SANS | GIAC CERTIFICATIONS

WHITE PAPER

Detecting Attacks on Web Applications from Log Files

Roger Meyer

Detecting Attacks on Web Applications from Log Files

GCIA Gold Certification

Author: Roger Meyer

Adviser: Carlos Cid

Accepted: 26 January 2008

Outline

1 Abstract.....3
2 Introduction.....4
3 Attacks on Web Applications.....5
3.1 Web server log files.....6
3.2 Primer on HTTP.....8
3.2.1 HTTP Evasion Techniques.....12
3.3 Regular Expressions (Regex).....14
4 Detecting Attacks.....15
4.1 Rule-based Detection (static rules).....20
4.1.1 Negative Security Model.....20
4.1.2 Positive Security Model.....21
4.2 Anomaly-based Detection (dynamic rules).....21
4.3 Detecting the OWASP Top Ten 2007.....22
4.3.1 A1 - Cross Site Scripting (XSS).....22
4.3.2 A2 - Injection Flaws.....26
4.3.3 A3 - Malicious File Execution.....32
4.3.4 A4 - Insecure Direct Object Reference.....33
4.3.5 A5 - Cross Site Request Forgery (CSRF).....35
4.3.6 A6 - Information Leakage and Improper Error Handling. 37
4.3.7 A7 - Broken Authentication and Session Management....38
4.3.8 A8 - Insecure Cryptographic Storage.....39
4.3.9 A9 - Insecure Communications.....40
4.3.10 A10 - Failure to Restrict URL Access.....41
5 Conclusion.....42
6 References.....42

1 Abstract

Web traffic (Hypertext Transfer Protocol, HTTP) has overtaken P2P traffic and continues to grow. [Ellacoya, 2007] Web site hacks are on the rise and pose a greater threat than the broad-based network attacks as they threaten to steal critical customer, employee, and business partner information stored in applications and databases linked to the Web. [Greenemeier, 2006]

The increasing shift towards web applications opens new attack vectors. Traditional protection mechanisms like firewalls were not designed to protect web applications and thus do not provide adequate defense. Current attacks cannot be thwarted by just blocking ports 80 (HTTP) and 443 (HTTPS).

Preventive measures (like Web Application Firewall rules) are not always possible. Reactive methods – to detect what happened previously – are usually easier but have the disadvantage of always being behind the actual event.

This paper explains how to detect the most critical web application security flaws. Web application log files allow a detailed analysis of a users actions. Log files have its limits, though. Web server log files contain only a fraction of the full HTTP request and response. Knowing those limits, the majority of attacks can be recognized and acted upon to prevent further exploitation.

2 Introduction

Internet usage and online applications are experiencing spectacular growth. Worldwide, there are over a billion Internet users at present. A big reason for the success of the Internet is the simplicity and that you can access the applications from anywhere. This growth in popularity has not gone unnoticed by the criminal element – the simplicity of the HTTP protocol makes it easy to steal and spoof identity. The business liability associated with protecting online information has increased significantly and this is an issue that must be addressed.

The SANS Top-20 Internet Security Attack Targets (2007 Annual Update) [SANS Top-20, 2007] is a consensus list of vulnerabilities that require immediate remediation. According to this list the number one targeted server-side vulnerability are Web Applications. Some examples of Web Applications are Content Management Systems (CMS), Wikis, Portals, Bulletin Boards, Shops, Banking Systems and discussion forums. There are plenty of online resources which state Web Applications are amongst the most attacked targets ([SC Magazine, 2007] and [IT Week, 2006]).

This has made detecting and preventing these activities a top priority for every major company. This paper addresses the detection of attacks on web applications by analyzing log files from web servers (like Apache and IIS). Why should you bother analyzing log files instead of using a network intrusion detection system? There might be several reasons for this:

- The HTTP traffic may be SSL encrypted (HTTPS);
- There may be no NIDS (hard to deploy; another zone of attack);

- High traffic load makes it difficult to analyze network traffic (in real time);
- NIDS are designed to work on the TCP/IP level, and thus they may not be as effective on the HTTP layer;
- IDS evasion techniques (HTTP, encoding, fragmenting, ...).

Analyzing web traffic out of log files has some advantages and disadvantages over analyzing traffic from the network. Those differences will be explored in the chapter '4. Detecting Attacks'.

3 Attacks on Web Applications

Attacks on web applications are on a constant change. A report from Fortify Software Inc. [Fortify, 2006] outlines four trends:

- Bots are being used in more than half the attacks against web applications;
- Attackers are finding flawed web applications using Google and other search tools;
- Directed attacks are growing more sophisticated; and
- Attackers operating from bases around the world are getting better at covering their tracks.

Similar trends have been found by [KYE: Web Application Threats, 2007]. While most automated attacks targeted well-known applications like PHPBB, Mambo, AWStats, etc., web application security flaws can be categorized and rated. The OWASP Top Ten

Project [OWASP Top Ten Project, 2007] lists the 10 most critical web application security flaws. The OWASP Foundation is a not-for-profit organization which provides information about application security. The primary aim of the OWASP Top 10 is to educate developers, designers, architects and organizations about the consequences of the most common web application security vulnerabilities. In the chapter '4.3 Detecting the OWASP Top Ten' we will go into detail on how to detect these attacks.

The 2007 update brought up some new vulnerabilities like Malicious File Execution and Cross Site Request Forgery (CSRF). The biggest jump forward was made by Cross Site Scripting (XSS) though. Today, XSS is generally believed to be one of the most common application layer hacking techniques.

Attackers are targeting web applications with different goals. Every flaw has its own consequence. Attacks like XSS (A1) and CSRF (A5) target the applications' users, while all the other (Top 10) attacks target the web application itself. Taking advantage of XSS vulnerabilities allows the attacker to insert active code into a user's browser and executing it in the context of the current application. Exploiting injection flaws for example usually targets directly the application by inserting malicious code into the backend or by reading unauthorized data from the database.

3.1 Web server log files

Standard web servers like Apache and IIS generate logging messages by default in the Common Log Format (CLF) specification. The CLF log file contains a separate line for each HTTP request. A line is composed of several tokens separated by spaces:

host ident authuser date request status bytes

If a token does not have a value, then it is represented by a hyphen (-). Tokens have these meanings:

- host: The fully qualified domain name of the client, or its IP address.
- ident: If the IdentityCheck directive is enabled and the client machine runs identd, then this is the identity information reported by the client.
- authuser: If the requested URL required a successful Basic HTTP authentication, then the user name is the value of this token.
- date: The date and time of the request.
- request: The request line from the client, enclosed in double quotes (").
- status: The three-digit HTTP status code returned to the client.
- bytes: The number of bytes in the object returned to the client, excluding all HTTP headers.

