Global Information Assurance Certification Paper

Secure Your Web Application through Security Testing

Darren Gannon – System Administrator

Submitted: July 13, 2004

Abstract

The internet has forever changed the way companies do business. Technology progresses quickly and what may have been considered secure a year ago often is considered insecure today. In the past simply having a firewall was considered a considerable investment in securing a businesses information assets. Today malicious users are attacking systems and going right through the firewall. They do this by attacking the web application sitting behind the firewall. Many web applications are vulnerable to specially crafted attacks. Often the applications go untested for these attacks. Developers may lack knowledge of new vulnerabilities in software, the QA department may lack the knowledge of security testing techniques and the networking team is often untrained in application security.

Three common attacks are buffer overflows, sql injection and cross site scripting. There are simple ways to test applications for all three exploits and often there are tools to automate the process. In this essay we will examine the need for security testing as a separate function in a development cycle. It will explain the mechanics behind the three attacks above and how to test your applications for vulnerability to these three attacks.

For many companies the security of their web applications and infrastructure is an afterthought. While corporate executives claim that security is a top priority, often the security budget does not substantiate that claim (Jaquith). Customers

have grown accustomed to the reactionary nature of the software development companies building the applications. The “release now, fix later” approach is the norm amongst major software vendors and we’re starting to reap the results of this faulty process. In many development shops timelines and budgets do not allow for thorough security testing, while the client assumes the application is secure because it was purchased from a well known vendor (often at high cost). Others have a false impression that security is a network layer issue addressed by installing expensive firewalls and network security devices.

3.1 We’re safe, we’ve installed a firewall!

One would be hard pressed to find a Fortune 500 company whose private network was connected to the Internet without a firewall. Many businesses use a combination of firewalls along with elaborate intrusion detection systems. However, if these devices are the magical key to a secure website why is it that popular sites belonging to EBay, Microsoft, The NY Times and the FBI, to name a few, have been successfully attacked in recent years? Had they not invested in secure firewalls? Did they not employ trained security experts? One would have to assume that this is not the case, but if almost every business is protected at the network layer how are web sites and applications being successfully compromised?

It’s true that a firewall is only as secure as it’s configured to be. You can spend thousands on a home security system but if you disable the sensors on the windows so you can climb through when you’ve lost your keys, then obviously the effectiveness of the alarm is compromised by the insecure configuration. However, for the purposes of this article we’ll assume every firewall is configured properly and locked down. Even when this is the case, most firewalls leave open a gaping hole.

The increase in the use of firewalls has resulted in a change in tactics amongst the hacker community. In the late 90’s the preferred weapons amongst malicious users “were crude Molotov cocktails aimed at firms’ network infrastructure,” reports Andrew Jaquith of @Stake (Jaquith). Today the network layer is heavily guarded so it’s no surprise that the majority of successful attacks are taking place at the application layer. The path of least resistance for an attacker is to hit the application directly since most firewalls protecting web servers allow all http traffic through via port 80 (Abner).

Web Applications, the New Target

“Web services’ fundamental architecture opens the door for serious security breaches,” states Adam Kolawa (Kolawa). The very nature of a web application makes it insecure due to its public accessibility. Even when your website is behind a firewall it’s very difficult to block malicious input coming directly down the same path proper input takes. By opening up any portion of your web

application for public access you immediately have the potential for misuse of that access and attacks on the application. SPI Labs estimate that approximately "70% of all successful attacks are now coming through the application layer" (SPI Labs).

In today's competitive software market, customers demand feature rich applications with expanding complex functionality. This immediately gives the attacker a head start since, as an @Stake report concludes, "complexity is the enemy of security beyond all others" (Abner). With millions and millions of lines of code in many complex applications a malicious user can attack one component at a time with a handful of tools and well known exploits until something reveals an unexpected result, giving up clues as to further weaknesses in the system.

Some blame the development process itself in which hundreds of developers often work in a structured environment on individual components of an application. Many developers are more concerned with getting something to work rather than checking to see what could make it fail or produce the unexpected (Cenzic). In many organizations, applications are released for production with the assumption that the network infrastructure will provide security. In other cases the application developers lack the security know-how while the network infrastructure team is unfamiliar with the application. The web server and firewall may be locked down while the web application is full of potential holes (Wood).

