MP 99B0000087

MITRE PRODUCT

Secure Configuration of the Apache Web Server

Apache Server Version 1.3.3 on Red Hat Linux 5.1

Rev 1.12 – 24 Apr. 2001

Kenneth Jones
Rosalie McQuaid
Charles Schmidt

Revisions by Trent Pitsenbarger, National Security Agency
W2Kguides@nsa.gov

Sponsor:	NSA	**Contract No.:**	DAAB07-99-C-C201
Dept. No.:	G021	**Project No.:**	0799N030-WB

MITRE
Center for Integrated Intelligence Systems
Bedford, Massachusetts

MITRE Department Approval: ______________________________

Marion C. Michaud
Department Head
Information Warfare and Secure
Systems Engineering

MITRE Project Approval: ______________________________

Julie L. Connolly
Project Leader, 0799N030-WB

Preface

Style Conventions:

- Apache module names are given in *italics*.
- Apache directive names are given in **bold**.
- Parameters to Apache directives are given in ***bold italics***.
- Configuration files and information returned by the command line are expressed in `courier new font.`
- Text entered in the command line is expressed in **`bold courier new font.`**
- Abstract configuration information is given in *`italic courier new font`*.
- Words that the authors wish to emphasize, but which otherwise have no specific meanings, are underlined.
- The rest of the document is written in normal Times New Roman font.

Warnings

- **Do not attempt to implement any of the settings in this guide without first testing in a non-operational environment.**
- This document is only a guide containing recommended security settings. It is not meant to replace well-structured policy or sound judgment. Furthermore this guide does not address site-specific configuration issues. Care must be taken when implementing this guide to address local operational and policy concerns.
-
- Please keep track of the latest security patches and advisories.

Table of Contents

Section 1

Introduction

1.1 Purpose

MITRE has performed a secure configuration analysis of the Apache Web Server on Linux. This investigation was initiated to provide an understanding of the security mechanisms within the Apache Web Server. The Apache Web Server is the most popular web server on the Internet; more than 50 percent of the existing web servers use Apache. Due to this popularity, MITRE has identified the need to provide secure configuration guidelines for the Apache Web Server on Linux.

1.2 Scope

This document is intended to detailed descriptions for the configuration of a "secure" web site using the Apache Web Server. This document assumes no prior knowledge of the Apache Web Server, and only limited understanding of web servers in general. It does, however, assume some understanding of the UNIX operating system as implemented on Linux. Readers should be familiar with file security, file structure, and basic UNIX/Linux commands.

The Apache Web Server is an extremely powerful and adaptable product. A complete documentation of all its features is out of the scope of this document. Included in this guide are the features of the web server which have a direct influence on the security of the web site, or that are so common, that no reasonable treatment of modern web servers could be expected to exclude them. The Apache security services are described along with examples that outline possible configurations.

1.3 Background

Due to this increased reliance on and widespread use of web technologies, MITRE was tasked to complete a secure configuration guide for the Apache Web Server. This task was completed by establishing a test bed for the Apache Web Server Version 1.3.3; this was installed on Red Hat Linux 5.1. Test configuration files were developed to implement and test the security services of the web server. Based on the test results, this secure configuration guideline was developed. The secure configuration guide covers the security services of authentication, access control, availability, and auditing. It does not cover nonrepudiation, confidentiality, and integrity due to the fact that standard Apache does not implement these security services. These services are available in Apache SSL. This guide does address other security issues which may be present in web servers including Common Gateway Interface (CGI), Server Side Includes (SSI), redirection, virtual hosts, and aliasing.

1.4 Document Organization

This document consists of three sections pertaining to the Apache Web Server Secure Configuration Study. Section 2 consists of an overview of the Apache Web Server. This overview includes a section describing basic web server principles along with the details of the Apache architecture. Section 3 provides the secure configuration guidelines along with corresponding configuration issues. The Apache security services, authentication, access control, availability, and auditing, are described along with the modules and directives used to implement the security service. Recommended configurations are included for each security area. Other security relevant issues are also discussed in this section. Section 4 provides a brief summary and recommendations.

Section 2

Apache Overview

The Apache Web Server is the most popular web server on the Internet; more than 50 percent of the existing web sites use Apache. It was developed by a worldwide group of volunteers known as the Apache Group that jointly manage the Apache Hypertext Transfer Protocol (HTTP) Server Project. The Apache Group has worked hard to produce a robust, highly configurable, and freely-available web server.

The first version of Apache was released in April 1995 (Version 0.6.2) and is currently at Version 1.3.6. The core contributors of the Apache Group used NCSA httpd 1.3 as the base for the initial release of Apache Version 0.6.2. The Apache user community grew rapidly after the initial release and development, and refinement continued on the Apache HTTP Server project; Version 0.7x was being designed during May—June, 1995. Although the initial release was a big hit, the group decided that the server needed a new architecture. This was designed and implemented in Version 0.8.8, released in August 1995. The new server architecture consisted of a modular structure, an API for better extensibility, a new forking process model, and pool-based memory allocation.

This section provides an overview of the Apache Web Server. The overview provides a section containing basic web server principles along with the complete Apache Web Server architecture.

2.1 World Wide Web Model

The World Wide Web (WWW) has been described as a distributed heterogeneous collaborative information system. The WWW mission is to provide easy access to an information resource in a format that is well defined and can be readily displayed. It consists of a model made up of web servers, web clients, a transmission protocol, and a format specification for data. The web server and client are considered software components; the transmission protocol, and data format specifications are protocol components. These model components are described in the following sections.

2.1.1 Web Server Component

The aspect of a distributed system of information resources is met by the web server component. Web servers can be installed on a wide variety of computer platforms and essentially serve as the controller/provider of information resources. It is the server component of the WWW model that provides information to a browser (the client) through the transmission protocol. The browser interprets the data returned from the server and graphically renders the information; however, there are some text-only browsers that do not have a graphical capability.

2.1.2 Web Client

The web client provides for the ready display of multimedia information and is commonly known as the browser. The web client must be versatile since it has the role of interpreting the data provided by the web server and displaying it, in the intended form, to the user. Many web browsers have the capability to execute embedded instructions. These instructions (e.g., Java Applets, JavaScript, VBScript, and others) can instruct the server to execute a program residing in local memory/disk space or pass a request to another server resource. The output of either of these actions can be directed back to the client through the web server or by the called server resource.

2.1.3 Transmission Protocol: HyperText Transfer Protocol

The HTTP is the request/reply protocol used for communication between the web browser and the web server. This mechanism to transmit the information from the location of the resource to the location of the client is unseen. It is a set of rules that govern how the web browser makes requests and the web server responds; these are formatted according to the specifications of the protocol. HTTP is part of the Advanced Research Projects Agency ARPANet family of protocols. This family includes other request/reply protocols, such as File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), and Telnet.

HTTP is encapsulated within a Transmission Control Protocol (TCP) connection. The request/reply nature of the protocol results in a stateless protocol; no information from an earlier request is retained for use in a later request. The general rule is that there is a reply for each request. The reply may be an interim status message that will be followed by a complete response when the server is able to process and transmit the information requested by the client. As the request for a document is satisfied, the underlying TCP connection is closed. A new request for a document (e.g., clicking on a hyperlink) will result in a new TCP connection.

There are three primary message types for HTTP; the GET request, the HEAD message, and the POST request. The GET request is used to retrieve information from the web server identified in the Universal Resource Locator (URL). The HEAD message is similar to the GET message; the difference is that the server responds with the header information only and not the body of the document. Robots which build/update search engine databases typically use this type of message. The POST message is used to post a message or submit form data.

New versions of web servers and web browsers support TCP/HTTP Keep-Alives. They are a feature of HTTP Version 1.1. Keep-Alive establishes sessions and avoids the overhead of constantly creating and closing separate TCP connections.

2.1.4 Data Format Specification Component

The final aspect of the WWW model is the data format specification; this describes how the multimedia information can be readily displayed at the client. Most commonly, a family of special purpose scripting languages, known as markup languages, describes the form and content rules. They allow consistent information display across a wide variety of web browsers. The most familiar of the markup languages is the HyperText Markup Language (HTML). New data format specifications are constantly being added to the WWW model. The parent set of markup language specifications is the Standard Generalized Markup Language (SGML).

2.2 Apache Architecture

Apache is a powerful and widely used web server with a completely modular architecture. It can be implemented on many widely used computer platforms including most flavors of , Windows95/NT, and other server operating systems. This, along with its modular architecture, makes it extremely popular throughout the web. It offers many features and functions that can be added or removed depending on the needed functionality of the web server. Some features include: Server-side image maps, configurable HTTP error reporting, directory aliasing/fancy indexing, content negotiations, URL rewriting, and resource management for child processes. These features are implemented through Apache's modular architecture described in the following sections.

The Apache architecture provides the configuration of a customized web server. The Apache architecture can be divided into Apache file system layout, component module architecture, configuration mechanisms, and run-time modes. Each of the architecture areas can be customized. These areas are detailed in subsequent sections.

2.2.1 Apache File System Layout

The Apache file system layout is configured when Apache is installed on the server. It can be installed either in the default mode or in a customized file configuration, using the default mode all apache files are located under the server root directory. The server root is specified during the installation process. The default file layout is configured as shown in Figure 1. The recommended custom file layout is shown in Figure 2.

The default file layout is located under the server root directory. Within this directory there are subdirectories created called bin, conf, and logs. In the "bin" directory the httpd executables and the associated utilities are located. The "conf" directory holds the configuration files and configuration information. The "logs" directory holds the Apache log files.

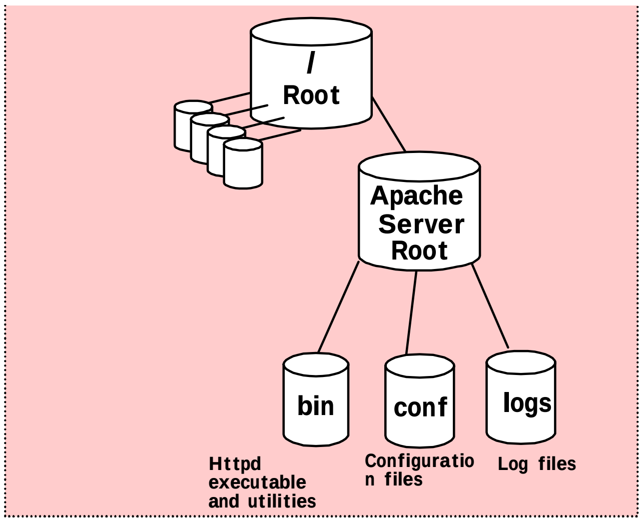


Figure 1. Default File Layout

The customized file layout is constructed to conform with UNIX standards. These standards would use the following format: “/usr” stores the subdirectory for Apache executable and utilities, “/var” stores the subdirectory for the log files, “/etc” stores the subdirectory for the configuration files, and the “/home” directory stores the subdirectory for the published material and CGI scripts.

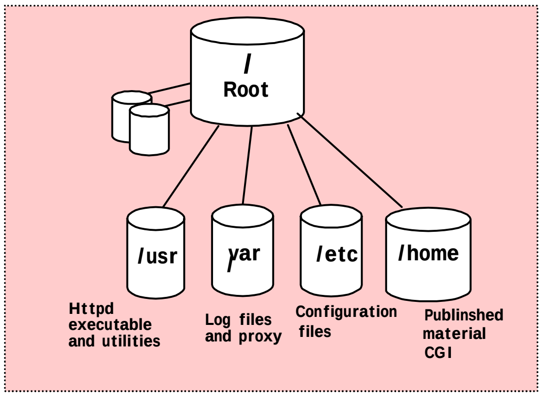


Figure 2. Customized File Layout

Each file system layout has its own advantages. The default layout has all the Apache files in one central location; under the server root. This allows the administrator to know exactly where to look for any Apache files; it is also helpful if the web server and its content is moved to another platform. The customized file layout is standard for UNIX platforms and would conform with the configuration used for other services on the platform. This has the advantage of already having the directory permissions set properly.

2.2.2 Component Module Architecture

The Apache Web Server is implemented using a modular architecture. The component module architecture consists of a core component along with a variety of add-in features. These are implemented through default, standard, and third party modules. A module is a software component that adds specific functionality to Apache. Combining the core module with other modules allows the user to construct an Apache Web Server that is customized to a specific site application; this is shown in Figure 3. The following sections detail the elements of the Apache component module architecture.