A request may contain additional data like the referer and the user agent string. Let us consider an example of log entry (in the Combined Log Format [Apache Combined Log Format, 2007]):

```
127.0.0.1 - frank [10/Oct/2007:13:55:36 -0700]
"GET /index.html HTTP/1.0" 200 2326
"http://www.example.com/links.html" "Mozilla/4.0 (compatible; MSIE
7.0; Windows NT 5.1; .NET CLR 1.1.4322)"
```

127.0.0.1 : the IP address of the client

- : The "hyphen" in the output indicates that the requested piece of information is not available. In this case, the information that is not available is the RFC 1413 identity of the client determined by identd on the clients machine.

frank : This is the userid of the person requesting the document as determined by HTTP authentication.

[10/Oct/2007:13:55:36 -0700] : The time that the server finished processing the request.

"GET /index.html HTTP/1.0" : The request line from the client is given in double quotes.

200 : This is the status code that the server sends back to the client.

2326 : This entry indicates the size of the object returned to the client, not including the response headers.

"http://www.example.com/links.html" : The "Referer" (sic) HTTP request header.

"Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322)" : The User-Agent HTTP request header.

3.2 Primer on HTTP

HTTP (short for Hyper Text Transfer Protocol) is the language that web servers and browsers speak. Internet standards are proposed, discussed and, eventually, specified in documents known as RFCs (Request For Comments) and HTTP is no exception: its latest incarnation, Rev. 1.1, is described in RFC 2616 [Fielding

et al., 1999].

HTTP is a request/response system. Only the client is allowed to make requests. It asks the web server to deliver a certain page. If the file cannot be located, an error response is sent instead. Such a location on a web server is described with Universal Resource Locators or URLs:

scheme:	//host	[:port]	/path/to/resource	[?query]
Protocol identifier, e.g. http, https, ftp	FQDN of the host: e.g www.example.com	Optional port number of the service: e.g 80	Location of the resource, as known to the server	Optional context information

In a HTTP request, the client contacts the server over the network, and sends a properly formatted request describing what resource it wants. This is packaged in chunks called the request header and the request body. The server parses the request, if possible carries it out, and reacts by returning whether the operation was successful, any meta-information regarding the result data (if any), and the content that was retrieved as result of the transaction. This is formatted and packaged in chunks known as the server status, the response headers and the response body.

Let us have a look at what the client and server actually tell each other. The most basic request is a typical GET transaction, where a web browser asks for a HTML page to be sent.

The client	
GET /index.html HTTP/1.1	Request type (GET), path, protocol & version.
Host: www.example.com	The host this request is for.
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.8) Gecko/20071004 Iceweasel/2.0.0.8 (Debian-2.0.0.8-1)	Identifier of the client.
Accept: text/xml,application/xml,application/xhtml +xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5 Accept-Language: en-us,en;q=0.5 Accept-Encoding: gzip,deflate Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7	Data types and encoding this client can handle (MIME types).
Keep-Alive: 300	Keep the connection open for further requests.
Connection: keep-alive	Don't close the connection.

The server	
HTTP/1.x 200 OK	The result status: numeric and description.
Date: Fri, 14 Dec 2007 10:25:51 GMT	Time stamp.
Server: Apache	Identifier of the server software.
Content-Type: text/html; charset=ISO-8859-1	MIME formatted information on type (html) of the result.
	An empty line separates the response headers from the response body.
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"> ...	Now the document (HTML as expected) is sent.

The GET request fulfills the basic need of serving a URL to a client. The POST request, for instance, allows the client to send data to the server for further processing, and is the foundation upon which fill-out web forms are built. All these requests, however, follow the general schema outlined above, give or take a few MIME headers.

3.2.1 HTTP Evasion Techniques

There are different types of evasions in different places of the HTTP protocol. They can occur in the request URI portion of the HTTP protocol, other parts of the HTTP header, the HTTP body, etc. Evasion types can be protocol decoding evasions, simple obfuscation techniques, or more advanced evasions, like inserting additional characters to deceive the IDS system. Evasions are particularly effective against the URL and the URL parameters. We will place our focus on those two types as they are both visible in the web server log file.

The first evasions were as simple as adding multiple slashes to directories and other path traversal attacks. The following URI have all the same meaning:

```
/admin/index.html
//admin/index.html
/admin/./index.html
/admin/../admin/index.html
/admin/./../admin/index.html
```

There could be infinite combinations of those evasions. The goal of these techniques is to evade the detection by an IDS but still get executed by the web application. Here are a few more normalisation methods [WASC, 2006]:

1. URL-decoding (e.g. %XX)
2. Null byte string termination
3. Self-referencing paths (i.e. use of /./ and encoded equivalents)
4. Path back-references (i.e. use of /../ and encoded

equivalents)

5. Mixed case

6. Excessive use of whitespace

7. Comment removal (e.g. convert DELETE/**/FROM to DELETE FROM)

8. Conversion of (Windows-supported) backslash characters into forward slash characters.

9. Conversion of IIS-specific Unicode encoding (%uXXYY)

10. Decode HTML entities (e.g. c, ", ª)

11. Escaped characters (e.g. \t, \001, \xAA, \uAABB)

A very popular evasion technique is to obfuscate the URL and its parameter by using different encoding schemes. According to [Roelker, 2003], there are only two RFC standards for encoding a request URI: hex encoding and UTF-8 Unicode encoding. Both methods are encoded using the '%' character to escape a one encoded byte. There are other encoding schemes but these are all server specific and non-RFC compliant.

Hex Encoding

The hex encoding is the simplest way of encoding a URL. It consists of escaping a hexadecimal byte value for the encoded character with a '%'. To encode the letter C, which has an ASCII hexadecimal value of 0x43, the encoding would look like this:

- %43 = 'C'

How would such a request look like in a log file? Let us make

another request (/index.html). This is how it would look like:

- GET /index.html HTTP/1.1

Now we encode the URI with hex values:

- GET /%69%6E%64%65%78%2E%68%74%6D%6C HTTP/1.1

Further encodings are the double percent hex encoding, the double nibble hex encoding, the first nibble hex encoding and the second nibble hex encoding. These will not be supported by all web server though.