4.1 Security Testing, Not an Option

Under these conditions, it's imperative that the application be thoroughly tested for security. In a perfect world security would be a part of both the requirements and design phases of development. However, until the market requires software developers to accept more responsibility for the security of their products this is unlikely to happen (Abner). The attitude that if something's broken we'll fix it later is accepted as the norm. If any security testing is done, it's usually the responsibility of the QA department, who may not be trained to test for security gaps. It can be a daunting task, but it is one that needs to be handled thoroughly.

QA testers often test applications for success or failure under normal working conditions. In order to do proper security testing, applications and web sites need to be tested for the abnormal. The one advantage security testers have is that most hackers resort to a handful of well known exploits when attacking your web applications. Some of the most popular are:

- Buffer Overflows/Overruns
- SQL Injection
- Cross-Site Scripting

5.1 Buffer Overflows

Buffer overflows (also know as buffer overruns) have been wreaking havoc on web applications (especially C and C++ apps) for decades (McGraw). Instead of diminishing in popularity, they are still a favorite in a seasoned hacker's toolbox of nasty tricks. Just last year Microsoft released security bulletin MS03-007 regarding a buffer overrun vulnerability in an operating system component used by WebDav in Windows 2000 and NT boxes running IIS (Microsoft).

In order to effectively test for buffer overruns, we must first understand what they are and how they are initiated. Simply put, a buffer overrun is when more data is written to a buffer than the buffer can hold. A buffer is a region of memory used by the program to temporarily store input data. The application requests a buffer of the exact size of the data it has to store. If the amount of data is too large for the buffer, it overwrites the allocated buffer corrupting whatever instruction was contained next in memory. The result can be damaging to the application, usually resulting in poor performance or an application crash (Cole).

Buffer overruns can also be used with stack smashing to compromise a system entirely. If buffers in the program's run time stack are overrun and the hacker can insert code to open a shell you've got a much bigger problem (Cole). Some might claim the chances of this occurring are remote, but when a hacker has almost unlimited time and patience to try over and over again it's imperative that we remove any possibility, remote or not. Since buffer overruns can be avoided altogether through thorough testing and a secure development process, there's no reason to give the enemy a head start in cracking your application.

Testing for buffer overruns should begin at the development phase with careful code checks and memory calculations. This doesn't protect against a faulty code check or the problems associated with faulty assumptions, such as the byte size for all characters being one byte, or Unicode characters all being 2 bytes (Ash). Therefore, all code should be checked for buffer overruns by the security tester whose goal is to input more data into the buffer than is allowed by the application. This can be done with simple brute force tactics. For example:

- Knowing the expected maximum data size of a given user input, insert a value that goes beyond the maximum allowed characters and observe the result. Experiment with double byte characters or even 4 byte GB18030 characters (Ash).
- If an application can be run from the command line, try running the executable followed by various amounts of unexpected characters and recording the outcome (Ash).
- Try adding unexpected parameters to the URL of a web application and watching for unusual behavior or server errors. At one time Netscape and Internet Explorer had URL size limitations. IIS was configured to allow the maximum amount of characters that Internet Explorer was capable of sending. Netscape was capable of sending URLs much larger in size and a buffer overflow vulnerability was discovered (Ash).

To put it bluntly, the tester needs to use “dirty” testing techniques on the application. As Michael Howard suggests, “lie, cheat, swindle, confuse, destroy and be evil -- think like the attacker” (Howard). Knowing every path that data can take into the app, and the maximum amount of data that should be allowed, try different combinations of input to test validation. Can the database be accessed directly without using the user interface? Possibly the database will allow more data than the user interface. These testing techniques are other examples of brute force attacks. For many buffer overruns more extensive testing is required using specially designed tools (Howard).

There are many tools freely available on the internet for testing applications against buffer overflows. When used properly many of the same tools malicious users’ abuse to attack your site or application can be of great benefit to a security tester in identifying buffer overflows and other vulnerabilities. (Many of these tools can be found at www.packetstormsecurity.org.)

