Distributed Flash Movies - Real World Example*.
Retrieved from http://blog.guya.net/2008/09/14/encapsulating-csrf-attacks-inside-massively-distributed-flash-movies-real-world-example/ on March 4, 2009.

Ha.ckers (2007). Solving *CAPTCHAs for Cash.* Retrieved from http://ha.ckers.org/blog/20070427/solving-captchas-for-cash/ on March 9, 2009.

Hameed, C. (2008). *WS2008: Windows Service Hardening.* Retrieved from http://blogs.technet.com/askperf/archive/2008/02/03/ws2008-windows-service-hardening.aspx on April 27, 2009.

Heiko (2008). *CSRF - An Underestimated Attack Method.* Retrieved from http://www.rorsecurity.info/2008/05/05/csrf-an-underestimated-attack-method/ on March 20, 2009.

Help Net Security (2008). *The Web Application Vulnerability Landscape.* Retrieved from http://www.net-security.org/secworld.php?id=6501 on April 20, 2009.

McCormack, J. (2008). *Javascript XSS: Cross-Site Scripting and Digital Signatures.* Retrieved from http://www.virtualsecrets.com/misc_imgs/javascript/javascrpt-xss-cross-site-scripting.html on April 14, 2009.

Meier, J., et al (2003). *Checklist: Securing Web Services.* Retrieved from http://msdn.microsoft.com/en-us/library/aa302349.aspx on April 22, 2009.

Microsoft TechNet (2009). *Windows Server 2008 Security Compliance Management Toolkit.* Retrieved from http://technet.microsoft.com/en-us/library/cc514539.aspx on April 27, 2009.

Moscaritolo, A. (2009). *Web Apps Account for 80 Percent of Internet Vulnerabilities.* Retrieved from http://www.scmagazineus.com/Web-apps-account-for-80-percent-of-internet-vulnerabilities/article/129027/ on March 26, 2009.

OWASP (2005). *OWASP Presentations.* Retrieved from http://www.owasp.org/index.php/Category:OWASP_Presentations on February 18, 2009.

OWASP (2009). *XSS (Cross Site Scripting) Prevention Cheat Sheet.* Retrieved from http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet on March 21, 2009.

Safe Mashups (2009). *OpenAJAX and Similar Mashups.* Retrieved from http://www.safemashups.com/home/public_html/solutions_openajax.html on April 17, 2009.

Saltzman, R., Sharabani, A. (2009). *Active Man in the Middle Attacks: A Security Advisory. A whitepaper from IBM Rational Application Security Group.* Retrieved from http://blog.watchfire.com/AMitM.pdf on April 3, 2009.

SANS Institute (2008). *Tools for Defense In-Depth.* Retrieved from http://www.sans.org/whatworks/wall.php?id=2 on March 16, 2009.

Techreads.Com (2007). *Gmail Vulnerable to Contact List Hijacking.* Retrieved from http://www.cyber-knowledge.net/blog/2007/01/01/gmail-vulnerable-to-contact-list-hijacking/ on April 6, 2009.

WASC (2006). *MX Injection: Capturing and Exploiting Hidden Mail Servers.* Retrieved from http://www.webappsec.org/projects/articles/121106.shtml on April 17, 2009.