For other encoding variants like UTF-8 encoding, please see [Roelker, 2003] for an overview.

3.3 Regular Expressions (Regex)

Regular expressions enable a powerful, flexible, and efficient text processing. Regular expressions allow you, with a general pattern notation almost like a mini programming language, to describe and parse text. This powerful pattern language and the patterns themselves are called regular expressions.

Regular expressions are available in many types of tools, but their power is most fully exposed when available as part of a programming language. The goal of a regular expression is to match a certain expression within a lump of text.

Example regular expression:

```
/(java)?script/i
```

A regular expression pattern is usually enclosed within slashes ('/'). This regex finds all occurrences of 'script' or

'javascript'. Modifiers, usually appended after the closing slash, allow to set certain options like case-insensitive pattern matching (the 'i' modifier).

For more information about regular expressions, see [Friedl, 1997].

4 Detecting Attacks

Web applications are running on the OSI [OSI, 1994] layer 7 - the application layer. To detect attacks against web applications, the detection mechanisms have to be application layer aware and see the relevant traffic.

Attacks can be detected at different zones and devices in the network infrastructure. Each place has a different view of the traffic and has its advantages and disadvantages. We are now going to explore each of these places in the network.

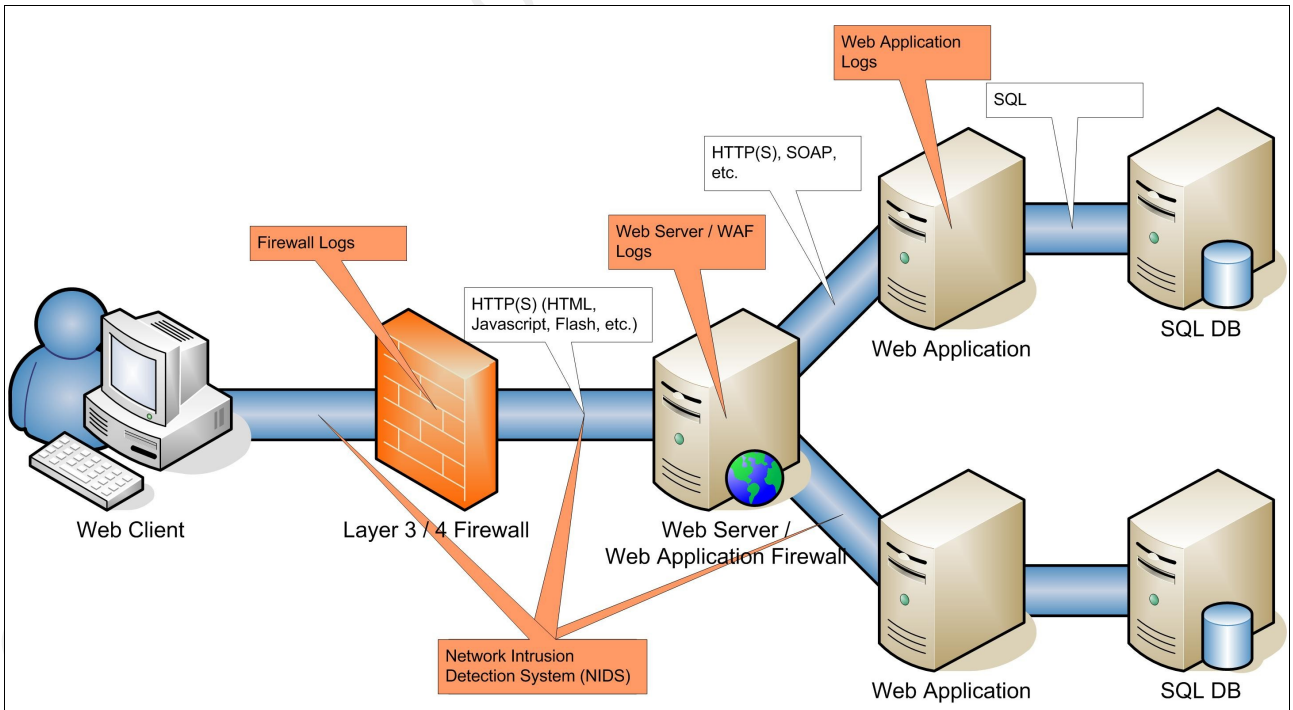

Illustration 1: Detecting Attacks in a Network

Layer 3/4 Firewall

A traditional (stateful and non-stateful) firewall is working on OSI layers 3 (Network Layer) and 4 (Transport Layer). The firewall analyzes traffic based on the common protocols like TCP, UDP and ICMP and their corresponding ports or types/codes. Firewalls can detect anomalies in the protocols they are aware of like fragmented IP traffic, but they are generally not the best place to detect attacks on the application layer. Firewall log files usually do not contain application layer data like HTTP data, only layer 3 and 4 information, so they are not very helpful in detecting what is going on higher layers.

Application layer firewall / web application firewall (WAF)

Web application firewalls are designed to work on the OSI layer 7 (the application layer). They are fully aware of application layer protocols such as HTTP(S) and SOAP and can analyze those requests in great detail. Compared to a layer 3/4 firewall, rules can be defined to allow/disallow certain HTTP requests like POST, PUSH, OPTIONS, etc., set limits in file transfer size or URL parameter argument length. WAF log files contain as much information as those from a web server plus the policy decisions of the filter rules (e.g. HTTP request blocked; file transfer size limit reached, etc.). A WAF provides a wealth of information for filtering and detection purposes and is thus a good place for the detection of attacks.

Web server

The web server is the end device of an HTTP request. Standard web servers like Apache and IIS are logging by default in the Common Log Format (CLF) specification. See chapter '3.1 Web server log files' for a detailed description of the CLF format.

Web server logs do not contain any data sent in the HTTP header, like POST parameters. The HTTP header can contain valuable data, as most forms and their parameters are submitted by POST requests. This comes as a big deficiency for web server log files.

A web server can also act as a web application firewall (see previous section). The Apache module mod_security [Breach Security, 2007] allows for example to set detailed rules on HTTP data, exactly like on a WAF. Of course, such rules will have access to the full HTTP header information, including POST parameters and can enable additional logging of such parameters (successful or denied access).

Web application

A web application consists of a framework (PHP, ASP, J2EE, etc.) which implements the business logic. It is considered to be best practice to perform input/output validation in this tier. A strong input validation policy will detect malformed and malicious input and can log security related information to a log file. The application has access to the full user trail - each step a user takes (logging in, making a transfer, logging out, etc.). A comprehensive logging at the application tier enables the detection of misuse and fraud and allows a full reconstruction of a user's steps.