One valuable tool is Lookout from Algenta Software. Lookout allows the tester to connect to a networked system on any port and send raw commands followed by repeating characters of a length specified by the user. The tester can try different payloads in an attempt to crash the app. In order to use the tool effectively the tester will need to know the raw commands for communicating with the specific service. In the following example (figure 6.1), Lookout initiates an SMTP session with a sendmail server on port 25. Lookout connects to the server and when entering the domain the client is connecting from, it will send my domain name followed by a repeating character. Since the maximum length of a domain name should be 64 characters (Section 4.5.3 of RFC 821 indicates that the maximum length of a domain name is 64 characters), I started with a 64 character domain name which was accepted by the server, as evidenced by the “pleased to meet you” response. I then increased the number of appended characters incrementally, going up from 64 until I received a failure at 1000 characters. In this instance I didn’t appear to overflow any buffers (although sendmail has been riddled with them in the past), but it’s a good example of the kind of input testing that can be done with Lookout.

Figure 6.1

```
220 whistler.somecompany.com ESMTP Sendmail 8.12.3/8.12.3; Tue, 6 Jul
2004 15:58:29 -0400 (EDT) 250 whistler.somecompany.com Hello
[192.168.x.y], pleased to meet you
helo someserver.com
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
250 whistler.somecompany.com Hello [192.168.x.y], pleased to meet you
helo someserver.com
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
250 whistler.somecompany.com Hello [192.168.x.y], pleased to meet you
helo someserver.com
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
250 whistler.somecompany.com Hello [192.168.x.y], pleased to meet you
helo someserver.com
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
250 whistler.somecompany.com Hello [192.168.x.y], pleased to meet you
helo someserver.com
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
250 whistler.somecompany.com Hello [192.168.x.y], pleased to meet you
helo someserver.com
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx 501 5.0.0 Invalid domain name
```

For extensive application buffer testing NTOMax will stress test your server for DOS and Buffer overflow weaknesses. It runs from the command line and with a few simple modifications to the script you can run a complete buffer overflow test specifying the minimum and maximum buffer size with which to hit the server (figure 7.1 shows a sample script). Only the first four parameters ([IP Address] [Port] [minimum buffer size] [maximum buffer size]) are required. The script takes the minimum buffer size parameter and injects a buffer payload of that size in place of the asterisk. It then increases the buffer size by one byte with each subsequent loop command until the maximum buffer size is reached. The results are logged in an output text file that the user specified in the run command (figure 7.2 is an example of the command syntax). In the example, metatest.txt is the script called by the program and the results of the test are logged in metaresults.txt. The tester can examine the results file looking for server time outs and the exact buffer size which crashed the web application. The tool can also be used with mail servers by including the buffer payload within the username and password.

Figure 7.1

```
host:10.20.1.11,80,40,50,1000,250,250,2,true,true,false,false
lc:GET /some*url/ HTTP/1.0
lc:POST /some * url/ HTTP/1.0
```

Figure 7.2

```
C:\apps\NTOMax20\NTOMax20>ntomax /s <metatest.txt> metaresults.txt
```

7.3 SQL Injection

“The database is the heart of most Web applications” says Mitchell Harper in “SQL Injection Attacks – Are you safe?” (Harper). Everything from credit info to usernames and passwords are stored in the database. Therefore it’s no surprise that databases often grab the attention of the malicious user community. How do you attack the database if it’s not directly accessible? Use the web application itself. Most web applications allow for user interaction, and since the web app interacts with the database, it opens a door for malicious attacks.

In SQL injection attacks SQL queries are injected into the database by way of the web application. If fields that allow user input (such as username and password) are not validated then they can be used to fool the database into giving up confidential information, dropping tables, deleting the database, modifying stored procedures and modifying or creating user accounts and permissions (Mookhey).

As with buffer overflows, the best way to eradicate SQL injection attacks is through better coding and validation. Many database developers recommend replacing all single quotes with double quotes. Until that happens it remains the security tester's task to ensure that every parameter of every script has been tested.