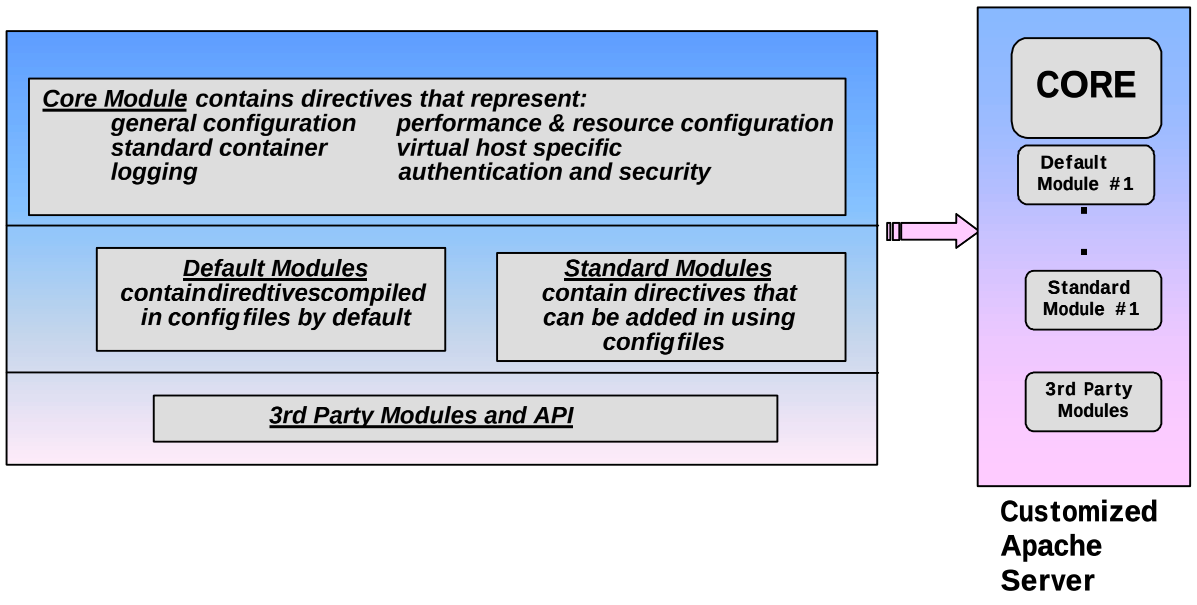

Core Module contains directives that represent:
general configuration
performance & resource configuration
standard container
virtual host specific
logging
authentication and security
Default Modules
containdiredtivescompiled in configfiles by default
Standard Modules
contain directives that can be added in using configfiles
3rd Party Modules and API
CORE
Default Module #1
Standard Module #1
3rd Party Modules
Customized Apache Server

Figure 3. Apache Component Module Architecture

Modules and Directives

The module architecture consists of a core module, default modules, standard modules, and third party modules. Each module is a software component that adds specific functionality to the web server through the use of directives. They are commands used to control the behavior of the web server. To use a specific directive, the module containing that directive must be included in the web server. These modules can be added to or removed from Apache at compilation by using a configure utility.

The core module contains core directives that control general configuration, performance and resource configuration, authentication and security, logging, and other features. The core directives listed in Table 1, are always available. Container directives are directives that are enclosed by container tag “<””>” pairs.

- The general configuration directives address fundamental settings for the server and for virtual hosts.
- The standard container directives apply a group of other directives to a particular file, directory, location, etc.
- The virtual host directives are used specifically for creating virtual hosts; some directives listed in the general configuration section also apply to virtual hosts.
- The performance and resource directives are used to control Apache processes, system resources, and make persistent connections.
- The logging directives enable server log information.
- The authentication and security directives define access control and security policies for the server.

Table 1. Core Directives

General Configuration	Standard Container	Virtual Host	Performance and Resource	Logging	Authentication and Security
AccessConfig	<Directory>	AccessConfig	AddModule	ErrorLog	AllowOverride
accessFileName	<DirectoryMatch>	AccessFileName	ClearModuleList	IdentityCheck	AuthName
BindAddress	<Files>	NameVirtualHost	KeepAlive	LockFile	AuthType
BS2000Account	<FilesMatch>	<VirtualHost>	KeepAliveTimeout	LogLevel	require
ContentDigest		ServerAlias	LimitRequestFields	PidFile	Satisfy
CoreDumpDirectory	<Limit>	ServerPath	LimitRequestFieldsi	ScoreBoardFil	
DefaultType	<Location>		LimitRequestLine		
DocumentRoot	<LocationMatch>		MaxClients		
ErrorDocument			MaxKeepAliveRequ		
Group			MaxRequestsPerChi		
HostNameLookups			MaxSpareServers		
Include			MinSpareServers		
Listen			RLimitCPU		
ListenBacklog			RLimitMEM		
<IfModule>			RLimitNPROC		
Options			ServerType		
<IfDefine>			StartServers		
Port			ThreadsPerChild		
ResourceConfig			SendBufferSize		
ServerName			TimeOut		
ServerAdmin					
User					
ServerRoot					
ServerTokens					

The default modules contain directives that are included in the server by default; these can be removed if desired. The standard and third party modules contain directives that add functionality to the server. These directives are used in configuration files that are read at runtime. There are many available modules in the default and standard set of Apache modules. New modules and third modules are constantly being added to this list. Table 2 lists the default and standard modules for Apache Release 1.3.3. A list of third party modules available can be found at the Apache web site (url:www.apache.org). Apache also contains an Application Programming Interface (API) that allows developers to build modules to provide a specific functionality to the web server. This module architecture and the ability to add and remove modules allows customization of the Apache web server.

Table 2. Apache Default and Standard Modules

Module Name	Function
mod_access *Default*	Host-based access control.
mod_actions *Default*	Filetype/method-based script execution.
mod_alias *Default*	Aliases to map one part of the server's file system to another and url redirection.
mod_asis *Default*	Documents can be sent without HTTP headers or as is.
mod_auth *Default*	User authentication using text files.
mod_auth_anon *Standard*	Allows anonymous user access to authenticated areas FTP style.
mod_auth_db *Standard*	User authentication using Berkeley DB files.
mod_auth_dbm *Standard*	User authentication using DBM files.
mod_autoindex *Default*	Provides automatic directory indexing.
mod_cern_meta	Support for HTTP header metafiles.
mod_auth_external *Check this out if available*	Support for third party authentication modules.
mod_cgi *Default*	Provides for execution of CGI scripts.
mod_digest *Standard*	Provides for user authentication using MD5 Digest Authentication.
mod_dir *Default*	Provides basic directory handling; any request for a directive that does not include a trailing slash character is redirected.
mod_env *Standard*	Passes environments to CGI scripts.
mod_example *Standard*	Illustrates many aspects of the Apache API and demonstrates callbacks triggered by the server.
mod_expires *Standard*	Applies Expires setting HTTP header in server response.

Module Name	**Function**
mod_headers *Standard*	Enables the addition of arbitrary HTTP response headers.
mod_imap *Default*	Provides for .map files.
mod_include *Default*	Provides for server-parsed html documents.
mod_info *Standard*	Provides comprehensive overview of server configuration.
mod_isapi *Default*	Provides support for ISAPI Extensions.
mod_log_agent *Standard*	Provides for logging of client user agents.
mod_log_config *Default*	Provides for logging of requests made to the server.
mod_log_referer *Standard*	Provides for logging of the documents which reference documents on the server.
mod_mime *Default*	Support for determining the types of files from the filename.
mod_mime_magic *Standard*	Used to determine the MIME type of a file by looking at a few bytes of its contents.
mod_mmap_static *Standard*	Experimental module that maps a list of statically configured list of frequently requested but unchanged files.
mod_negotiation *Default*	Provides for content negotiation.
mod_proxy *Standard*	Provides for an HTTP 1.0 caching Proxy server.
mod_rewrite *Default*	Uses a rule-based rewriting engine to rewrite requested URLs.
mod_setenvif *Default*	Provides for the ability to set environment variables based upon attributes of the request.
mod_so *Standard*	Experimental module for loading modules into Apache at runtime via the DSO.
mod_speling *Standard*	Automatically corrects minor typos in URLs.
mod_status *Default*	Allows a server administrator to check server perfomance.

Module Name	Function
mod_userdir *Default*	Provides for user-specific directories.
mod_unique_id *Default*	Generates unique request identifier for every request.
mod_usertrack *Default*	User tracking using Cookies.

2.2.3 Configuration Mechanisms

Apache uses configuration mechanisms to specify server functionality. These mechanisms are four configuration files: *srm.conf, httpd.conf, access.conf,* and *mime.type*. These text files contain comments and directives used to specify the behavior of the server. The Apache source distribution has sample configuration files called srm.conf-dist, httpd.conf-dist, access.conf-dist. These files can be renamed and edited to use for the site-specific installation. The configuration file *httpd.conf* is used as the primary configuration file; specifying to the operation of the server daemon. The configuration file *srm.conf* is the resource configuration file. It configures the server to offer specific resources/documents on the Apache server. The configuration file *access.conf* is used to set permissions for files, directories, and scripts on the web server. *Mime.type* is not modified for most web server installations and will not be discussed. Another mechanism, *.htaccess,* is used to apply configuration directives on a per directory basis. The configuration files are read into the system at runtime.

2.2.4 Apache Run-Time Modes

Apache can be configured to run using two different modes; stand-alone and inetd. These run-time modes are configured by using the core directive “ServerType” with either **stand-alone** or **inetd** specified in the directive. The ServerType directive is specified in the *httpd.conf* configuration file.

Although the functionality of the server is the same; the performance will vary greatly depending on the run-time mode. In the **stand-alone** mode, the web server child process lives for a time before closing down; this allows reuse of the process if a request is received during this time. In the **inetd** mode, the web server process exits as soon as it is finished servicing a request. The stand-alone mode is more efficient because it does not have to launch a new process each time a request is received. These modes are described in more detail in the following sections.

2.2.4.1 Standalone Mode

In the stand-alone mode the server will run as a daemon process. This is the default setting for the directive "ServerType stand-alone" used in the *http.conf* file. As shown in Figure 4, the server listens for a connection request on a specific port. When the connection request is received, the primary server launches a child process to service the request. The child process will not shut-down immediately. It will continue to service requests until a specified request threshold has been met.

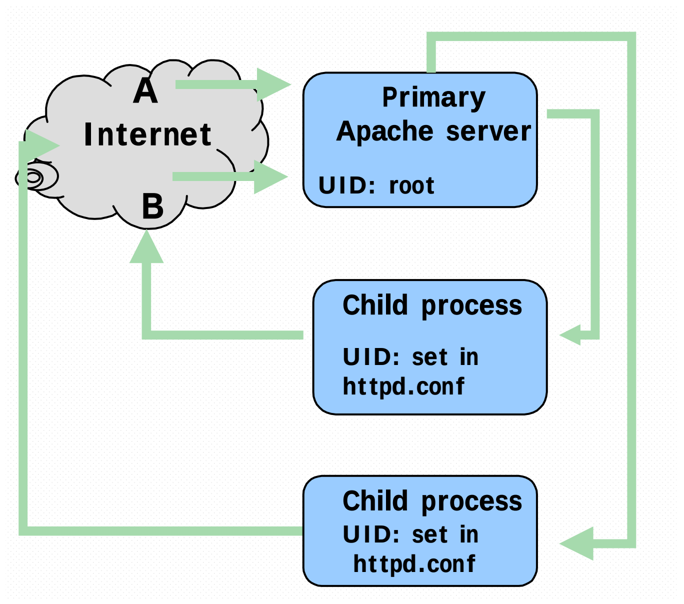


Figure 4. Stand-Alone Mode

The primary server listens for client requests on a specific port. This port is defined using the "Port" directive. Typically, the HTTP port is 80. If the server is not running under the "root" user context, then a port between 1024 and 32768 must be specified.

When running in the stand-alone mode, the child user ID and the group ID must be specified using the core directives "User" and "Group," respectively. The parent process can run as root with the child running as a different user/group. For security reasons, the child process should not be run as "root;" this would allow the child to have "root" user privilege. The default for the user and group ID is the user/group "nobody." This is a low-privileged user that belongs to a low privileged group. It should be noted that the user/group ID can only be changed if the primary server process is being run under the "root" user context. If the primary server is being run by user "Jane;" the child processes will have the same privileges as "Jane."

2.2.4.2 Inetd Mode

In the inetd mode the web server is run as an inetd child process. The inetd process is the Internet daemon. This is specified in the httpd.conf file with the directive "ServerType inetd." The inetd server process listens for connection requests on ports 0 through 1023 as shown in Figure 5. When connection requests are received, the inetd daemon launches the one httpd process per request. The httpd process services the request and then exits.

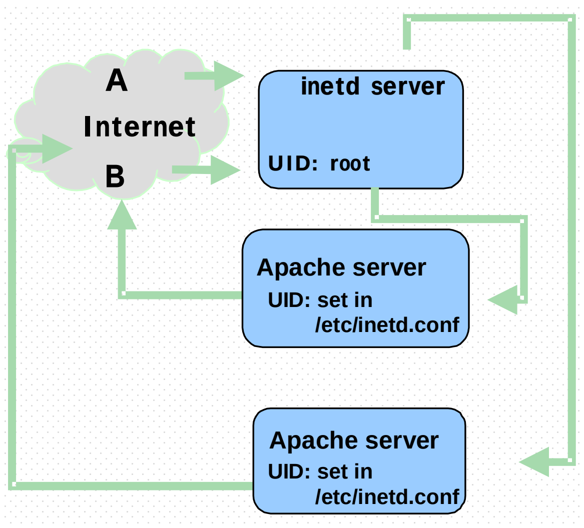


Figure 5. Apache inetd Mode

When running Apache in the inetd mode, the inetd.conf file (an operating system configuration file) must have a record added for Apache. This record includes the service name, socket type, protocol, flags, user ID, and server path. The httpd service must be run as a particular user. A special user should be created to run the httpd service; this user would be similar to the user "nobody." This user should have access privileges to the web directives and the log file directories. For example, if the user "httpd" is used, the inetd.conf entry would be:

```
httpd stream tcp nowait httpd /usr/sbin/httpd –f /etc/httpd/conf/httpd.conf
```

An entry must also be made in the /etc/services file. This entry should be:

```
httpd 80/tcp/ httpd
```

This describes the httpd service as available on port 80.