Network Intrusion Detection System (NIDS)

A Network Intrusion Detection System (NIDS) is placed in the network infrastructure where it can see the traffic to and from the web application. It usually resides on its own machine and analyzes the web traffic without touching the firewalls and the application itself. The NIDS has a number of disadvantages over a WAF:

- If the HTTP traffic is SSL encrypted (HTTPS), the NIDS might not decrypt the traffic;

- A high traffic load can make it difficult to analyze network traffic in real time;

- NIDS are designed to work on the TCP/IP level (OSI layer 3/4), and thus may not be as effective on the HTTP layer;

- Attackers might use IDS evasion techniques (HTTP, encoding, fragmenting, etc.) which the IDS is not aware of.

Snort, the most powerful open source IDS, has over 800 rules (community rules, released 2007-04-27) for detecting malicious web traffic (over 400 for PHP alone). With the help of preprocessors like frag3 (IP defragmentation), stream4 (stateful inspection/stream reassembly) and http_inspect (normalize and detect HTTP traffic and protocol anomalies) snort tries to assemble packets and avoid IDS evasion techniques. These hurdles have to be overcome before anything can be detected.

Analyzing logs vs. full traffic

Log files contain only a partial set of the full traffic going over the network. Depending on the application which writes the logs, this can be a full audit trail or just some data. The following table shows the most significant differences between analyzing log files and full traffic.

	Advantages	Disadvantages
Log files	data is easily available to be analyzed (in files)	logs usually contain only a fraction of the full data (e.g. missing HTTP POST parameters)
Full traffic	all the information can be analyzed	- data has to be captured first - data might have to be assembled, defragmented, normalized, etc. (IP packets, IP fragments) - it might be difficult to capture data (encrypted traffic, high traffic load, etc.)

The biggest benefit of log files is the relative simple availability and analysis of their content. Web servers like Apache have logging enabled by default. Applications usually do some logging to ensure the traceability of their actions. While full traffic provides additional information, its acquisition and processing costs usually outweigh their benefit. The collection of network traffic requires a) visibility to packets and b) usually additional hardware. Watching traffic can be achieved with hubs, SPAN ports, taps or inline devices. All these devices have to be purchased, installed and supported. Once the data is collected, it has to be processed into a suitable format so that it can be analyzed. Only now the collected network traffic is in the same form as the log files and is ready for being analyzed. In the end,

log files provide an easy available and easy to process possibility to do security monitoring.

The next section discusses the two detection strategies - static rules and dynamic rules.

4.1 Rule-based Detection (static rules)

Attacks can be detected by two different strategies: rule-based and anomaly-based. The rule-based strategy defines static rules which have to be defined before the analysis can be made. Those can be simple rules like the detection of certain characters or more complex rules like session fixation attacks.

Anomaly-rules consist of dynamic rules, they will be discussed in the next section.

Static rules are defined once and stay the same during the detection phase. They have to be defined and specifically crafted for each application. Static rules make most sense for pre-known values like certain input characters, a fixed length of a parameter or an upper limit of a transfer amount.

Static rules can be further divided into two detection models: the negative and the positive security model.

4.1.1 Negative Security Model

The negative security model, or the blacklist approach, has a default policy of allow everything. This means that everything is allowed to pass, or everything is considered "normal", accepted traffic. The policy (the blacklist, or the rule-base) defines what is not allowed or in IDS terms what will be flagged as an attack.

This model is usually considered the easier one as it is

easier to implement, it is however not a good approach security-wise. The biggest disadvantage is that the detection will only be as good as the policy. It has to be adopted to new findings and updated to recognize new attack vectors. One of the positive points is that it yields very little false-positives as the rules will usually look for specific, well known attack strings or behaviour.

The Snort rule set is an example of a blacklist approach.

4.1.2 Positive Security Model

The positive security model is the opposite of the negative security model. The default policy here is deny all, the policy will then define what is allowed. The policy, or the whitelist, defines what is considered "normal", good traffic. This whitelist can be defined automatically in a learning phase or be manually defined. It is important that the learning phase consists of legitimate traffic, as everything else will be considered as malicious.

This model is the preferred way from a security standpoint. False negatives can be reduced to a minimum, while false positives help to improve the whitelist.

Firewalls are usually configured this way. The default policy will be deny. For every server/service there has to be a new whitelist entry for this specific machine and port.

4.2 Anomaly-based Detection (dynamic rules)

Anomaly-rules consist of dynamic rules. As the name implies, those rules are not static nor are they manually defined. Instead, the rules are defined through a learning phase. In this learning

phase, good traffic is recorded as “normal“. It is of greatest importance, that this traffic is “clean“ and free of attacks, as this will be used as our baseline. Usually, a simulation environment is used for such kind of tests. The goal of a learning phase is to define how “normal“, accepted traffic looks like to eventually flag anomalous traffic which does not look like “normal“ and raise an alarm. Deviations from this ruleset will be flagged as anomalous traffic.

4.3 Detecting the OWASP Top Ten 2007

This chapter describes how the OWASP Top Ten 2007 can be detected. The detection only applies to data stored in web application log files. This reduced dataset limits the detection to the available data, which of course simplifies the analysis but - more importantly - narrows the detection. Each vulnerability will be briefly described and explained how it can be detected with a sample regular expression.

4.3.1 A1 - Cross Site Scripting (XSS)

XSS flaws are currently the No. 1 flaw on Mitre's Common Vulnerabilities and Exposures (CVE) [MITRE, 2007] site - a considerable growth from 12 months ago. XSS vulnerabilities comprised one in five of all CVE-reported bugs in 2006 [Christey, 2007].

Cross Site Scripting attacks work by embedding script tags in URLs/HTTP requests and enticing unsuspecting users to click on them, ensuring that the malicious javascript gets executed on the victim's machine. These attacks leverage the trust between the user and the server and the fact that there is no input/output validation on the server to reject javascript or other active code

characters. Simple attacks contain HTML tags like <h1> or <script>. An often used example is <script>alert('XSS')</script>. A simple way is to detect such HTML tags. The following regular expression recognizes tags:

```
/((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)/ix
```

Explanation:

((\%3C)|<)	check for opening angle bracket or hex equivalent ('3C')
((\%2F)|\/)*	the forward slash for a closing tag or its hex equivalent ('2F')
[a-z0-9\%]+	check for alphanumeric string inside the tag, or hex representation of these (the additional percent character)
((\%3E)|>)	check for closing angle bracket or hex equivalent ('3E')

The modifiers 'i' and 'x' (at the end of the regex, after the closing slash '/') are used in order to match without case sensitivity and to ignore whitespaces, respectively.