The simplest way to test for exposure to SQL injection hacks is to test all the input fields of a web application. Just as when testing for buffer overflows, it's important for the tester to know every possible path for data to find its way from the application to the database, including fields in cookies. The three most widely used SQL characters used to perform SQL injection are the single quote (sometimes the hex version of this character is used (SPI Labs, 2004)), semi-colon and double dash.

In a SQL query the single quote ends the string. Often the attacker will use the single quote in an input field to terminate the string and then insert a parameter that is always true, such as "Or 1 = 1", to gain access to the database (Hurlbut). The easiest way to test for this kind of injection is to test each parameter individually with a single quote and a SQL keyword leaving all other parameters intact with valid data so that there's no doubt about which parameter caused the injection. Be sure to include every input field in your test case (SPI Labs, 2004).

You can also use the single quote test in URLs. Pages built with scripting languages like ASP can sometimes be compromised with added or modified parameters. Here is an example URL:

```
http://www.afakeurl.ca//default.asp?Section=books&Lang=en
```

To test this example, we could try replacing the entire parameter "books" with a single quote and observe the results. Also, we could insert a single quote in the middle of a parameter and check for errors (Kwon).

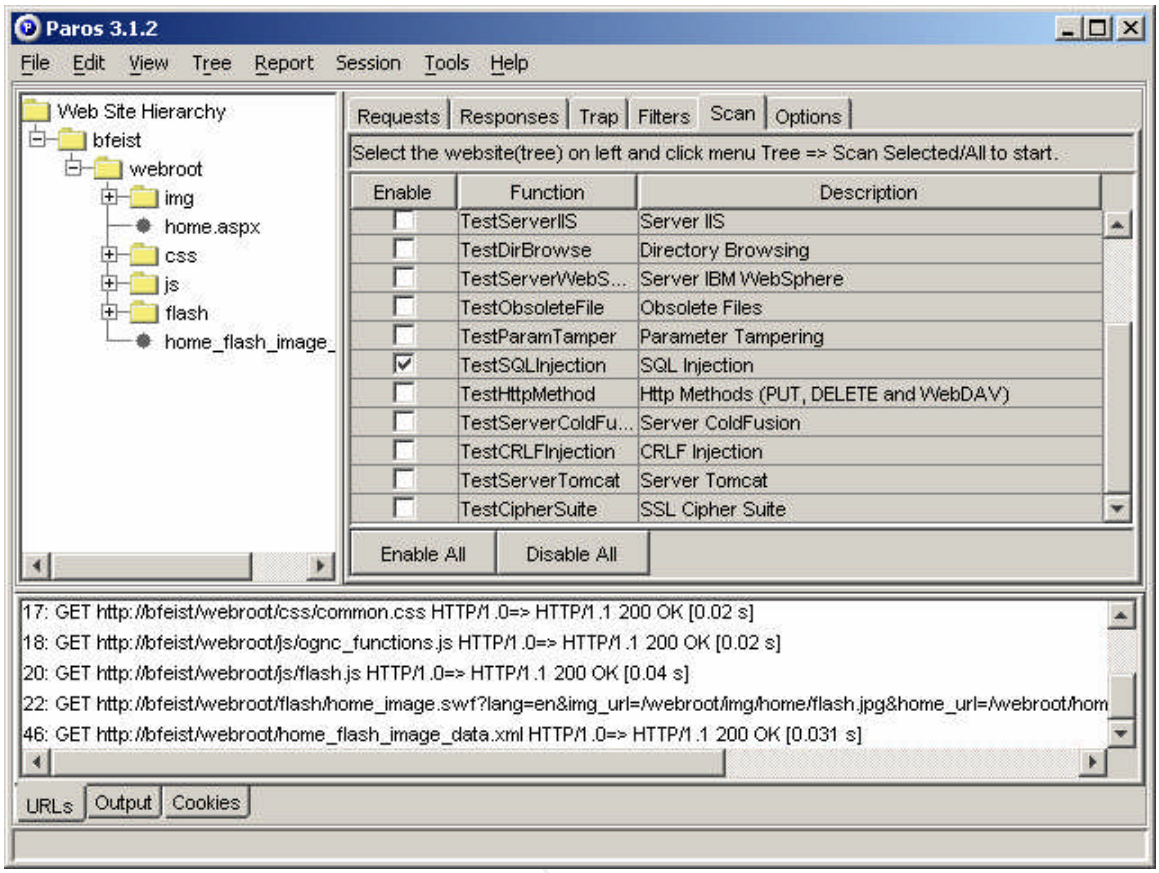
Evidence of an occurrence of a SQL injection is not always obvious. Look for any unusual database related errors, but also be aware of error messages in http headers, redirects to the home page, generic http errors, internal server error messages, 500 server error pages or anything at all that's out of the ordinary. Sometimes you'll have to view source on the page in order to view the complete error.