For perfomance reasons, the inetd mode should not be used for a high-traffic web server. The server should be run in the stand-alone mode.

2.3 Apache Configuration Roadmap

The Apache architecture provides the configuration of a customized web server and consists of Apache file system layout, component module architecture, configuration mechanisms, and run-time modes, as described in previous sections. Each of the architecture areas can be customized for installation. The Apache environment roadmap, in Figure 6, shows how the pieces of the Apache architecture combine to produce the customized web server.

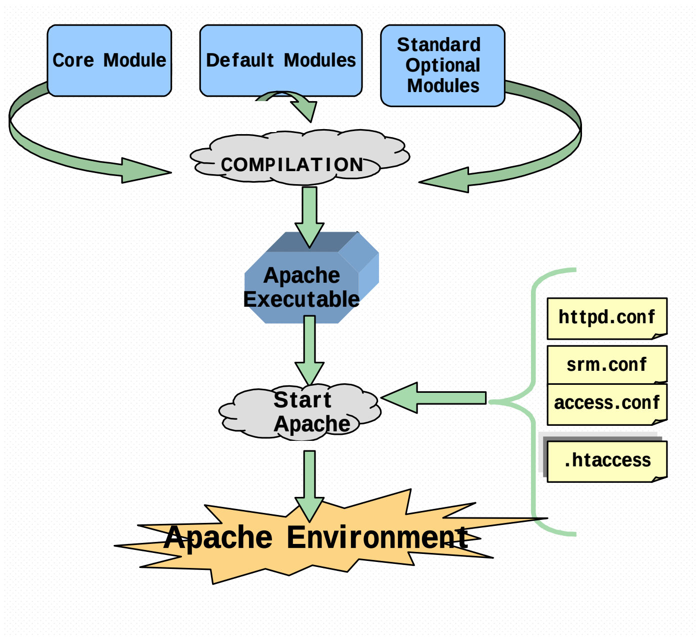


Figure 6. Apache Environment Roadmap

The core, default, and standard modules that comprise the Apache executable are specified by the configure script. The *AddModule* statement is used in the configure script to add modules to the Apache functionality. The configure script runs and creates the Makefile used to compile the Apache source file. Compilation produces the Apache executable, ***httpd***. When ***httpd*** is started, the configuration files, *httpd.conf*, *srm.conf*, *access.conf* and *.htaccess* are read. This produces the customized Apache environment.

2.4 Apache Security Services

This section provides a brief description of the security services provided in the Apache Web Server. These services have been divided into seven distinct security areas: authentication, access control, auditing, availability, integrity, confidentiality, and nonrepudiation. The Apache security services are:

Authentication: Authentication provides the mechanism to verify the identity of a client through the exchange of information. This is done when the client sends an authorized username/password pair to the server. Apache provides the functionality to set up access control policies based on the authenticated identities of clients.

Access Control: Access control provides the mechanism to grant access to system resources based on the identity of the user. Apache provides the functionality to set up host-based access control where the requesting host is either allowed access or not allowed access to resources on web server.

Auditing: Auditing provides the mechanism which ensures that actions performed by all users authorized or otherwise, are recorded. Apache auditing provides a record of the activities taken on the Apache server. Administrators can determine server usage, resource usage and service response difficulties. They can also determine if and when security attacks have been attempted on the server.

Availability: Availability provides the mechanism to ensure that system data is available to users when required. Apache provides availability within the core module by allowing the administrator to set configuration variables to maximize performance and availability.

Confidentiality: Confidentiality provides the mechanism to ensure that data is protected from accidental or purposeful disclosure. This process would include message encryption to or from the server. This service requires the use of modern cryptographic protocols, usually SSL. Unfortunately, the standard Apache does not support SSL at this time but an Apache derivative for SSL version is available commercially.

Integrity: Integrity provides a mechanism for the protection of system data from accidental or purposeful alteration in transit; there is an important distinction between confidentiality and integrity. This is usually done through the use of a cryptographic checksum or other similar procedure. The Apache server does not implement this service which means that an attacker could tamper with Apache messages in transit.

Nonrepudiation: Nonrepudiation is the mechanism to assure or prove that the source of data or message cannot deny authorship. It is not implemented in Apache and may not be implemented in web servers.

These security services are described in more detail in Section 3. Each service is described along with the modules and directives used to implement the security service. Recommended configurations are included for each security area.

Section 3

Secure Apache Configuration and Configuration Issues

This section considers each of the security relevant features of the Apache Web Server. It also describes how to implement them in typical configurations. The first section, General Server Settings, describes directives that provide support to the other areas of functionality. The subsequent sections describe modules that implement specific security services or areas of functionality that have a security impact.

3.1 General Server Settings

This section describes directives that are part of the core structure which are used by other areas of functionality, but do not actually fit into one of the areas of functionality themselves. These directives are divided into their areas of primary functionality. These are: container directives, group and user directives, handling directories in URLs, and the **Options** directive. This section does not present any examples, reasoning that the directives will be demonstrated in subsequent sections, or the directives are self-explanatory. Nonsecurity relevant directives will not be discussed.

3.1.1 Container Directives

Container directives are used in the configuration files to specify the resources or actions to which an enclosed set of directives is to apply. There are several container directives (sometimes called "section" directives):

- **<Directory>—**Specifies a file system directory either explicitly, using a wildcard expression, or using an extended regular expression.
- **<DirectoryMatch>—**Specifies a file system directory using an extended regular expression.
- **<Files>—**Specifies a specific filename in the file system either explicitly, using a wildcard expression, or using an extended regular expression.
- **<FilesMatch>—**Specifies a specific filename in the file system using an extended regular expression.
- **<Location>—**Specifies a URL either explicitly, using a wildcard expression, or using an extended regular expression.
- **<LocationMatch>—**Specifies a URL using an extended regular expression.
- **<Limit>—**Specifies one or more HTTP methods.
- **<VirtualHost>—**Specifies a virtual host by IP address and, optionally, port. See Section 3.10 for more on the subject.

In all cases, the resources, (directories, files, URLs, or methods) appears immediately after the name of the directive (separated by a space) but before the closing angle bracket (>). A container directive is used to begin a block. The block ends with the same directive preceded by a backslash (/). For example, a **<Directory>** block would be closed by **</Directory**>. All directives enclosed within the block are applied only to the resource that was specified in the opening directive. All contained directives, the opening, and closing markers, should be on separate lines.

It is possible for multiple container directives to apply to a single resource. If this happens, then the order in which the directive blocks are applied becomes important. Container directive blocks are not applied sequentially; that is, they are not applied in the order they appear in the configuration file. Directive blocks are applied in the following order:

1. **<Directory>** without regular expressions
2. .htaccess files (See Section 3.1.5)
3. **<DirectoryMatch>** and **<Directory>** with regular expressions
4. **<Files>** and **<FilesMatch>**
5. **<Location>** and **<LocationMatch>**

When two directive blocks apply to the same resource and share the same level in the above list, the most specific block will be applied later.

If two directive blocks apply to the same resource and contain the same directive, the directive in the later applied directive block will overwrite the directive in the previously applied directory block.

Previously applied directive blocks that address a particular resource will be overridden by similar directive blocks applied later.

Directives regarding a particular resource within a directive block that is applied later will override all instances of the same directive present in directive blocks that were applied previously. This is not always completely intuitive; a specific directive in a “later” block could override itself and override other directives associated with it, causing those directives to revert to their default values.

3.1.1.1 <Directory> and <DirectoryMatch> Directives

The **<Directory>** and **<DirectoryMatch>** directives cause their directive blocks to apply to the specified directory, including subdirectories and all contents. The **<Directory>** directive allows a wildcard expression. All characters match themselves except:

- ? - matches any one character
- * - matches any group of characters
- [] - matches any one of the characters listed between the brackets

If the “Directory” in the container directive is followed by a space followed by a tilde (~), the parameter following it, is treated as an extended regular expression (see the `man` page on `grep` for more on the extended regular expression language). The **<DirectoryMatch>** directive takes a regular expression to specify its target.

In the Linux operating system, multiple paths to a given directory may exist through the use of links. (See the `man` page on `ln` for more information on links.) If a **<Directory>/<DirectoryMatch>** directive matches one path, accessing the given directory by another path, may not cause the container directive block to be applied.

The **<Directory>/<DirectoryMatch>** directives match the file system path of the directory in order to determine which resources they cover. This may not necessarily reflect the URL used to access this resource. Additionally, some directives do not make sense when applied within this context. These are the directives that are only meant to apply to the server as a whole, as opposed to specific resources. For example, the **ServerName** directive would not belong within a **<Directory>/<DirectoryMatch>** block since attempting to do so would imply trying to change the name of the server on a directory by directory basis.

3.1.1.2 <Files> and <FilesMatch> Directives

The **<Files>** and **<FilesMatch>** directives cause their directive blocks to apply to resources whose filename (excluding path) matches the specified value. The result is that every file in the entire file system whose name matches the value within the opening directive will have the container directive block applied.

Like the **<Directory>** directive, the **<Files>** directive allows a wildcard expression. If the “Files” in the container directive is followed by a space followed by a tilde (~), the parameter following it is treated as an extended regular expression. Matching is done against the file system rather than the URL and the two may differ. The **<FilesMatch>** directive takes an extended regular expression to specify its target.

Since it is not always desirable for every file of a given name in the entire file system to receive the same treatment, it is possible for **<Files>/<FilesMatch>** directives to be placed within **<Directory>/<DirectoryMatch>** directives. In this case, the filename matching will only occur within the context of the specified directories. As with **<Directory>/<DirectoryMatch>**, not all directives make sense within a **<Files>/<FilesMatch>** directive block. All directives which cannot be placed within **<Directory>/<DirectoryMatch>** directive blocks cannot be used within a

<Files>/**<FilesMatch>** directive blocks. In addition, these directive blocks cannot contain the **Options** directive, as well as others.

3.1.1.3 <Location> and <LocationMatch> Directives

The **<Location>** and **<LocationMatch>** directives cause their directive blocks to be applied to all resource requests that match the argument in the opening directive. The argument is applied against the URL being referenced. The argument should not contain the protocol or the server name unless the directive is being applied to a proxied resource. (See Section 3.11.1 for a brief overview of proxying with the Apache server.) For example, in order to match the request, `http://www.mitre.org/sample/doc.html`, the pattern in the **<Location>**/**<LocationMatch>** directive should be "`/sample/doc.html.`"

The **<Location>** directive allows a wildcard expression. If the "Location" in the container directive is followed by a space followed by a tilde (~), the parameter following it is treated as an extended regular expression. Unlike the **<Directory**> and **<Files>** directives, **<Location>** directives do not consider the resource's location in the file system ; they look only at the URL. The **<LocationMatch>** directive takes an extended regular expression as its argument.

Due to the fact that **<Location>**/**<LocationMatch>** directives do not involve the file system there are many directives that would be nonsensical within this context block. In addition to those directives which cannot be placed within **<Directory**>/**<DirectoryMatch>** directive blocks, **<Location>**/**<LocationMatch>** directive blocks cannot contain **<Files>** or **<FilesMatch**> directives, **Options** directives with parameters involving symbolic links, or other directives which are focused on the file system .

3.1.1.4 <Limit> Directives

The <**Limit>** directive is used to control access to specific HTTP methods on the server. The HTTP 1.1 specification specifies eight methods; seven have defined functionality. These are:

- OPTIONS—Requests that the server specify the communication options associated with a specific URL, resource, or the server.
- GET—Requests that a specified resource be sent to the client. This is the most common method used.
- HEAD—Identical to the GET method, but only the header information is returned.
- POST—Requests that information sent from the client be processed by the resource at the specified URL.
- PUT—Requests that the specified resource be replaced with new information. This is similar to an FTP **`put`** command.

- DELETE—Requests that the server delete the resource specified by a given URL.
- TRACE—Requests that the target server simply return the message it receives back to the client.
- CONNECT—This is a reserved method, but as of HTTP 1.1, its functionality is undefined.

The **<Limit>** directive can take a space separated list of the following parameters: ***OPTIONS***, ***GET***, ***POST***, ***PUT***, ***DELETE***, and ***CONNECT***. Each parameter corresponds to the matching HTTP method with the exception of ***GET*** which corresponds both to the GET and HEAD methods. There is no parameter to restrict the TRACE method.

The contents of the directive block are applied to any attempts by a client to use this method. Generally, only access control directives make sense within this container directive block. The **<Limit>** directive can stand alone in the configuration file, in which case it applies to all requests to the server, or within other container directive blocks in which case it only applies to requests of those resources.

3.1.2 The User and Group Directives

As described earlier in the discussion of the Apache runtime architecture (see Section 2.2.4.1), when the Apache server starts, it forks off several processes which are then tasked with servicing individual client requests. These child processes have their own user and group identities that can be used to control the amount of access they have to the file system . These identities are set by the **User** and **Group** directives respectively. The operating system uses the identities in its access control calculations. If the username under which a child process is running is not allowed to access a given document, that child will not be allowed to serve it to a client.

It is important that the user and group assigned to the server child processes be permitted access only to files and directories that it has a need to use. Operating the child process under a user or group with access to other files or executables opens the possibility that the server may publish or even alter files of a sensitive nature on the file system .