This will of course detect any XML/HTML tag, including any legitimate user input as usually happens in an Internet forum, and may lead to many false positives.

This was a nice regex to start with. Unfortunately, javascript can be included in many more places and tags. One popular place is the 'img' tag, where users might be able to set their own image file name (an avatar for example). The 'src' parameter of the 'img' tag will work well as a javascript vector. There are many more HTML tags, where javascript can be included.

Examples:

- <img src="javascript:alert('img src XSS')">image 1

- <img src="" onerror="alert('img onerror XSS')">image 2

- <img src=&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#97&#108&#101&#114&#116&#40&#39&#88&#83&#83&#39&#41>image 3 # javascript:alert('XSS')

- link

- <body onload="alert(String.fromCharCode(88,83,83))">

A basic approach to detect above attacks would be to look for the tag name, e.g. img:

/(((\%3C)|<)((\%69)|i|(\%49))((\%6D)|m|(\%4D))((\%67)|g|(\%47))[^\n]+((\%3E)|>)/I

Explanation:

(\%3C)|<)	opening angled bracket or hex equivalent ('3C')
(\%69)|i|(\%49)) ((\%6D)|m|(\%4D)) ((\%67)|g|(\%47)	the letters 'img' in varying combinations of ASCII, or upper or lower case hex equivalents
[^\n]+	any character other than a new line following the '<img'
(\%3E)|>)	closing angled bracket or hex equivalent ('3E')

A more successful approach would be to look for all the possible expressions which may trigger javascript or other active code. Here is a list of possible script inclusions:

HTML tags:

- javascript, vbscript, expression, applet, meta, xml, blink, link, style, script, embed, object, iframe, frame, frameset, ilayer, layer, bgsound, title, base

Javascript event handlers (excerpt):

- onabort, onactivate, onafterprint, onafterupdate, onsubmit, onunload, ...

Let us write a regex to detect some of those keywords:

/(javascript|vbscript|expression|applet|script|embed|object|iframe|frame|frameset)/i

Explanation:

(javascript|vbscript|...)	Each keyword inside the parantheses will be matched. The pipe character ('|') denotes an OR.

But even looking for all of the above expressions is no guarantee to find all XSS injections. The context of the code injection is the crux. If the injection takes place inside a javascript code part, there is no need for a tag or one of the above expressions, one can usually just insert javascript code. Those kind of XSS injections are very hard to detect.

Detecting real world XSS attacks

For a real world analysis we need logs from a web server. The Honeynet Project used to provide regular Honeynet Challenges to analyze attacks and share their findings. The challenge in Scan 31 was to analyze web server log files looking for signs of abuse [Honeynet Project - Scan 31, 2004].

The log files from Scan 31 can be downloaded from the Honeynet Project website. Analyzing the apache access_log file with the above regular expressions yields interesting findings. Here are two example requests:

```
    217.160.165.173 - - [12/Mar/2004:22:31:12 -0500]
"GET /foo.jsp?<SCRIPT>foo</SCRIPT>.jsp HTTP/1.1" 200 578 "-"
"Mozilla/4.75 [en] (X11, U; Nessus)"

    217.160.165.173 - - [12/Mar/2004:22:37:17 -0500] "GET /cgi-
bin/cvslog.cgi?file=<SCRIPT>window.alert</SCRIPT> HTTP/1.1" 403
302 "-" "Mozilla/4.75 [en] (X11, U; Nessus)"

    These are two requests of a Nessus scan, trying to find
scripts which are vulnerable to XSS. According to the HTTP
status code, in the first request the web sever responded with a
200 OK, which means that foo.jsp was there and served a page. We
don't know if this page is vulnerable, though. We would have to
try this request manually to find out. The second request
(cvslog.cgi) was not successful, the server responded with a 403
Forbidden response, which means that the web server denied the
access.
```

4.3.2 A2 - Injection Flaws

Code injection can be any type of code like SQL, LDAP, XPath, XSLT, HTML, XML and OS command injection. XSS (see A1) is in fact a subset of HTML injection. Here, we are focusing on the most prevalent injection, the SQL injection. For SQL injections to work, the attacker has to jump out of the original SQL statement. This is usually done by the single-quote (') or the double-dash (--). The single-quote acts as a delimiter for an SQL query; the double-dash is the comment character in Oracle and MS SQL.

```
/(\')|(\%27)|(\-\-)|(#)|(\%23)/ix
```

Explanation:

(\')|(\%27)	the single quote and its URL encoded version
(\-\-)	the double-dash
(#)|(\%23)	the pound sign and its URL encoded version

We first detect either the hex equivalent of the single-quote, the single-quote itself or the presence of the double-dash. These are SQL characters for MS SQL Server and Oracle, which denote the beginning of a comment, and everything that follows is ignored. Additionally, if you're using MySQL, you need to check for the presence of the '#' or its hex-equivalent. Note that we do not need to check for the hex-equivalent of the double-dash, because it is not an HTML meta-character and will not be encoded by the browser. Also, if an attacker tries to manually modify the double-dash to its hex value of %2D (using a proxy like Achilles), the SQL injection attack fails.

The previous regex fails when there is neither a single-quote nor a double-dash in the attack pattern. SQL injection is possible even without the single-quote [Anley, 2002]. Let us take this example SQL statement:

select value1, numeric_value2 from table1 where numeric_value2=**user_input**

Here, an attacker may execute an additional SQL query, by supplying an input like:

7; select * from users

The above detection pattern could be easily extended with the semi-colon (';'). Unfortunately, the semi-colon is a common character in URLs. Example:

POST /login.jsp;jsessionid=HLQxtLQ13

The detection could be narrowed down by detecting the equals character ('='), and only look for semi-colons in URL parameters. Malicious user input could look like this:

POST /login.jsp?username=bill&password=1234;select * from users

Let us modify our previous regex to detect this kind of attack.

/((\%3D)|(=))[^\n]*((\%27)|(\')|(\-\-)|(\%3B)|(;))/i

Explanation:

((\%3D)|(=))	The equals sign ('=') or its URL encoded version
[^\n]*	zero or more non-newline characters
((\%27)|(\')|(\-\-)|(\%3B)|(;))	The single-quote, the double-dash or the semi-colon or their URL encoded versions

This pattern looks for the equals sign followed by zero or more non-newline characters and then a single-quote, a double-dash or a semi-colon.