Testing every single parameter at any entry point into the database can be quite time consuming. Fortunately there are several tools available on the market that

can automate this process. One popular tool for testing applications for SQL injection vulnerabilities as well as a multitude of other vulnerabilities is Nessus (available at www.nessus.org). Nessus and its windows counterpart, Newt (www.tenablesecurity.com), use NASL scripts which are easily uploaded as plug-ins. Known vulnerabilities are grouped into categories and one can easily select a category with the option to scan for all vulnerabilities or just those that are not detrimental to the system you're scanning. Most of the vulnerabilities that will concern web application testers reside in the CGI abuses category. The SQL injection plug-in isn't exactly obvious but an email to Tenable Security support revealed the SQL injection plug-in is named "wpoison (nasl version)" with script ID 11139.

Another easy to use tool for SQL injection testing is Paros (available for free at www.proofsecure.com). When activated, Paros runs as a proxy server on your local machine. You then setup your browser to point to the localhost as a proxy server on the default port of 8080. Paros automatically adds any sites accessed by the browser to the explorer style left hand window as seen in Figure 10.1. You can then use the spider feature to scour the site looking for each and every page. There are 16 different vulnerabilities it will scan for, including cross-site scripting, SQL injection, buffer overflows, parameter tampering and directory browsing. After selecting the vulnerabilities to scan for, right-click on the site and select "Scan Selected Node". Upon completion, Paros opens up an HTML report detailing each issue and rating its risk level. Paros can also be used to trap http commands and test with modified parameters. Unfortunately, Paros doesn't give any detail regarding the criteria used for the scans and the documentation is sparse.

Figure 10.1



10.2 Cross-Site Scripting

In August of 2001 Jeremiah Grossman was able to break into Microsoft's Hotmail system with 3 lines of code. He was then able to access Passport IDs and credit card info. Fortunately, Jeremiah wasn't attempting to do any damage and he quickly alerted Microsoft to the vulnerability. Jeremiah didn't hack into the firewall but rather used a technique known as cross-site scripting, a form of attack that security experts were aware of years before the Hotmail incident (Acohido). Today cross-site scripting vulnerabilities account for a good portion of the Bugtraq and CERT alerts.

In a cross-site scripting attack, an innocent user is enticed to visit a vulnerable website by clicking on a URL which contains a malicious script embedded in the URL. The website with the vulnerability outputs the script, which it doesn't recognize, back to the browser where it is run. The malicious script can then, unbeknownst to the user, give the hacker access to that user's cookies. This in turn can allow the hacker to authenticate as the innocent user, or it may present a site that looks legitimate, but is actually logging passwords or credit card info. The innocent user may see an HTTP 404 file not found error and think there's something wrong with the website without giving it any further thought. What he may not realize is that he's just been the target of a cross-site scripting attack.

At the heart of the vulnerability are three main issues:

1. A good number of web sites that output to HTML unvalidated input in the URL.
2. Most client browsers are set up to automatically interpret and run scripts in web pages (CERT).
3. Uneducated users tend to be very trusting of URLs. Well-know domains are assumed to be legitimate, and the URL is rarely examined. Hackers will further thwart examination by encoding the URL in hexadecimal, or making the URL long enough to obscure the script is beyond the user's view in the address field of the browser.