By default, both **User** and **Group** are set to “nobody,” a default user on the Apache system. Unlike other operating systems, the user “nobody” does not represent a given level of access but is simply a name and group like any other. However, as it is extremely unlikely that any files or directories will exist on the server that are specifically granted to “nobody,” access will usually be prevented to all files and directories except those with world access permitted.

As other applications sometimes use the “nobody” user and group, Apache administrators may wish to create a user for the exclusive use of Apache child processes. This user can be created in the same way as a normal user on the operating system. It important that the new user not be assigned to a preexisting group as this would actually increase the access provided to the child processes. A new group should be created along with the new user to ensure the child process is not assigned to a group that already has access to other files.

Once the new user and group are created and set in the server using the **User** and **Group** directives respectively, the administrator must ensure that all publishable content is readable by this user and that the user can access all the appropriate directories. Likewise, since CGI and other executables run under the name of the server child process, it is important that all files they depend on be accessible by that user. Log files, configuration files, and the like are accessed by the parent process and do not need to be accessible by the child process user.

3.1.3 Handling Directory References in URLs

Most modern web servers provide the means to handle direct references to directories in the URL. A directory reference occurs when a user requests a directory rather than a specific file. The server cannot return the entire directory to the client. Directories may or may not be requested with a trailing backslash (/) character and the server must be able to handle either condition. This section describes the modules that the Apache server uses to control this functionality.

The *mod_dir* module serves two functions in directory handling. First, it automatically, without the use of any directives, provides the functionality for trailing-slash redirects. A trailing-slash redirect is simply an HTTP redirect message (HTTP code 301) sent to any client that requests a directory without including a backslash (/) at the end of the directory name. Using this method, the Apache server can always be certain that the only valid path of a directory includes the trailing backslash. This is important with directives such as the **<Location>** directive that perform pattern matching on the URL.

The second feature that the *mod_dir* module provides is the ability to specify a default index file the server will look for in each directory. This functionality is provided by the **DirectoryIndex** directive. This directive takes a list of file names as parameters. When the server receives a request for a directory, the server checks the directory for one of the file names listed after the **DirectoryIndex** directive, returning the file that matches the earliest of its parameters. Index documents normally contain links into the contents of the directory.

When an index file does not exist, the Apache server uses functionality provided by the *mod_autoindex* module. This module automatically, without the use of any directives, enables the server to dynamically create an HTML page listing the contents of the directory. There are several directives in the *mod_autoindex* module that pertain to how this page appears to the user. It is possible to turn auto-indexing off using the **Options** directive, in which case the server returns a “Forbidden” error (HTTP error 403). See Section 3.11.3 regarding the security implications of auto-indexing.

Both *mod_dir* and *mod_autoindex* are part of the default Apache build and, unless the administrator wishes to replace their functionality with modules of their own design, it is recommended that they remain untouched.

3.1.4 The Options Directive

The **Options** directive is a general directive that controls a broad range of capabilities on the Apache server. It is part of the *core* module and, therefore, always available. This directive takes one or more of a list of possible arguments, each argument enabling a particular capability on the server. As the **Options** directive may be placed within a container directive block, this control may be applied on a directory basis.

Each argument of **Options** is associated with a specific capability. These are described below:

- ***All*—**This argument enables all arguments *except* ***MultiViews***, which must be explicitly added. This is the default setting of the **Options** directive.
- ***ExecCGI*—**This argument permits CGI scripts to be executed. See Section 3.7 for more on CGI scripts.
- ***FollowSymLinks*—**This argument allows symbolic links to be followed. Symbolic links are a feature of UNIX operating systems (including Linux) that allow multiple paths to a given resource. For more information, consult the **`man`** pages for the **`ln`** command.
- ***Includes*—**This argument allows server side includes to be executed. See Section 3.8 for more on server side includes.
- ***IncludesNOEXEC*—**Same as the ***Includes*** argument, but it prevents the use of the **`#exec`** command and use of the **`#include`** command, if the included file is executable. See Section 3.8 for more on server side includes.
- ***Indexes*—**This argument enables the auto-indexing capability described in Section 3.1.3. See Section 3.11.3 for more on indexing.
- ***MultiViews* –** This argument enables MultiViews. MultiViews is an advanced feature of the Apache server. If a requested resource is not present, it will check the given directory for files whose names starts with the name specified in the URL followed by a dot with any extension. Note that this argument is not enabled by the ***All*** argument and must be added explicitly.
- ***SymLinksIfOwnerMatch* –** This is a special case of the ***FollowSymLinks*** argument. The client will be allowed to follow symbolic links, but only if the link is owned by the same user ID as the target file or directory.

The **Options** directive is useful for its ability to block the use of features whose arguments it is not given. For example, if a user wishes to prevent CGI scripts from being executed, they would add an **Options** directive with an argument list that did not include the ***ExecCGI*** argument.

3.1.5 .htaccess Files

In addition to using container directives (Section 3.1.1), the Apache server can also apply configuration information to directories based on special files within the directory itself. These files are called .htaccess files. The name of these files is set using the **AccessFileName** directive in the *core* module. By default the file name is ".htaccess." This document refers to all such files as .htaccess files regardless of their actual name.

The behavior of .htaccess files is virtually the same as that of the **<Directory>** container directive with two exceptions. The first difference is that, while multiple directories may be specified in the parameter of the **<Directory>** directive, .htaccess files only apply to the directory in which the file is located. As with the **<Directory>** directive, the directives within an .htaccess file will apply to the specified directory and all files and directories within it.

The second difference between .htaccess files and **<Directory>** directives is that, using the **AllowOverride** directive, the administrator can specify which directives a .htaccess file may contain, or whether the file is even consulted. **AllowOverride** is a valid directive in **<Directory>** and **<DirectoryMatch>** container directives, and administrators are able to use this to control the permitted directives in .htaccess files on a directory by directory basis. The directive may have one or more of the following parameters:

- ***All***—Indicates that all directives that are valid within the ".htaccess" context are allowed within the specified .htaccess files. (No other parameters should be listed.)
- ***None***—Indicates that .htaccess files should not be consulted by the server. The files will not even be read. (No other parameters should be listed.)
- ***AuthConfig***—Permits the .htaccess file to contain directives relating to authentication. (See Section 3.2 for more on authentication directives.) These directives include **AuthName**, **AuthType**, **require**, and all directives contained within the *mod_auth*, *mod_auth_db*, *mod_auth_dbm*, *mod_auth_anon*, and *mod_digest* modules.
- ***FileInfo***—Permits the .htaccess file to contain directives controlling document types. These directives include those in the *mod_mime* module.
- ***Indexes***—Permits the .htaccess file to contain directives relating to directory indexing. These directives include those in the *mod_autoindex* module.

- ***Limit—***Permits the .htaccess file to contain directives relating to host based access control. These directives include **order**, **deny**, and **allow**.
- ***Options—***Permits the .htaccess file to contain directives relating to directory options. These directives include the **Options** and **XBitHack** directives. Administrators should be wary of providing this parameter as the directives it describes, as it can grant some powerful capabilities.

When filling out a .htaccess file, one simply lists the directives that one wishes to have applied. Only specific directives are permitted within .htaccess files. Even though the **<Directory>** directive behaves in a similar way, not all directives that are appropriate within a **<Directory>** container will be permitted within a .htaccess file. The **AllowOverride** directive is a notable example that is acceptable in the former but which is disallowed in the latter. The documentation for individual directives will specify whether the directive is allowed in the "directory" and/or the ".htaccess" contexts. In addition, administrators must be careful not to place directives in an .htaccess file that are prohibited by the **AllowOverride** directive. If the .htaccess file contains illegal directives, when the user attempts to access it or any file or directory within it, the Apache server will return a "Internal Server Error" message (HTTP code 500) to the client that explains that an illegal directive was in the .htaccess files and lists the offending directive.

One other note is that, unlike the standard three configuration files, the Apache server does not need to be restarted to implement changes to .htaccess files. Modifications to .htaccess files take effect the moment the changes are saved.

3.2 Authentication

Authentication is the procedure by which the server attempts to verify the identity of a client through the exchange of information. This is accomplished when the client sends the server a recognized username and password pair. The Apache Web Server allows administrators to set up access control policies based on the authenticated identities of users. The opposite of authenticated access is anonymous access wherein no authentication information is transferred. This is the normal behavior of web servers. This section will offer a brief description of the authentication capabilities of the Apache Web server as well as the steps to configure the two most common authentication setups: username-password authentication and anonymous authentication.

3.2.1 Modules

The Apache Web Server Version 1.3.3 contains five modules that relate to authentication. These modules are:

- *mod_auth*—Provides username authentication capabilities. The usernames and passwords are stored in a plain text file on the server with the password encrypted using the UNIX `crypt` function. The client sends the username and password over the network in uuencoded format.
- *mod_auth_db*—Provides username authentication capabilities. The usernames and passwords are stored in a Berkeley-DB database on the server with the password encrypted using the UNIX `crypt` function. The client sends the username and password over the network in uuencoded format.
- *mod_auth_dbm*—Provides username authentication capabilities. The usernames and passwords are stored in a DBM database on the server with the password encrypted using the UNIX `crypt` function. The client sends the username and password over the network in uuencoded format.
- *mod_digest*—Provides username authentication capabilities. The usernames and passwords are stored in a plain text file on the server with the password hashed using the MD5 message digest algorithm. The client sends the username and password over the network after it is hashed using MD5.
- *mod_auth_anon*—Provides anonymous authentication capabilities. The client must enter one of the specified "anonymous" usernames and be recognized as a valid user (with some additional control provided by the modules' directives). The client sends the username, and possibly an e-mail address as a password, over the network in uuencoded format.

Most of the modules are virtually identical. Specifically, the *mod_auth*, *mod_auth_db* and *mod_auth_dbm* modules contain the same three directives with only slightly different names. (I.e., **AuthUserFile**, **AuthDBUserFile**, and **AuthDBMUserFile**.) The functionality of these three modules is identical except in respect to the format of the file in which the username and password data is stored. This document will use the *mod_auth_db* module for its username-password authentication; it is more efficient than straight text files and is slightly easier to configure than DBM databases using the tools packaged with Apache. If the administrator wishes to use one of the other two modules, the directives can be converted in a straightforward manner and the username-password files rewritten in the appropriate format. The description of a username-password configuration will include additional instructions for the use of other modules when necessary.

The *mod_digest* module implements MD5 message digest authentication. This tells the client to send the username and password to the server using an MD5 hash. While this is much more secure against eavesdropping than normal uuencoding, the technique does have some security problems as described in RFC 2831. Of course in order use digest authentication, a compatible web browser must be used. At this time only the latest releases of Internet Explorer (version 5.5) and Netscape Communicator (version 6.0) implement Digest authentication.

The *mod_auth_anon* module is used for anonymous authentication. Anonymous authentication should be distinguished from anonymous access. In the former, the user must undergo some log-in process, even though it does not attempt to identify the user. Anonymous access has no log-in process at all. Usually anonymous authentication behaves similarly to anonymous FTP login wherein the user of the client browser specifies "anonymous" (or some other widely recognized username) as their username and then provides their E-mail address in the password field. The *mod_auth_anon* module provides several directives which allow the administrator to specify what information needs to be entered in the log-in panel for an anonymous authentication to succeed. This includes the ability to specify the contents of the username field and, in a more limited sense, the contents of the password field as well.

The *mod_auth*, *mod_auth_db*, *mod_auth_dbm*, and *mod_auth_anon* modules all contain some variation of "authoritative" directives. This paper will not cover these directives for reasons that will be explained in Section 3.2.6.

3.2.2 Default Configuration

The *mod_auth* module is part of the default Apache build. However, the default configuration files contain no authentication directives.

3.2.3 Background Information

This section discusses information that is not a straightforward part of the configuration process. This information is deemed necessary for a proper understanding of how authentication is implemented.

3.2.3.1 Username-Password File Creation

The username-password files contain a list of valid username and password pairs that may be provided by a client and recognized by the server. They are created using tools distributed with Apache. (1) To create a plain text username-password file, such as would be used by the *mod_auth* module, the `htpasswd` command would be used. (2) To create a Berkeley-DB database for the *mod_auth_db* module, the `dbmmanage` utility with the `adduser` option would be used. (3) To create a DBM database for the *mod_auth_dbm*, the `dbmmanage` utility with the `adduser` option would be used. The resulting file then needs to be renamed so it has a .db extension. (4) Password files for the *mod_digest* module can be created using the `htdigest` utility. All three of these utilities contain `man` pages which describe how to use them.

The passwords associated with usernames should not be the same passwords used to grant access to other services on the system. In all but the *mod_digest* module, the passwords are sent over the wire in uuencoded format, which is trivial to decode. This allows anyone capable of listening to the wire, the capability of learning the username-password pairs used to log onto the system. Additionally, the server does not keep track of failed authentication attempts, allowing an attacker any number of guesses when trying to gain access. For these reasons, it is important that web passwords not be used with other resources in order to localize any possible damage when these passwords are revealed.

3.2.3.2 Username-Password File Security

All implementations of authentication must have a reference to a username-password file. The security of the username-password files is very important. With the exception of files created using the `htdigest` utility, which can only be used with the *mod_digest* module, the format of the username-password files provide no security. Any user who can read these files will, at the very least, be able to launch an off line dictionary attack to crack the passwords. For this reason, it is important that access to the files is provided only to the people and applications that need it.