Another typical attack vector is by using the SQL keyword 'or'. An example SQL attack might look like 1' or '2'='2. Of course, there are infinite variations of this like 1' or 1<2--. The only constant part is the single-quote followed by the word 'or'. Let us try to write a regular expression to detect this attack:

/\w*((\%27)|(\'))(\s|\+|\%20)*((\%6F)|o|(\%4F))((\%72)|r|(\%52))/ix

Explanation:

\w*	zero or more alphanumeric or underscore characters
(\s|\+|\%20)*	zero, one or more white spaces or their HTTP encoded equivalents
((\%27)|(\'))	the single-quote or its hex equivalent
((\%6F)|o|(\%4F)) ((\%72)|r|(\%52))	the word 'or' with combinations of its upper and lower case hex equivalents

Beside the 'or' keyword, there is another SQL keyword which is commonly used in attacks – the UNION keyword. UNION is used to combine the result from multiple SELECT statements into a single result set. Attackers can use it to combine a select statement given by the application with an attacker specified select statement. This allows an attacker read tables different from the one specified in the statement by the application. Let us expand the previous regex with some more interesting SQL keywords.

/((\%27)|(\'))(select|union|insert|update|delete|replace|truncate)/ix

Explanation:

(\%27)|(\')	the single-quote and its hex equivalent
(select|union| insert|update| delete|replace| truncate)	the SQL keywords

If the backend database runs on MS SQL, there are some especially dangerous stored procedures, which the attacker will try to exploit. These procedures start with the letters 'sp' or 'xp' respectively. One of the infamous procedures is the 'xp_cmdshell' extended procedure, which allows the execution of Windows shell commands through the SQL Server. The access rights with which these commands will be executed are those of the

account with which SQL Server is running - usually Local System.

```
/exec(\s|\+)+(s|x)p\w+/ix
```

Explanation:

exec	the keyword required to run the stored or extended procedure
(\s|\+)+	one or more whitespaces or their HTTP encoded equivalents
(s|x)p	the letters 'sp' or 'xp' to identify stored or extended procedures respectively
\w+	one or more alphanumeric or underscore characters to complete the name of the procedure

Other injection flaws like OS command injections are a bit more difficult to detect as there is no predefined pattern available. Although there are some special characters like the pipe symbol ('|') which is rarely used in an URL:

```
/(\||%00|system\(|eval\(|`|\\)/i
```

Explanation:

|	The pipe symbol: used in commands to "pipe" the stdout of one program into stdin of another. This can be abused to execute another command.
%00	The NUL character (decimal and hexadecimal 0) is used in C/C++ based programs as a string delimiter (the last element in a char array). It can sometimes be abused to trick those programs to treat this character as the last char and ignore any further characters. Other languages like Perl or PHP will happily read past the NUL character and execute the code.
system\(	System() is a function in programming languages like Perl and PHP which executes an external program and displays the output.
eval(	Eval() is a function in PHP, Perl and other languages which evaluates a string as PHP/Perl/... code.

`	The backtick operator is similar to the system() function in that it executes an external program.
\\	The backlash is used for escaping characters. If the escaping backlash can be escaped, attackers can jump out of the escaped sequence.

Detecting real world injection attacks

We are analyzing again the Scan 31 from the Honeynet Project Challenge [Honeynet Project - Scan 31, 2004]. Here are two example requests, detected with the above regular expressions:

```
81.171.1.165 - - [13/Mar/2004:10:46:43 -0500] "HEAD
http://www.sweetgeorgia.com/cgi-bin/af.cgi?_browser_out=|
echo;id;exit| HTTP/1.0" 200 0 "http://www.sweetgeorgia.com/cgi-
bin/af.cgi?_browser_out=|echo;id;exit|" "Mozilla/4.0
(compatible; MSIE 6.0; Windows NT 5.1)"
```

```
66.138.147.49 - - [13/Mar/2004:13:33:06 -0500] "GET
http://login.korea.yahoo.com/config/login?.redir_from=PROFILES?.
&login=&.tries=1&.src=jpg&.last=&promo=&.intl=us&.bypass=&.partn
er=&.chkP=Y&.done=http://jpager.yahoo.com/jpager/pager2.shtml&lo
gin=blood`1234567890&passwd=password HTTP/1.0" 200 566 "-" "-"
```

The first request is trying to execute OS commands. The variable '_browser_out' contains a pipe symbol, followed by Unix system commands ('|echo;id;exit|').

The second request is calling a login function. In its login name (parameter 'login') is a back tick symbol ('blood`1234567890'). This might be a simple brute force attack or a test how the application handles the back tick symbol.

4.3.3 A3 - Malicious File Execution

Applications which allow the user to provide a filename, or part of a filename are often vulnerable if the input is not carefully validated. Allowing the attacker to manipulate the filename may cause the application to execute a system program or an external URL.

In the past PHP has very often been criticized for the possibility to allow URLs in include and require statements. It is the cause for the most dangerous vulnerabilities in PHP applications: the often called remote URL include vulnerabilities. The following include statement will include and execute everything POSTed to the server:

```
include "php://input";
```

The following include statement will include and execute the base64 encoded payload. Here this is just phpinfo():

```
include "data:;base64,PD9waHAgcGhwaW5mbygpOz8+";
```

Let us first try to catch the remote file inclusions. If a file is referenced on a remote machine, there will be a protocol and a path, like http://www.example.com/bad.inc. We can try to detect these protocol specifiers:

```
/(https?|ftp|php|data):/i
```

Explanation:

(https?|ftp|php|data)	the protocols http(s), ftp, php and data followed by the colon

Applications which allow file uploads have the additional

risk of executable code being placed into the application. However, the log file will only contain the script, where the file will be uploaded to (e.g. POST /upload.php HTTP/1.1). The content of the file is only visible in the application.

4.3.4 A4 - Insecure Direct Object Reference

Applications often expose internal objects, making them accessible via parameters. When those objects are exposed, the attacker may manipulate unauthorized objects, if proper access controls are not in place.