These three factors combined create a dangerous situation that is easy for hackers to take advantage of (CERT).

Preventing cross-site scripting attacks involves addressing the three issues mentioned above. Developers need to be aware of the security issues when programming. Careful input validation at all entry points to your application, along with proper data escaping could eliminate the majority of XSS attacks. CERT advises developers to "restrict variables used the in the construction of pages to those characters that are explicitly allowed" (CERT). If at all possible, users should disable the browser function to automatically run scripts downloaded from websites. They should also not trust URLs found on message boards or sent in emails, which are common methods used by attackers to distribute malicious URLs (Sharma). The prevention methods for developers and programmers are similar for most vulnerabilities and it often comes down to better input validation. However, mistakes will always be made and changing the bad habits of users is a difficult task. Thus, it's crucial for the security tester to be vigilant in testing each and every entry point into the application. One unvalidated entry point can render futile all of the other input validation performed in other areas of the application. Now, let's examine some techniques for testing for XSS vulnerability.

To reiterate, the objective in testing for XSS vulnerabilities is to see if any entry point into the application can cause the application to output a script which could potentially be run by the browser. The first step is documenting every entry point into the application, which could include URL parameters, form fields, cookies, query strings and http headers. Once these entry points are documented, we can run some specific tests against them (Howard, When Output Turns Bad).

To test URL parameters, try changing a variable in the URL and hitting enter. On the resulting page (which may generate an error), view the source and search for your variable. If the new variable was included in the output page, it may be possible to inject malicious variables or script and you have a bug that needs fixing. (Pennington)

Of all the potentially dangerous characters in XSS attacks, the "<" and ">" characters are worthy of special attention as they specify the start and end of a

script. Security conscious developers will have these characters escaped. To confirm this, try changing your variable to “<somevariable>” and searching the resulting page source for “<somevariable>” (Grossman). If you find your variable between the “<” and “>” characters, you have a potential XSS vulnerability. Properly escaping the greater-than and less-than characters is one step in building a secure application.

To test for script execution try replacing your variable with a JavaScript alert statement, such as “<script>alert(‘Hello’)</script>” (which would display a message dialog containing the text, “Hello”). If your application executes the script or returns the string you have a vulnerability in your application (Pennington). URL parameters are not the only input fields to test with this technique. Every form input should also be tested to see if scripts will be accepted and executed.

Free tools such as Nessus and Paros, or their commercial counterparts, such as WebInspect from SPI Dynamics, can automate the task. Web Scarab, a project under development by the OWASP group, promises to be an excellent vulnerability testing tool (currently, it is still under development). Many experienced security testers are of the opinion that the best way to test for XSS vulnerabilities is to manually check the application with a thorough testing plan employing the techniques detailed above. As stated previously, this requires a complete knowledge of the input entry points, and a thorough understanding of how XSS vulnerabilities are exploited by malicious users.

12.1 Conclusion

As the use of web applications continues to grow we can expect the number of attacks against them to increase. The threats are real and every year millions of dollars are lost to downtime and fraud due to unsecured applications. The only way to properly ensure the security of your application is to employ thorough security testing. Software development firms must take make this a part of the development process. If you are a client, you too must ensure that the application you’ve purchased has been tested. The majority of vulnerabilities and exploits are well known, and there is no excuse for allowing these holes to exist in a professionally developed application. This paper examined a few of the threats including buffer overflows, SQL injection and cross-site scripting. While there are several other exploits you will want to familiarize yourself with, a thorough knowledge of the three we discussed will enable you to thwart many of the attacks your application will face.