The Apache server reads the username-password files in order to perform authentication. The Apache server runs under the user and group specified respectively in the **User** and **Group** directives in its configuration files (see Section 3.1.2). The username-password file must be readable by this user or group.

The web site administrator is the only other user who needs access to these files. If the administrator is the root operator of the Linux host, they will have access to the file no matter what the security settings are. In this case, only the Apache server user ID should have access. If the web server administrator uses a different user ID, it is necessary that the security settings of the file be set to give them both read and write access to the username-password files. This should be verified for every username-password file created since it is unlikely that the Linux operating system will create files with these characteristics by default.

Additionally, the location of the username-password files is important. Under no circumstances should the files be placed in a published directory of the web server since this will give remote users the ability to download them. Username-password files should be placed in a location that both the Apache Web Server and the web server administrator are able to reach, but that is not part of the servers published path.

3.2.3.3 Group Files

Since an administrator may wish to grant access to a list of authenticated users, the Apache server provides the ability to group users together. This alleviates the need to enter long lists of usernames into the configuration file. Unlike the username-password files, group files are optional and added only for the convenience of the administrator.

The method to create group files depends on the authentication module used. For the *mod_auth_db* and *mod_auth_dbm* modules, the `dbmmanage` utility with the `add` option is used. For the *mod_auth_dbm* module, the created file needs to be renamed with a .db extension. When using the `dbmmanage` utility in both cases, the username is given as the key and a comma separated list of groups (no spaces) is given as the "password."

There is no utility to create group files for the *mod_auth* module. However, the format of the file is simple enough that it can be easily created using a simple text editor. Each line of the file should contain the name of the group followed by a colon (no space) followed by a space separated list of the users who are members of this group. The file should be created using a text editor which will not add formatting marks to the document. Both `emacs` and `vi` are suitable for the task.

The *mod_digest* module does not support group authentication in the current version of Apache.

The group files, unlike the username-password files, do not contain sensitive information. However, the same comments made about the security of the username-password files apply to the group files. This will prevent users from altering the files and changing the security policy of the web server.

3.2.4 Configuration Information

There are two types of authentication that can be configured. The first is username-password authentication in which a client provides a username and secret password to the server in order to verify the identity of the person using the client browser. Anonymous authentication allows the user to enter a well-known username and their e-mail address as a password and be given access to the system. The E-mail address can optionally be recorded in a log file. The configuration commands for these authentication scenarios is described below.

3.2.4.1 Username-Password Authentication

This example of username-password authentication uses the *mod_auth_db* module. To create configuration entries for the *mod_auth* and *mod_auth_dbm* modules, simply interchange the directives with their corresponding directive in the desired module. If additional changes are necessary, these will be mentioned as appropriate. The format of the username-password and group files must be appropriate to the module being used.

Since this example uses *mod_auth_db* which is not, by default, compiled into the Apache server, the parameter `--enable-module=auth_db` must be given as an argument of the `configure` command. If *mod_auth_dbm* is used, the argument would be `--enable-`

`module=auth_dbm`. Since the *mod_auth* module is enabled by default, it does not require extra arguments in the `configure` command.

```
<Directory "/usr/local/apache/share/httpd/">
AuthType Basic
AuthName "Sample Server"
AuthDBUserFile "/usr/local/apache/etc/userfile1"
AuthDBGroupFile "/usr/local/apache/etc/groupfile1"
AuthDBAuthoritative on
require group group1 group2
</Directory>
```

Figure 7. Configuration File Example for mod_auth_db

Figure 7 shows an excerpt from a configuration file. The given directives enable username-password authentication for the target directory using the *mod_auth_db* module. The following describes each line in the above figure:

`<Directory /usr/local/apache/share/httpd/>`

Specifies the resource that requires authentication. Any of the container directives may be used for this purpose (see Section 3.1.1).

`AuthType Basic`

Authentication is either “Basic” or “Digest.” “Basic” authentication must be used for all modules except *mod_digest*. This information is passed on to the client and instructs it on how to format its authentication information.

`AuthName Sample Server`

This specifies a string sent to the client. It is provided to the client so they can know the resource to which they are authenticating. In this case, the client would be prompted with the string “Enter username for Sample Server at BATMAN.G021LAB.ORG” where “BATMAN.G021LAB.ORG” is the name of the machine running the web server.

`AuthDBUserFile /usr/local/apache/etc/userfile1`

This specifies the path and name of the username-password file. It must be readable by the Apache server user ID. It must also be in the appropriate format. NOTE: If **AuthDBMUserFile** was used, the filename of the database is listed *without* the .db extension even though the file itself *must* have this extension. The DBM library automatically appends a .db extension to the filename specified in the directive before looking for the file.

`AuthDBGroupFile /usr/local/apache/etc/groupfile1`

This specifies the path and name of the group file. It must be readable by the Apache server. It must also be in the appropriate format. If there is no group file, this directive should be omitted. NOTE: If **AuthDBMUserFile** was used, the filename of the database is listed *without* a .db extension even though the file itself *must* have this extension. The DBM library automatically appends a .db extension to the filename specified in the directive before looking for the file.

`AuthDBAuthoritative on`

The authoritative directives are used to control the interaction of several authentication methods covering a single resource. This is a subject is complicated and, for this reason, we recommend all resources have only a single authentication method applied to them. Setting the “authoritative” directive to “on” states that this authentication method has the final say in terms of which users are recognized.

`require group group1 group2`

This directive specifies the users and/or groups which, once authenticated, are permitted to access the specified resource. When a user provides a recognized username and password, he will not be allowed access unless the username or associated group are listed in the **require** directive. The first parameter is either “user,” “group,” or “valid-user.” If the first parameter is “user,” the subsequent parameters are a space separated list of the users which will be allowed access to the resource. If the first parameter is “group,” the subsequent parameters are a space separated list of the groups whose members will be allowed access. If the first parameter is “valid-user” then there will be no further parameters and any recognized username-password pair will be allowed access.

`</Directory>`

This directive is used to close off the context block that started this example.

Behavior: Using the example above, the behavior will be as follows. When a client attempts to access any resource within the directory "/usr/local/apache/share/httpd/" (or its sub directories), the server returns an "Authentication Required" error (error code 401), if it does not provide a username-password pair, or it provides an invalid username-password pair. On most browsers, this causes a panel to pop up on the client browser where the user can enter their username and password. This is returned to the server. The server first consults the username-password file (as specified in the "UserFile" directive) to see if the username-password pair is recognized. If it is not recognized, authentication fails. Otherwise, it consults the **require** directive to see who is allowed access. If the first parameter is "valid-user," access is allowed. If the first parameter is "user," and the username provided by the client matches one of the subsequent require parameters, then access is allowed. If the first parameter is "group," then the group file is consulted (as specified in the "GroupFile" directive). If the authenticated user belongs to the appropriate group, then access is granted. Otherwise, access is denied.

If access is granted, then the requested resource is returned to the client. If access is denied, the server returns an "Authentication Required" error (error code 401). Most browsers will pop up a prompt that informs the client user that authentication failed and give the user the opportunity to try again.

The directive block in the example would override any directive blocks placed on previous directories. It would affect all the contents and subdirectories of the specified directory unless these, in turn, contained their own set of authentication directives. The rules by which individual authentication directives are overridden by others is complicated; for safety it is recommended that every directive of an enclosing block be explicitly overridden by the contents of any new block. This means that every block which specifies authentication should include the **AuthType**, **AuthName**, and **require** directives, as well as some module's set of "UserFile," "Authoritative," and, if necessary, "GroupFile" directives. If the sub block does not use the group file for its access control decision, it is not necessary to override any "GroupFile" directives in the parent block.

Authentication directives do not overlap but instead replace each other. That is, if a parent block will only allow access to an authenticated user X, and a sub block will only allow access to an authenticated user Y, then accessing the sub block will only require authenticating as Y.

3.2.4.2 Anonymous Authentication

The anonymous authentication functionality is provided by the *mod_auth_anon* module. This module is not part of the default Apache build, it must be enabled by including the parameter `--enable-module=auth_anon` in the `configure` command when Apache is being built.

The following figure is an excerpt from a configuration file which implements anonymous authentication.

```
<Directory "/usr/local/apache/share/httpd/">
AuthType Basic
AuthName "Sample Server".
Anonymous anonymous guest
Anonymous_Authoritative on
Anonymous_LogEmail on
Anonymous_MustGiveEmail on
Anonymous_NoUserID off
Anonymous_VerifyEmail off
require valid-user
</Directory>
```

Figure 8. Configuration File Example for Anonymous Authentication

The example shown in Figure 8 is described in the following paragraphs.

`<Directory /usr/local/apache/share/httpd/>`

Same as username-password authentication.

`AuthType Basic`

Specifies the type of authentication being performed. Anonymous authentication only supports the “Basic” type.

`AuthName Sample Server`

Same as username-password authentication.

`Anonymous anonymous guest`

Specifies the usernames which will indicate anonymous authentication is being attempted. The usernames are presented in a space separated list after the directive.

`Anonymous_Authoritative on`

Specifies that this authentication module makes the final decision for user authentication.

`Anonymous_LogEmail on`

Specifies that the value written into the password field of an authentication attempt is to be recorded in the error log. It is an “info” level event, so the log level must be set appropriately (see below). If the parameter was “off” then this information would not be retained.

`Anonymous_MustGiveEmail on`

Specifies that the client must provide a value in the password field for anonymous authentication to succeed. If the parameter was set to “off” then the password field could be left empty.

`Anonymous_NoUserID off`

Specifies that a username must be provided when attempting anonymous authentication. If the parameter was set to “on” then a blank username field would also indicate an anonymous authentication attempt.

`Anonymous_VerifyEmail off`

Specifies that no checks are performed on the contents of the password field beyond making sure that it is not empty. If the parameter was set to “on” then the server would check to make sure that the password field contained at least one “@” and at least one “.” since these would be present in any valid E-mail address.

`require valid-user`

Specifies that if an anonymous authentication attempt passes all of the above tests, then the user is to be given access. The **require** directive must have the “valid-user” parameter when anonymous authentication is used. It is not possible to specify a “user” listed after the **Anonymous** directive.

`</Directory>`

Closes off the context block which begins this example.

```
LogLevel info
```

Specifies that the error log should record events of importance “info” and higher. This configures the log so that it will record the E-mail address provided by a client in the password field. If the log level were left at a higher level, then the error log would consider the events which record the E-mail address to be too minor to record and the information would be lost. (For more on the **LogLevel** directive, see Section 3.4.3.1.) Note that **LogLevel** is part of the *core* module and is always available.

Behavior: The behavior of the above example is detailed as follows. This scenario operates similar to the username-password configuration given above. The only difference would be that, instead of providing a username and a secret password, the client would provide either “anonymous” or “guest” as the username, and then enter their E-mail address in the password field. Authentication would only fail if a username other than “anonymous” or “guest” was provided, or if one or more of the username-password fields were left blank in the log-in prompt. If the **Anonymous_VerifyEmail** directive was set to “on,” anonymous authentication would also fail if the password field did not contain a “.” and “@”. As before, a successful authentication would cause the server to return the requested resource while a failed authentication would cause the server to send an “Authentication Required” error (error code 401).

As described in the username-password authentication example, this authentication block will effectively override all authentication directives present in a higher level directory. Likewise, the block will cover all resources in the specified directory and its sub-directories unless specifically overridden by another block of container directives. The fact that anonymous authentication and username-password authentication contain few directives which correspond to each other is not a problem—simply specify all directives of the new module when changing between them.

If the **Anonymous_LogEmail** directive is set to “on,” then every time a client attempts an anonymous authentication, the server will create an error log event with priority “info” which includes whether or not the server accepted the authentication attempt and the value of the password field. This includes a log event with a priority of “info” means that, if the log level (set by the **LogLevel** directive—see Section 3.4.3.1) is set at a priority above “info” (as it is by default) then the event will not actually get written to the log file.

The E-mail address returned during anonymous authentication should not be trusted. There is no way to verify that the E-mail address presented is the users actual E-mail address. Even the **Anonymous_VerifyEmail** directive only provides the most cursory check of the fields’ contents.

3.2.5 Synopsis and Recommendations

All modules except *mod_digest*, transmit password information in a highly insecure format. For this reason, the authentication mechanism is of limited use in controlling the dispersal of the web sites' contents. Since Apache cannot encrypt content, anyone who could watch the wire for a username-password pair could also simply read the resources returned by an authenticated request from a valid client. It is recommend that the username-password pairs do not apply to any services other than the web server.

Although Apache does contain the capability to use multiple authentication modules to control access to a single resource, this type of configuration should be left to experienced Apache administrators. The mechanics of which module takes precedence over another are not straightforward and beyond the scope of this paper. Likewise, it is recommended that, each block of authentication directives should explicitly contain all authentication directives contained in the module being used and not just the directives that have different parameters from a previous block.

Finally, it is important to understand that the authentication mechanism does not imply more security than has been described above. If a user successfully authenticates, this does not imply that the remainder of the communication session will be protected in any way. The way the mechanism works is that it indicates that the client knew a recognized and required username and password pair (assuming username-password authentication was used). While this does provide some security, administrators need to be aware that its control of content distribution, especially given the lack of any confidentiality service, can be overcome using relatively simple techniques.