Internal Objects might include:

- Files or Directories
- URLs
- Database keys, such as acct_no, group_id etc.
- Other database object names such as table name

Files can be recognized by their name or ending. Especially dangerous files might be /etc/passwd, /etc/shadow or cmd.exe. Directories can be traversed by using the dot-dot-slash attack (../), or path traversal.

```
/(\.|(%|%25)2E)(\.|(%|%25)2E)(\/|(%|%25)2F|\\|(%|%25)5C)/i
```

Explanation:

(\.|(%|%25)2E) (\.|(%|%25)2E)	Two dots & their URL encoded equivalents, including the double percent hex encoding.
(\/|(%| %25)2F|\\\|(%| %25)5C)	the slash and the backslash (& their URL encoded equivalents), as a directory separator can be "\" but also "/"

Additionally, encoded requests can be combined in many different ways. Some examples of URL encoding and double URL encoding:

Encoding variant	Representation
%2e%2e%2f	../ (%2e : dot; %2f : slash)
%2e%2e/	../
..%2f	../
%2e%2e%5c	..\ (%5c : backslash)
%2e%2e\	..\
..%5c	..\
%252e%252e%255c	..\ (This is a double percent hex encoding: the %25 represents a percent char)
..%255c	..\ (another double percent hex encoding)
..%c0%af	../ (UTF-8 encoding)
..%c1%9c	..\ (UTF-8 encoding)

Database records are usually referenced by a URL parameter like DocumentID, AccountID, StatementID or simply id. If those keys are numerically only, one can search for non-numeric arguments.

Web applications often use the account number as the primary key. Therefore the account number can be directly manipulated in a

parameter field. Attackers would usually try to loop through all (or some) of the possible account numbers, trying to find valid user accounts. This can be detected by recording the IP addresses and parameter values. If a single IP address tries more than a certain amount of account numbers, an attack is happening. (see also A7 - Broken Authentication and Session Management)

Detecting real world directory traversal attacks

Scan 31 from the Honeynet Project Challenge [Honeynet Project - Scan 31, 2004] will serve us again as an example for a real world directory traversal attack:

```
68.48.142.117 - - [09/Mar/2004:22:29:43 -0500]
"GET /scripts/..%255c../winnt/system32/cmd.exe?/c+dir HTTP/1.0"
200 566 "-" "-"
```

The '%255c' is a double percent hex encoding. The '%25' resolves to a percent character ('%'), the resulting '%5c' resolves to a backslash ('\'). The request tries to access the cmd.exe program, the windows command shell to execute the 'dir' command (list all files in a directory). This request is very common for the Nimda worm. Nimda uses the Unicode Web Traversal exploit to attack unpatched Microsoft IIS web servers.

4.3.5 A5 - Cross Site Request Forgery (CSRF)

Cross site request forgery attacks are probably the most spread attack in web applications today. It takes advantage of one of the most basic HTML feature – the link. Any process with a request like this is vulnerable:

```
https://www.example.com/transfer.php?amount=100&toAcct=12345
```

Basically, every application whose requests are based only on credentials automatically submitted such as a session cookie is vulnerable.

To exploit a CSRF vulnerability, an attacker could post the following to a forum:

<img src="https://www.example.com/trx.php?amt=100&toAcct=1234" width="0" height="0">

The browser will try to load the zero-width (i.e., invisible) image by making a request to the specified URL. It is not important that the image URL does not refer to a proper image, the request will be sent anyway (providing the browser did not disable to download images).

How can this be detected in a log file? One approach would be to measure the time-difference of the requests of a user. If there was no user input for several minutes and then suddenly some transfer requests are coming in, it could be an indicator that this request was triggered by something/someone else. It would be an error prone way as one would have to define certain time limits, which will vary from user to user.

A better approach is the use of the referer. The "referer" (sic) is a HTTP request header which will be logged in the Combined Log Format. The referer indicates the last URL, which linked to the current request. In case of a CSRF, this URL will be the attackers site. Let us make an example: an attacker prepares a website on the following URL:

http://www.attacker.com/freestuff.php

The attacker tricks a legitimate user – which is logged into

his online banking application (https://www.bank.com/) to surf to his prepared site. This site contains a XSRF attack which makes a transfer on the user's online banking application. The log file on the banking application would look something like this:

192.168.4.6 - - [10/Oct/2007:13:55:36 -0700] "GET /trx.php?amt=100&toAcct=12345 HTTP/1.0" 200 4926 "http://www.attacker.com/freestuff.php" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322)"

Two fields are important here, the requested URL (/trx.php?amt=100&toAcct=12345) and the referer ("http://www.attacker.com/freestuff.php"). Usually, the referer is an URL from the same site (www.bank.com). Here is a sample perl snippet, how this could be detected:

```
# assuming $referer is set with the, well, referer
     if ( ( $referer ne '-' ) &&
          ( $referer !~
     /^https?:\/\/www.bank.com\/(login|overview|trx)\.jsp/ )
          ) {
          # handle XSRF attack
          print(“XSRF attack: $referer\n”);
     }
```

4.3.6 A6 - Information Leakage and Improper Error Handling

Information leakage usually happens in error pages which give away too much information. Error messages can contain valuable data like if a user name exists on the system, application paths, server information and configuration files. Error pages can be recognized with the three-digit HTTP status code, which is logged for every request. These HTTP status codes are mostly used for monitoring the server and debugging purposes. We can analyze them

to detect attacks like user name enumeration or an abnormal amount of error-requests. Here is an overview of the HTTP status codes:

Status code	*Meaning*
1XX	Informational
2XX	Successful
3XX	Redirection
4XX	Client Error
5XX	Server Error

In general, all error codes >= 400 indicate some serious error and should be looked into. Some error codes like the 404 (Not Found) are obviously very common. However, they can be as interesting as the others if they give out (too) much information.

4.3.7 A7 - Broken Authentication and Session Management

Most applications implement their own authentication, password management and timeout, and are thus not easily detectable in web server log files. There are some basic principles though, how it should not be implemented. There are two notable principles which are visible in log files:

- Do not accept new, preset or invalid session identifiers from the URL or in the request. This is called a session fixation attack;
- Do not expose any session identifiers or any portion of

valid credentials in URLs or logs (no session rewriting or storing the user's password in log files).

Generally, a URL query string should not be used for any sensitive data like session IDs, user/session information, user names, passwords, etc. URLs are stored in the browser cache and are logged in web proxies and stored in the proxy cache. If session IDs are cached by a proxy, it might be possible that other users will be able to access this users account.

Examples:

https://www.example.net/login?userid=bill&password=1234

https://www.exampl.net/doSomething?varA=123;jsessionid=1234

Parameter names are application specific but are easily identifiable:

/login\.jsp.*\?.*(userId|password)=./

/;jsessionid=./

Explanation:

login\.jsp.*\?.*(userId|password)=.	The login.jsp script followed by a question mark and either the parameter userId or password.
;jsessionid=.	Search for the jsessionid parameter in the URL.