13.0 Cited Works

Jaquith, Andrew. “The Security of Applications: Not All Are Created Equal.” February 2002. URL: http://www.atstake.com/research/reports/acrobat/atstake_app_unequal.pdf (20 June 2004)

Abner, Germanow, Chris Wysopal, Dr. Dan Geer, and Chris Darby. “The Injustice of Insecure Software.” February 2002. URL: http://www.atstake.com/research/reports/acrobat/atstake_injustice.pdf (20 June 2004)

Kolawa, Adam. “Preventing Web Service Security Breaches with Unit Testing.” URL: http://www.stickyminds.com/se/S6072.asp (21 June 2004)

SPI Labs Inc. “Complete Web Application Security: Building Web Application Security into Your Web Development Process.” URL: http://www.spidynamics.com/whitepapers/Webapp_Dev_Process.pdf (21 June 2004)

Cenzic. “Enabling Security in the Software Development Lifecycle.” May 2004. URL: http://www.cenzic.com/pdfs/CenzicWpEsisd.pdf (1 July 2004)

Wood, Peter. “Web Application Hacking: Exposing Your Backend.” 11 November 2003. URL: http://www.net-security.org/article.php?id=599 (1 July 2004)

Microsoft. “Microsoft Security Bulletin MS03-007.” 30 May 2003. URL: http://www.microsoft.com/technet/security/bulletin/MS03-007.mspx (1 July 2004)

Cole, Eric, Jason Fossen, Stephen Northcutt, and Hal Pomeranz. Sans Security Essentials and the CISSP 10 Domains: Internet Security Technologies. The Sans Institute, January 2004. 54

McGraw, Gary, John Viega, “Make your software behave: Learning the basics of buffer overflows.” (1 March 2000) URL: http://www-106.ibm.com/developerworks/security/library/s-overflows (1 July 2004)

Ash, Lydia. The Web Testing Companion: The Insiders Guide to Efficient and Effective Tests. Indianapolis: Wiley, 2003. 253-260

Hurlbut, Robert. “SQL Server Security: SQL Injection.” 28 September 2003. URL: http://sqljunkies.com/WebLog/rhurlbut/archive/2003/09/28/243.aspx (3 July 2004)

Howard, Michael. “Testing for Buffer Overruns.” URL: http://archive.devx.com/upload/free/Features/zones/security/articles/2000/11nov00/mh1100[1]-1.asp (2 July 2004)

Postel, Jonathon. “RFC 821, Simple Mail Transport Protocol” August 1982. URL: http://www.ietf.org/rfc/rfc0821.txt (2 July 2004)

Harper, Mitchell. “SQL Injection Attacks – Are you safe?” 17 June 2002. URL: http://www.sitepoint.com/article/794 (3 July 2004)

Mookhey, K.K., Nilesh Burghate. “Detection of SQL Injection and Cross-Site Scripting Attacks.” 17 March 2004. URL: http://www.securityfocus.com/infocus/1768 (2 July 2004)

Kwon, Regina. “Testing for Website Vulnerabilities.” 1 November 2002. URL: http://www.baselinemag.com/article2/0,1397,666906,00.asp (3 July 2004)

SPI Labs. “SQL Injection: Are your Web Applications Vulnerable” 2002. URL: http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf (3 July 2004)

Alcohido, Byron. “USA Today: Expert Hacks Hotmail in One Line of Code.” 8 August 2001. URL: http://www.usatoday.com/tech/news/2001-08-31-hotmail-security.htm (3 July 2004)

CERT. “CERT Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests” 2 February 2000. URL: http://www.cert.org/advisories/CA-2000-02.html (4 July 2004)

Sharma, Anand. “Prevent a cross-site scripting attack. First step: recognize the signs and halt an XSS intrusion.” 3 February 2004. URL: http://www-106.ibm.com/developerworks/library/wa-secsxx/ (3 July 2004)

Howard, Michael. “When Output Turns Bad: Cross-Site Scripting Explained.” 15 July 2002. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure07152002.asp (4 July 2004)

CGI Security. “The Cross Site Scripting Faq.” May 2002. URL: http://www.cgisecurity.com/articles/xss-faq.shtml#whatis (4 July 2004)

Pennington, Bill. “Penetration Testing: Re: Cross Site Scripting Vulnerabilities – XSS.” 6 August 2002. URL: http://seclists.org/lists/pen-test/2002/Aug/0010.html (5 July 2002)

Grossman, Jeremiah. “Penetration Testing: Re: Cross Site Scripting Vulnerabilities – XSS.” 6 August 2002. URL: http://seclists.org/lists/pen-test/2002/Aug/0010.html (5 July 2002)