3.2.6 Additional Topics

This section describes features of the Apache authentication mechanism that were not covered in the above description. Generally, these topics were skipped because they were unnecessary for the implementation of a secure web server. They have been included here to inform the reader that the additional functionality and options are in fact available should the reader wish to pursue them.

- *Using Multiple Modules to Control Authentication to a Single Resource*—It is possible to use instances of any and all the authentication modules with the exception of *mod_digest* to control authentication to a single resource. This can be done if username-password files already exist in different formats or if an administrator wishes to implement both username-password and anonymous authentication simultaneously. The "Authoritative" directives are used to control this behavior.
- *Directive Inheritance*—It is not strictly necessary to repeat all authentication directives whenever the authentication policy changes. There are actually cases where it is more convenient to override a single directive. However, not all directives

can be overridden in the same way. For example, overriding any directive in an authentication module causes *all* directives in that module to be overridden. For this reason, unless the administrator understands the ramifications, this document recommends complete override of previous directives.

3.3 Access Control

Access control provides the ability to grant access to system resources based on the identity of the client. Apache supports the functionality to set up host-based access control where the requesting host is either allowed access or denied access to particular resources on a web server. When a request is made for a particular resource, Apache checks to see if the requestor is allowed access to the requested resource.

3.3.1 Modules

The Apache Web Server version 1.3.3 contains one module that provide access control functionality:

- *mod_access* – The *mod_access* module provides host-based access control. It is based upon the client IP address or hostname. The directives contained in this module are: **allow, allow from env=, deny, deny from env=,** and **order.**

3.3.2 Default Configuration

By default, the *mod_access* module is enabled. The default Apache configurations files contain several active directives from the *mod_access* module. It also contains several directives that have been commented out, but which can be uncommented for swift implementation. The directives in the default Apache configuration files are provided below in Figure 9. This example is detailed in the following paragraphs.

```
<Directory "/home/httpd/htdocs">
AllowOverride None
Order allow, deny
Allow from all
</Directory>
```

Figure 9. Configuration File Example for Access Control.

`<Directory "/home/httpd/htdocs">`

This directive specifies the resource container that requires access control. Any of the container directives such as Directory, Location, Files may be used for this purpose. The container specified is the */home/httpd/htdocs* directory.

```
AllowOverride None
```

This directive tells the server which directives that have been declared in a .htaccess file can override earlier configuration directives. This directive can be set to **None**, **All**, **AuthConfig**, **FileInfo**, **Indexes**, **Limit** or **Options**. In this particular case the **None** specifies that no options within this directory block can be overwritten by a local access control file. This means that the server does not have to look for an access file for each request. See section 3.1.5.

```
Order allow,deny
```

This directive controls the order in which allow and deny directives are evaluated. There are several orders that can be specified; these are: **allow,deny**, **deny,allow**, and **mutual-failure**. In this case the **allow,deny** specifies that the allow directives are evaluated before the deny directives; the initial state is to deny all access.

```
Allow from all
```

This directive affects which hosts can access the specified container. This directive can specify the hosts in several different ways; hostname, IP address, and partial IP, domain-name addresses. The **all** specified in this case refers to all hosts; which means that all hosts can access this container.

```
</Directory>
```

This directive is used to close off the context block that started this example.

These example statements are contained in the default configuration file for the Apache server. The default settings allow every host access to the *…/htdocs/* container on the server. It also specifies that there are no .htaccess overides for this container. These statements can be changed to customize the Apache server access control. The following sections detail the available options.

3.3.3 Background Information

This section discusses information that is not contained in the default server configuration files. It is necessary if customizing the access control mechanism on the server.

3.3.3.1 Access Control Scope

The access control directives Allow, Deny, Order etc. provide access control at a number of levels within the server. These directives can be used within all the resource containers on the server. The standard container include: **<Virtual Host>**, **<Directory>**, **<DirectoryMatch>**, **<Files>**, **<FilesMatch>**, **<Location>**, **<LocationMatch>**, and **<Limit>**. The Apache server can be configured to have very fine tuned access control depending upon the use of the directives within specific containers. Access can be controlled per directory using the **<Directory>** context, per file using the **<File>** context, per URL location using the **<Location>** context and per HTTP request method using the **<Limit>** context. The **<Limit>** context has the narrowest scope of all containers.

3.3.4 Configuring Custom Access Control

The Apache server provides the ability to define custom access control. This is controlled through the directives provided in the *mod_access* module. Access control can be set from open to very restrictive depending on the customizations using the mod_access directives.

3.3.4.1 Allow Directive

The allow directive defines the hosts that are allowed to access a particular container. The directive syntax is : **allow from** *host* where *host* can be specified using several different methods; these are detailed below:

- <u>all</u> - this would allow all hosts access to the container; ex: **allow from all**
- <u>a partial domain name</u> - host names that match or end in a particular string are allowed access; ex: **allow from .mitre.org**
- <u>a full IP address</u> - an IP address of a host that is allowed access; ex: **allow from 129.83.40.1**
- <u>a partial IP address</u> - the first 1 to 3 bytes of an IP address of hosts that are allowed access, this is used for subnet restriction; ex: **allow from 129.93.40**
- <u>a network/netmask pair</u> - a network a.b.c.d and a netmask w.x.y.z pair of hosts that are allowed access, this allows fine-grained subnet restriction; ex: **allow from 129.83.40.0/255.255.255.0**
- <u>a network/nnn CIDR specification</u> - a network a.b.c.d address and a netmask that consists of nnn high order bits to specify hosts that are allowed access; ex: **allow from 129.83.40.0/24**

3.3.4.2 Deny Directive

The deny directive defines the hosts that are not allowed to access a particular container. The directive syntax is: **deny from** *host* where *host* can be specified using the methods desribed previously for the allow directive:

- <u>all</u> - this would allow all hosts access to the container; ex: **allow from all**

- a partial domain name - host names that match or end in a particular string are allowed access; ex: **allow from .mitre.org**
- a full IP address - an IP address of a host that is allowed access; ex: **allow from 129.83.40.1**
- a partial IP address - the first 1 to 3 bytes of an IP address of hosts that are allowed access, this is used for subnet restriction; ex: **allow from 129.93.40**
- a network/netmask pair - a network a.b.c.d and a netmask w.x.y.z pair of hosts that are allowed access, this allows fine-grained subnet restriction; ex: **allow from 129.83.40.0/255.255.255.0**
- a network/nnn CIDR specification - a network a.b.c.d address and a netmask that consists of nnn high order bits to specify hosts that are allowed access; ex: **allow from 129.83.40.0/24**

3.3.4.3 Order Directive

The order directive controls the order that Apache uses to evaluate the **allow** and the **deny** directives. There are three order options that can be used for evaluation. The directive syntax is: **order** *ordering* where *ordering* is one of the following:

- allow,deny – the allow directives are evaluated before the deny directives (the initial state is deny); ex: **order allow, deny**
- deny,allow – the deny directives are evaluated before the allow directives (the initial state is allow); ex: **order deny, allow**
- mutual-failure – only the hosts that appear on the allow list and do not appear on the deny list are granted access.

3.3.4.4 Allow From Env Directives

The **allow from env** directive controls access to the specified container by using environmental variables. The directive syntax is: **allow from env=***env* where *env* is an environmental variable that has been set using these environmental variables are defined using other directives such as **BrowserMatch**. An example is as follows:

```
BrowserMatch  "MSIE 4.01"  let_me_in
<Directory /apache/test>
  order deny,allow
  deny from all
  allow from env=let_me_in
</Directory>
```

In this case any browser using MSIE 4.01 will be allowed access to the /apache/test directory.

3.3.4.5 Deny From Env Directives

The **deny from env** directive controls access to the specified container by using environmental variables. The directive syntax is: **deny from env=***env* where *env* is an environmental variable that has been set using these environmental variables are defined using other directives such as **BrowserMatch**. An example is as follows:

```
BrowserMatch  "MSIE 4.01"  keep_me_out
<Directory /apache/test>
  order deny,allow
  allow from all
  deny from env=let_me_out
</Directory>
```

In this case any browser using MSIE 4.01 will be denied access to the /apache/test directory.

3.3.5 Implementation of Customized Access Controls

Implementation of customized access controls is shown in Figure 2 below. A detailed description of these access control commands is included below.

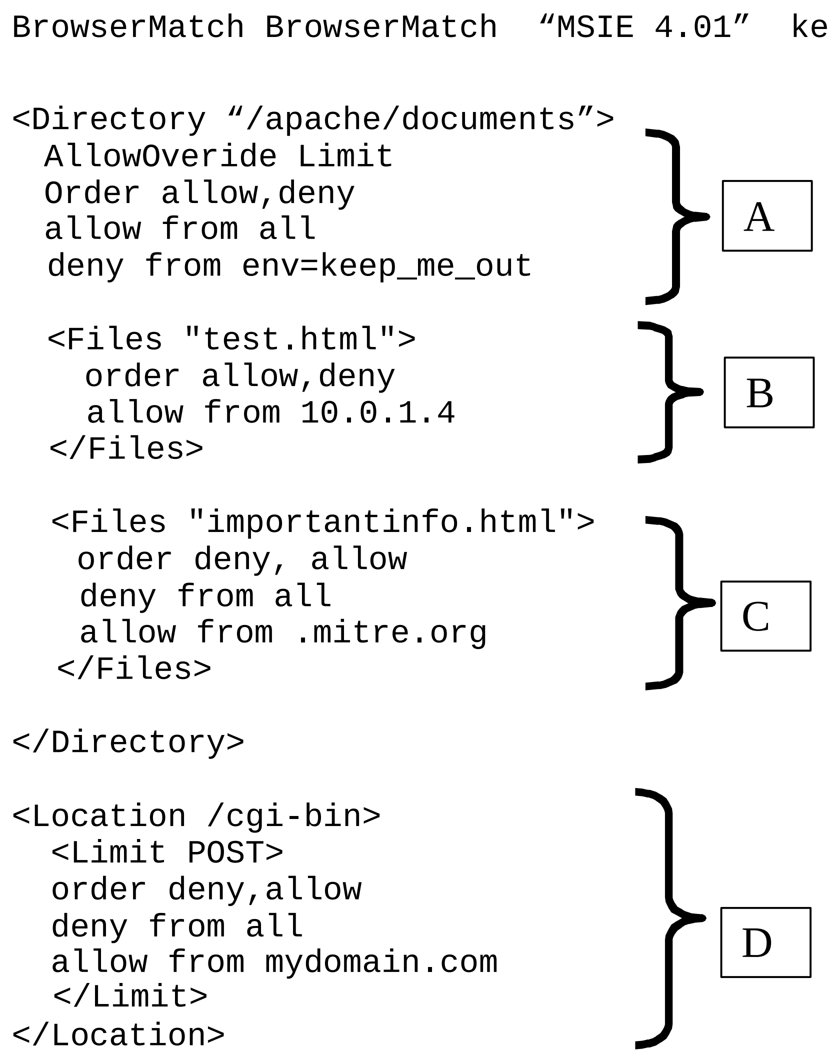
The **AccessFileName** directives designates the .htaccess file as the access control file to look for within each directory. The **BrowserMatch** directive is used to set the environmental variable used for the **deny from env=** directive.

A. This set of commands is set for the directory container “/apache/documents”. The **AllowOveride** directive gives the “/apache/documents/.htaccess” file the ability to override the **<Limit>** directives set earlier for this directory. The **Order** directive sets the initial state as forbidden , the allow directives are evaluated before the deny directives. Everyone is allowed access to this container except browsers using MSIE 4.01; this is set using the **deny from env=** and the **BrowserMatch** directives.

B. This set of commands is specific to the *test.html* file contained in the “/apache/documents” container. Only the host matching the IP address 10.0.1.4 can access this particular file.

C. This set of commands is specific to the *importantinfo.html* file contained within the “/apache/documents” container. Access to this file is denied to all except those that are part of the .mitre.org domain. The **<Directory>** directive closes the context block for the “/apache/documents” container.

D. This context block refers to the /cgi-bin location. Within this container the **<Limit>** directive is used to enclose a group of access control directives that apply to the HTTP access method POST. These commands specify that only hosts from .mydomain.com can use the POST method within the /cgi-bin location.



```
BrowserMatch BrowserMatch  “MSIE 4.01”  keep me out

<Directory “/apache/documents”>
  AllowOveride Limit
  Order allow,deny
  allow from all
  deny from env=keep_me_out

  <Files "test.html">
    order allow,deny
    allow from 10.0.1.4
  </Files>

  <Files "importantinfo.html">
   order deny, allow
   deny from all
   allow from .mitre.org
  </Files>

</Directory>

<Location /cgi-bin>
  <Limit POST>
  order deny,allow
  deny from all
  allow from mydomain.com
  </Limit>
</Location>
```

E.

Figure 2. Access Control Customizations

3.3.6 Synopsis and Recommendations

Apache access control provides the ability to grant access to specific system resources based on the identity of the host. When a request is made for a specific resource, Apache checks to see if the requestor is allowed access to the requested resource. As shown previously, access control can be set from open to very restrictive depending on the customizations using the *mod_access* directives. There are several recommendations for access control settings on the Apache web server. These are:

- Users should be stopped from overriding system wide settings; this is done by stopping users from being able to set up .htaccess files that override security settings configured for the server. Likewise, default access should be disabled; only permit specific access to specific locations. This is done as follows:

```
<Directory  />
   AllowOverride None
   Options None
   Order deny,allow
   Deny from all
</Directory>
```

- The interactions of **<Location>** and **<Directory>** directives should be carefully monitored. A **<Directory>** might deny access but a **<Location>** directive could overturn it.