4.3.8 A8 - Insecure Cryptographic Storage

Insecure cryptographic storage is not detectable with automated vulnerability scanning tools. There is also no trace in the web server log how sensitive data is stored.

4.3.9 A9 - Insecure Communications

Every HTTP application uses some form of authentication which has to be transmitted over the Internet. To ensure the safe and confidential transmission over an insecure medium like the Internet, all authentication traffic needs to go over SSL, not just the actual login request.

How can we verify that the application properly encrypts all authentication and sensitive communications? Web servers provide support for logging any SSL-related aspect of the request. Apache, for example, allows with the CustomLog directive to log information about the SSL parameters. Here are some interesting directives:

Directive Name	*Description*
%{SSL_PROTOCOL}x	the protocol version
%{SSL_CIPHER}x	cipher suites
%{SSL_CIPHER_USEKEYSIZE}x	key size

It is recommended to at least log the protocol version and chosen cipher suites:

```
CustomLog logs/ssl_request_log \
    "%t %h %{HTTPS}x %{SSL_PROTOCOL}x %{SSL_CIPHER}x
    %{SSL_CIPHER_USEKEYSIZE}x %{SSL_CLIENT_VERIFY}x
    \"%r\" %b"
```

Example log entry:

2007.12.04-04:43:30 10.3.78.36 on SSLv3 RC4-MD5 128 NONE "GET /index.html HTTP/1.1" 13552

This request is using a 128 bit SSLv3 connection with a RC4-MD5 cipher suite.

To ensure that all HTTP requests are SSL encrypted, the SSL requests log file can be monitored for non-encrypted requests. Example:

2007.12.04-20:09:49 10.73.60.22 off - - - - "GET /" -

If the web server only supports SSL connections over TCP port 443 in the first place, then one should only see encrypted connections in the log file. It is always a good idea to verify though.

4.3.10 A10 - Failure to Restrict URL Access

When the application fails to restrict access to administrative URLs, the attacker can access such pages by typing in the URL's into the browser. This is surprisingly common, for example:

add_account_form.php - checks for admin access before displaying the form.

This form then posts to add_acct.php which does the work, but doesn't check for admin privileges!

A consistent URL access control has to be carefully designed. For detecting such kind of attacks, there is no foolproof method available. Once again, the referer check can be helpful. If you see web sites other than the local one in the referer for some privileged page, this might be suspicious. One can also monitor

files which are not supposed to be accessed like include and library files.

5 Conclusion

There are two fundamentally different attack detection methods – rule-based detection (static rules) and anomaly-based detection (dynamic rules). Web server log analysis is a rule-based detection mode which concentrates on web attacks which are visible in default web server log files like Apache or IIS.

There are several hurdles which have to be overcome to detect attacks in log files. First, the attack vectors have to be known to make detection rules. Hence, it is important to know as many different attack variants as possible. Another hurdle is the different encoding variants and standards. Standards are important but can sometimes be difficult as different vendors implement them slightly different. Each web server also supports different standards, which have to be accounted for.

Once the different idiosyncrasies are studied, well known attacks can be easily detected and eventually reacted upon. A well defined set of regular expressions allow the identification of many of the OWASP Top Ten most critical web application security flaws.

6 References

The Honeynet Project & Research Alliance (2007). Know your Enemy: Web Application Threats. http://www.honeynet.org/papers/webapp/

Friedl, J. (1997). Mastering Regular Expressions. Sebastopol,

CA: O'Reilly Media, Inc.

Kruegel, C., Vigna, G. (2003). Anomaly Detection of Web-based Attacks. New York, NY: Association for Computing Machinery.

Mookhey, K. K., Burghate N. (2004). Detection of SQL Injection and Cross-site Scripting Attacks. http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-mookhey/old/bh-us-04-mookhey_whitepaper.pdf

Anley, C. (2002). Advanced SQL Injection In SQL Server Applications. http://www.nextgenss.com/papers/advanced_sql_injection.pdf

Roelker J. D., (2003). HTTP IDS Evasions Revisited. http://docs.idsresearch.org/http_ids_evasions.pdf

Web Application Security Consortium, (2006). Web Application Firewall Evaluation Criteria. http://www.webappsec.org/projects/wafec/

SANS Institute, (2007). SANS Top-20 Internet Security Attack Targets. http://www.sans.org/top20/

SC Magazine, (2007). Web app exploits biggest hacking target in 2007. http://www.securecomputing.net.au/print.aspx?CIID=72867

IT Week, (2006). Web applications are easy targets. http://www.vnunet.com/articles/print/2148638

Fortify Software Inc., (2006). Web Applications Under Attack. http://www.fortifysoftware.com/reports/threatreport.jsp

The Open Web Application Security Project (OWASP), (2007). OWASP Top Ten Project.

http://www.owasp.org/index.php/OWASP_Top_Ten_Project

Apache HTTP Server, (2007). Combined Log Format. http://httpd.apache.org/docs/1.3/logs.html#combined

The Open Systems Interconnection, (1994). The Open Systems Interconnection Basic Reference Mode (ISO standard 7498-1:1994) http://en.wikipedia.org/wiki/OSI_model

Breach Security (2007). ModSecurity for Apache. http://www.modsecurity.org/

The MITRE Corporation (2007). Common Vulnerabilities and Exposures (CVE®). http://cve.mitre.org/about/

Christey, S., Martin, R. A., (2007). Vulnerability Type Distributions in CVE. http://cve.mitre.org/docs/vuln-trends/

The Honeynet Project, (2004). Scan 31 - Discover how an OpenProxy is abused. http://www.honeynet.org/scans/scan31/

Fielding et al., (1999). RFC 2616: Hypertext Transfer Protocol — HTTP/1.1. http://www.ietf.org/rfc/rfc2616.txt

Ellacoya Networks, Inc. (2007). Press release: Ellacoya Data Shows Web Traffic Overtakes Peer-to-Peer (P2P) as Largest Percentage of Bandwidth on the Network http://www.ellacoya.com/news/pdf/2007/NXTcommEllacoyaMediaAlert.pdf

Greenemeier L., (2006). InformationWeek: Web App Hack Incidents Are Up As Businesses Take Cover http://www.informationweek.com/industries/showArticle.jhtml?articleID=185300842