3.4 Auditing

The Apache auditing mechanism allows an administrator to record the activities on the Apache server. By observing the log files, administrators can determine site traffic, requested resources, resource retrieval method and whether or not the server experienced difficulties servicing the request. It can help determine whether a client is attempting to perform illicit activities on the server and can give some insights as to the nature of an attack if a violation occurs. For this reason, the auditing mechanism is of great importance to the security of the server. This section will discuss the auditing mechanism implemented by Apache and the methods used to create custom audit files.

3.4.1 Modules

The Apache Web Server Version 1.3.3 contains six modules which relate to recording usage information. Of these six, two have been deprecated and should not be used. The modules are:

- *core*—The main Apache module. This contain directives concerning the error log which records problems and events experienced by the Apache server independent of requests made by clients.
- *mod_cookies*—Deprecated. Do not use.
- *mod_log_agent*—Use to record information about the nature of a client making a request.
- *mod_log_common*—Deprecated. Do not use.

- *mod_log_config*—The primary auditing module, this module allows an administrator to create audit facilities which will record wide range of information concerning client requests.
- *mod_log_referer*—Used to record information concerning what resource provided the link to the resource being requested.
- *mod_usertrack*—Allows the administrator to track the activities of a client when accessing the server

The functionality of the two deprecated modules has been replaced and expanded by other auditing modules (*mod_log_config* replaces *mod_log_common* while *mod_usertrack* replaces *mod_cookies*). The deprecated modules should never be used and, henceforth, will not be mentioned again.

All of the functionality of *mod_log_agent* and most of the functionality of *mod_log_referer* can be duplicated through specially formulated directives in the *mod_log_config* module. The only differences are minor variations in how the output data is formatted. For this reason, both *mod_log_agent* and *mod_log_referer* will have a brief explanation. The emphasis of this section is on the *mod_log_config* module.

The *mod_usertrack* module tells the Apache server to send identification cookies to clients making requests for the first time. These cookies will then be sent with every subsequent request from that client; providing the client is configured to return cookies. Using the *mod_log_config* module, these cookies can be recorded and used to track the patterns of usage on the server.

3.4.2 Default Configuration

By default, the *mod_log_config* module is enabled. The default Apache configurations files contain several active directives from both the *core* and *mod_log_config* modules. It also contains several directives that have been commented out, but which can be uncommented for swift implementation. The pertinent directives in the default Apache configuration files are provided as an example in figure 9. This example is described in detail in the following paragraphs.

```
ErrorLog /usr/local/apache/var/log/error_log
LogLevel warn
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
\"%{User-Agent}i\"" combined
     <ip of client> <remote logname> <username> <access time>
     "<first line of request>" <final status> <bytes sent>
     "<Referer field>" "<User-Agent field>"
LogFormat "%h %l %u %t \"%r\" %>s %b" common
      <ip of client> <remote logname> <username> <access time>
      "<first line of request>" <final status> <bytes sent>
LogFormat "%{Referer}i -> %U" referer
      <Refer field> -> <URL requested>
LogFormat "%{User-agent}i" agent
      <User-agent field>
 CustomLog /usr/local/apache/var/log/access_log common
```

Figure 9. Configuration File Example for Logging

`ErrorLog /usr/local/apache/var/log/error_log`

This line sets the location and name of the server error log.

`LogLevel warn`

This line sets the error log to report incidents of ***warn*** priority or higher. See Section 3.4.3.1 for more concerning what events each level record.

`LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined`

This line defines a log format and gives it the nickname “combined” for later use. This format definition would produce an event with the following form and information:

`<ip of client> <remote logname> <username> <access time> "<first line of request>" <final status> <bytes sent> "<Referer field>" "<User-Agent field>"`

`LogFormat "%h %l %u %t \"%r\" %>s %b" common`

This line defines a log format and gives it the nickname "common" for later use. (This format represents Common Log Format.) This format definition would produce an event with the following form and information:

```
<ip of client> <remote logname> <username> <access time> "<first line of
request>" <final status> <bytes sent>
```

`LogFormat "%{Referer}i -> %U" referer`

This line defines a log format and gives it the nickname "referer" for later use. (The resulting format is the same as that provided by the *mod_log_referer* module.) This format displays the following fields on each line:

```
<Refer field> -> <URL requested>
```

`LogFormat "%{User-agent}i" agent`

This line defines a log format and gives it the nickname "agent" for later use. (The resulting format is virtually the same as that provided by the *mod_log_agent* module.) This format displays the following field on each line:

```
<User-agent field>
```

`CustomLog /usr/local/apache/var/log/access_log common`

This line creates a new log file with the format associated with the "common" nickname. The new log file is named "access_log" and is located in the `/usr/local/apache/var/log/` directory.

`#CustomLog /usr/local/apache/var/log/referer_log referer`

This line creates a new log file with the format associated with the "referer" nickname. The new log file is named "referer_log" and is located in the `/usr/local/apache/var/log/` directory. The line is commented out so, by default, this log file is not created.

`#CustomLog /usr/local/apache/var/log/agent_log agent`

This line creates a new log file with the format associated with the "agent" nickname. The new log file is named "agent_log" and is located in the `/usr/local/apache/var/log/` directory. The line is commented out so, by default, this log file is not created.

```
#CustomLog /usr/local/apache/var/log/access_log combined
```

> This line creates a new log file with the format associated with the "combined" nickname. The new log file is named "access_log" and is located in the **`/usr/local/apache/var/log/`** directory. The line is commented out so, by default, this log file is not created.

By default, the Apache server creates two log files: The error log and the access log. The error log, named `error_log`, is part of the Apache *core* module and records events on the server independent of client requests. It is initially set to record events of level ***warn*** or higher. (See Section 3.4.3.1 concerning error log event levels.) The access log, named `access_log`, records information about user requests in Common Log Format. The remaining directives are included to ease implementation of additional logs.

3.4.3 Background Information

This section discusses information which is not a straightforward part of the configuration process. It is necessary if customizing the auditing mechanism.

3.4.3.1 The Error Log

The error log is used by the server to record information concerning its own general functioning. The **ErrorLog** directive is part of the *core* Apache module so it is always available. In fact, due to the importance of the error log, the Apache server will not start unless it can open the error log.

There are eight levels of events which can be written to the error log. The **LogLevel** directive (also in the *core* module) sets the threshold level below which events will be ignored. The error log levels and a few examples of what situations will raise events are listed below:

emerg—Emergencies which make the server unusable.

- Errors which result from the Apache server not being able to find functions with which it should have been compiled with.

alert—Situations which require prompt attention.

- Problems implementing the **User** and/or **Group** directives controlling the ID under which child processes run.
- A child process experienced a fatal error.

crit—Situations of critical importance.

- Failure to create or send data over a socket.
- The **Listen** directive is being used to listen to multiple ports and IPs on a system which cannot support it.

error—An error has occurred, usually having to do with a client request.

- Invalid configuration parameters were detected.
- Client attempted to authenticate as an unknown user, did not provide the correct password, or was not a member of the correct group.
- A malformed HTTP message or URL was sent.
- The server was unable to find the appropriate authentication files to perform authentication of the client.

warn—An unexpected event occurred.

- The Apache PID file was still present when the server is started; this indicates a bad shutdown last time.
- Apache failed in its initial attempts to kill a child process gracefully.

notice—A normal action of some significance has occurred.

- The Apache server is starting, restarting, or stopping as requested.
- A child was caught exiting due to a nonstandard signal.

info—Records' events of a purely informative nature.

- Records the password (Email address) of an Anonymous Login.
- The build time and date of a server is recorded as the server is started.
- A client connection timed out.

debug—Records' events indicating simple actions on the server. (This level is primarily of use to people writing their own Apache modules.)

Error log entries include the time, level of the event, and a brief description of the problem. It is possible to send Apache events to the system to be recorded using the `syslog` mechanism; this is not recommended because it intermingles Apache events with those of several other services.

3.4.3.2 Configuring Custom Log Formats

The Apache server *mod_log_config* module gives the ability to define custom formats for log information. Events can be set to record a wide range of information by using special %-codes within the format string.

- `b` - *bytes sent:* This code returns the number of bytes of content sent by the server in response to a client request.
- `f` - *filename:* This code returns the path and filename of the resource requested by the client according to the servers operating system.
- `{name}e` - *environment variable:* This code returns the settings of the specified Apache internal environment variable (not to be confused with environment variables from the operating system). This is useful for debugging new Apache modules.
- `h` - *resolved hostname:* This code returns the hostname of the client as resolved according to the procedure specified using the **HostNameLookups** directive. (See Section *****.) If **HostNameLookups** is set to ***off*** (its default value), then this code simply returns the IP address of the client. If **HostNameLookups** is set to ***on*** or ***double,*** then the server will determine the Internet name of the client machine, using simple or double reverse DNS resolution respectively, and return that value.
- `a` - *remote IP address:* This code returns the unresolved IP address of the client machine.
- `{name}i` - *HTTP field in client request:* This code returns the contents of the HTTP field in the client request message whose name matches the name in the braces preceding the `i`. The comparison between the name specifier and the field name is not case sensitive.
- `l` - *remote logname:* If the **IdentityCheck** directive is set to ***on***, the server will send a message to the `identd` server on the client machine. If the `ident` server is running and responds, it will return a string with the username associated with the client request which will then be returned by this code. If **IdentityCheck** is on, but for some reason there is no `identd` response from the client, this code will return `unknown`. (See Section ***** regarding **IdentityCheck**.)
- `{name}n` - *Apache note:* This code returns the value of the named "note" within the Apache server. Generally, the average administrator will not need to consider notes unless specifically instructed.
- `{name}o` - *HTTP field in server reply:* This code returns the contents of the HTTP field in the server reply message whose name matches the name in the braces preceding the `i`. The comparison between the name specifier and the field name is not case sensitive.
- `p` - *incoming port:* This returns value of the port to which the Apache server is listening as defined by the **Port** directive.
- `P` - *child process ID:* This code returns the process ID of the child process which serviced the request. (See Section 3.3.4.1 regarding child processes.)

- `r` - *first line of client request:* This code returns the first line of the HTTP request from the client. This line contains the HTTP command, the URL to be processed, and the HTTP version number which the client is using.
- `s` - *HTTP status code of server reply:* This item returns the HTTP status code of the server response to the given request.
- `t` - *time of request:* This code returns the time the request was made. The time is given in Common Log Format, being `[`*`day/month/year`*`:`*`hour`*`:`*`minute`*`:`*`second offset`*`]`. Alternatively, a time format may be placed between braces, just before the t using the same syntax as that given for the strftime(3) function.
- `T` - *time to process request:* This code returns the number of seconds (rounded) the server took to process the given client request.
- `u` - *remote user:* This code records the username which the client sent to the server. This value is suspect if the returned status code is 401 (Unauthorized).
- `U` - *requested URL:* This code returns the URL requested by the client.
- `v` - *servername:* This is the Internet name of the server processing the given request according to the **ServerName** directive setting.

A format can be defined using the **LogFormat** directive. When the format is used in the creation of a log file the format string is simply a text string which will be copied, character for character, to the log file every time a client makes a request. The exceptions are text that begins with a percent (%) character. The Apache server will attempt to convert these to %-codes and will return an error if it cannot. As such, the percent (%) character cannot be used in the string except as a percent code. The entire format string should be enclosed in double quotation marks ("). If the administrator wishes the log file output to contain double quotations marks, they should be preceded by a backslash (\). (I.e., "text \"quoted text\" text") The backslash will not appear in the output in this case. (I.e., the above string would read: text “quoted text” text.) The %-codes will be replaced with the appropriate information from the server when written to the log file.

If the format string contains a %-code for a piece of information which is not available or cannot be looked up or is just generally unavailable due to an error, the %-code will be replaced with a dash (-) in the log file.

3.4.3.3 The *mod_log_agent*, *mod_log_referer*, and *mod_usertrack* modules

These three modules will be described in this section; they are very small, simple to implement, and provide little security benefit over the *mod_log_config* directives. None of these modules are part of the default Apache build; to be used they must be enabled in the `configure` command.

The *mod_log_agent* has one directive: **AgentLog**. This directive specifies the path and name of a log file. Every time a client requests a resource, the Apache server checks the "User-agent" field of the client request. If it exists, the contents are written to the specified agent log file. No other log information is written. The "User-agent" contains information concerning the requesting client, browser type version, etc. This information is not guaranteed accurate. There is a difference between the use of this directive and a custom log with the format string "`%{User-agent}i`." When the **AgentLog** directive is used; nothing is written in the log if the client request does not contain a "User-agent" field. When the custom log is used, it will write a dash (-) indicating that the information was unavailable. This module and directive is not recommended. They provide little security benefit wise because of the lack of contextual information.

The *mod_log_referer* has two directives: **RefererLog** and **RefererIgnore**. The **RefererLog** directives specifies the path and name of a referer log file. When a resource is requested, the Apache server checks the "Referer" field of the client request. If it exists, the Apache server consults the **RefererIgnore** directive which contains a space separated list of strings. If any of the strings appear in the contents of the "Referer" field, the value is ignored. Otherwise the server will record the contents of the "Referer" field followed by requested resource, in the specified log file. The "Referer" field contains the complete URL of the referer (http://servername/path/resource.html). The difference between using the **RefererLog** directive and a custom log with the format "`%{Referer}i -> %U`" is that the **RefererLog** will write no entry if the "Referer" field does not exist (the client typed the URL in directly instead of linking from somewhere) while a custom log would record a dash (-) followed by the URL being requested. Custom logs do not give the ability to ignore events. These directives could be to learn what external sites had linked to the web page. This could be done by defining a referer log and then specifying the local server name in the **RefererIgnore** directive.

The *mod_usertrack* module contains two directives: **CookieTracking** and **CookieExpires**. When **CookieTracking** is given, the parameter ***on*** it sends an identity cookie in the response to every client request which did not include one. If the client is so configured, it will return this identity cookie with each subsequent request to this server. The cookie can be extracted and used to log the click-paths of individual clients using the "`%{cookie}n`" %-code. The **CookieExpires** directive specifies how long a cookie remains in effect. This can either be given in seconds, or in a string specifying years/months/days/etc. If the **CookieExpires** directive is excluded, the cookie will last until the client browser restarts. Cookies should be checked for expiration date because they only use 2-digit years. Most browsers recognize lower numbers as being after the year 2000, but it is inadvisable to make a cookie last very long.

The above directives may appear only once in any servers configuration.

3.4.3.4 Defining and Implementing Custom Log Files

The *mod_log_config* module contains three directives that can be combined to create custom audit logs. The directives are **CustomLog**, **LogFormat**, and **TransferLog**. The module also includes the **CookieLog** directive, but this directive is deprecated and should not be used. There are several ways this can be done using these directives to create audit files: **LogFormat** can describe a format and **TransferLog** can create the file, **CustomLog** can define the format and create the file, and **LogFormat** can define a nickname and **CustomLog** can create the file. The latter method is used in the default Apache configuration files and, although any method could be used in subsequent invocations, this paper will continue using it.

The *mod_log_config* directives may each be used any number of times to define any number of log file formats and to create any number of audit files.

3.4.3.5 Securing Log Files

All log files created by the Apache server are owned by the root user and are only writable by the root user and readable by all users. The default Apache installation places log files in a directory all users can enter and browse; but only the root identity can add files to this directory. For the most part, this is considered a secure setup; it prevents anyone but root from changing or replacing log files without first defeating the security of the Linux operating system. Many administrators find that letting users read the log files is more access than they can allow. Log files can sometimes contain sensitive information including, server vulnerability alerts, the true paths of resources, or simply information about server usage. For these reasons, many administrators choose to prevent read access to all but themselves.

This can be accomplished in several ways. The easiest is to modify the security of the directory into which the logs are written, making the directory owned by the server administrator. The administrator has full access to the directory while all other users are given no access. Since events are logged under the root user identity, the Apache server will always be able to record events no matter what the security settings are on the directory.

3.4.4 Configuration Information

The two log files defined in the default Apache configuration should be sufficient for most security purposes. Additional log files are necessary if there is additional information of particular interest. This example configuration will define one such scenario.

In the case described below there is a requirement for a log file that records data formats that the client will accept and the language that text should be in. These pieces of information are presented in the “Accept” and “Accept-language” fields of the client request respectively. To provide contextual information, the log file will also record the time of the request, the address of the requester, and the status code of the server reply. This demonstrates the use of the *mod_log_config* directives.

The *mod_log_config* module is enabled as part of the default Apache build.

```
LogFormat "(%t: %h :%s) Format: %{Accept}i; Language:
%{Accept-language}i" accepted
```

This directive defines the format and gives it the nickname “accepted.” (The events produced by this format will be discussed below.)

```
CustomLog "/usr/local/apache/var/log/accepts" accepted
```

This implements a new log file named “accepts” in the “/usr/local/apache/var/log” directory. This audit file uses the “accepted” format. From this point on, whenever a client request is serviced, an event message with the format defined by the “accepted” format will be written to the “accepts” log file.

A brief excerpt from the produced log file appears below:

```
([16/Aug/1999:08:58:07 -0400]: 10.0.1.4 : 200) Format: image/gif, image/x-xbitmap,
image/jpeg, image/pjpeg image/png; Language: en
([16/Aug/1999:08:58:16 -0400]: 10.0.1.2 : 200) Format: image/gif, image/x-xbitmap,
image/jpeg, image/pjpeg, image/png, */*; Language: en
([16/Aug/1999:08:58:17 -0400]: 10.0.1.2 : 200) Format: image/gif, image/x-xbitmap,
image/jpeg, image/pjpeg image/png; Language: en
([16/Aug/1999:08:58:40 -0400]: 10.0.1.5 : 200) Format: image/gif, image/x-xbitmap,
image/jpeg, image/pjpeg, image/png, */*; Language: en
```

With the exception of the %-codes, all text appears in the log file exactly as it does in the format string.

3.4.5 Synopsis and Recommendations

The audit files created in the default Apache configuration file should generally be sufficient for most security needs. Extra log files only need to be implemented when a site requires some additional piece of information not provided by the default logs.

It is important that the administrator monitor the log files regularly in order to detect possible attempts to violate the security of the system. Automated tools exists which can detect suspicious patterns in log files and the reader is advised to look into these. The error log is particularly useful if the administrator notices that the server is not acting as expected. Often bad configurations or altered resources will cause a descriptive event to be written to the error log that can then assist the administrator in fixing the problem.

Log files, if left alone, can grow to an unwieldy size. For this reason, administrators will often empty or replace log files so that they never grow beyond a certain size. Administrators must be aware that the Apache server keeps an internal counter as to where the end of the log files are and if a log file is altered, this value becomes invalid. The result is that no further events can be written to the file. For this reason, whenever a log file is altered, the Apache server must be restarted so it can acquire the new endpoint of the log file.

3.4.6 Additional Topics

This section describes features of the Apache auditing mechanism that were not covered in the above description. Generally, these topics were skipped because they were deemed unnecessary in the implementation of a basic secure web server. They have been included here to inform the reader that the additional functionality and options are in fact available should the reader wish to pursue them.

- *Auditing Processes*—It is possible, instead of sending audit events to a file, to send them to a process. This process can parse the event and then perform some action based on its content. For example, if a particular user authenticates, the administrator can be sent an email alerting them.
- *Multiple Events Being Logged to One File*—It is possible for more than one custom log to send different events to the same file. In this case, the events become interleaved in the file. Generally, doing this adds nothing to the usability of the audit file.
- *Logging to the `syslog` Mechanism*—Many services on Linux send their log events to the `syslog` mechanism so they will all be collocated. The Apache server can also pipe the events which would normally be sent to the error log to the `syslog` mechanism to be placed with the events from the other services. Administrators may find this more convenient. However, most beginning administrators are advised to keep the Apache logs separate to enhance readability.

3.5 Availability

Availability provides the mechanism to ensure that system data is available to users when required. Apache provides availability within the *core* module by allowing the administrator to set configuration variables to maximize performance and availability.

3.5.1 Modules

The directives that impact availability of the web server are part of the *core* Apache module; therefore, there are no additional modules required. These directives are:

- **KeepAlive—**Controls the persistence of TCP connections. A persistent TCP connection will allow more than one request per connection. This is a boolean value, on or off.
- **KeepAliveTimeout—**Controls the number of seconds a connection will remain open waiting for the next request. Must be used in conjunction with **KeepAlive**.
- **ListenBacklog—**The maximum size for the queue of pending TCP connections. An entry for **ListenBacklog** is not present in any of the three configuration files.
- **MaxClients—**Controls the upper limit on the numbers of simultaneous requests supported. Requests over this limit will be queued to the **ListenBacklog**.
- **MaxKeepAliveRequests—**Controls the maximum number of cumulative requests per connection. Must be used in conjunction with **KeepAlive**.
- **MaxRequestsPerChild—**Controls the maximum number of cumulative requests that a given child process will handle. When the maximum is reached, the process will die. Helps to reduce the occurrence of accidental memory leakage. Cannot be used on Win32 platforms.
- **MaxSpareServers—**Controls the desired maximum number of idle child processes. As demand on the web server increases child process may be created according to the next directive, **MinSpareServers**. If demand decreases and this maximum number of idle child processes is exceeded, the parent web server process will kill excess processes until the number is decreased to the maximum.
- **MinSpareServers—**Controls the desired minimum number of idle child processes. As demand on the web server increases, and the number of idle child processes falls below this minimum, the parent process will create new child processes at the rate of one per second until the minimum is reached.
- **RLimitCPU—**Controls the maximum number of seconds of CPU processor time that can be allocated to a process. Expressed in terms of seconds per process.
- **RLimitMEM—**Controls the maximum memory allocation per process. Expressed in terms of bytes per process.
- **RLimitNPROC—**Controls the number of processes per user. Impacts CGI processes when these processes are run under the web servers Userid. Can cause a "cannot fork" error message when the web server process reaches the **RlimitNPROC** value.
- **StartServers—**Controls the number of child processes created at startup of the web server.

- **ThreadsPerChild—**Applies to the Win32 environment only. Controls the number of simultaneous threads each child process can support. Equates to the **StartServers** directive on and Linux platforms.

3.5.2 Default Configuration

The default settings for most of the directives impacting availability are found in the httpd.conf file. Other directives have default values that are set in the source code. These directives usually do not have an entry in the default configuration files. However, the configuration files can be modified to specify a new value for these directives. **ListenBacklog**, **RLimitCPU**, **RLimitMEM**, and **RLimitNPROC** are examples of directives of this type. The default values for these directives are given in Table 3.

Table 3. Default Availability Directive Values

Directive	Default Value
ListenBacklog	511
RLimitCPU	Unset, use operating system default
RLimitMEM	Same as **RLimitCPU**
RLimitNPROC	Same as **RLimitCPU**

The default settings, in the httpd.conf file, for selected directives are given in Table 4. The units of measure, in parentheses, are given for reference only.

Table 4. Default Availability Directive Values

Directive	Default Value
KeepAlive	on
KeepAliveTimeout	15 (seconds)
MaxClients	150 (clients)
MaxKeepAliveRequests	100 (requests)
MaxRequestsPerChild	30 (requests)
MaxSpareServers	10 (servers)
MinSpareServers	5 (servers)
StartServers	5 (servers)

3.5.3 Configuration Information

These directives are used to tune the performance of the web server. Performance tuning is highly dependent on the unique operating environment of the web server. It is inappropriate to make specific recommendations concerning the values for individual availability directives. Some general guidance is appropriate.

The values for **StartServers** and **MaxClients** are probably the most appropriate directives to tune. By monitoring the process status on the web server during key periods, it should be easy to determine an appropriate value for both directives.

The value for **MinSpareServers** should be kept relatively low, as close to the default as possible. It should not be greater than **StartServers** or it will immediately supercede the **StartServer** value. An excess number of idle spare servers will require additional system resources. Also, the gap between **MinSpareServers** and **MaxSpareServers** should be small for the same reason. It is not prudent to use the same value for **MinSpareServers** and **MaxSpareServers**. The resource requirements to start a new child process are higher than the same requirements to maintain an existing process. The reasonable gap between these two directives takes advantage of existing processes without burdening the system with resources dedicated to idle processes. The **MaxRequestsPerChild** directive is probably best left at the default setting unless memory leakage is impacting web server performance.

3.5.4 Synopsis and Recommendations

The **ListenBacklog** directive normally does not need to be tuned. However, it can be increased as a temporary defensive measure against a TCP SYN flooding attack. Increasing this value will increase the maximum size of the queue and make the web server more resilient to a denial of service condition.

If Apache Web Server is deployed on a or Linux platform, use **MaxRequestsperChild** to control memory leakage by have the child process terminate after a reasonable number of requests.

3.6 Integrity, Confidentiality, and Nonrepudiation

The three services (integrity, confidentiality, and nonrepudiation) are listed here because they are commonly listed among the seven standard security services. (The other services being authentication, access control, auditing, and availability, all described earlier.) These three services are listed under one section because none of them are actually implemented by the standard release of the Apache Web Server.

Integrity is the procedure by which a recipient of a message can gain confidence that the message has not been changed in transit. This is usually done through the use of a cryptographic checksum or other similar procedure. Apache contains a **ContentDigest** directive in its *core* module designed to create an MD5 hash of the page so that the recipient may check its integrity. However, the lack of any cryptographic mechanisms means that any attacker capable of modifying a page's content in transit will also be able to create a new hash thus preventing the recipient from detecting the modification. As such, it cannot be claimed that Apache implements integrity. The fact that the Apache server does not implement this service means that a savvy attacker might be able to replace messages traveling to or from the server with messages of their own design. This is not necessarily easy to do, but the lack of the integrity check makes it possible. This service is not commonly implemented on web servers.

Confidentiality is the process by which messages to or from the server are encrypted in order to prevent an eavesdropper from learning the contents of the transaction. This service requires the use of modern cryptographic protocols, usually SSL. Unfortunately, the U.S. government has strict laws concerning the export of products that use cryptography. As the makers of the Apache Group wished that their product be freely distributed throughout the world, this meant that the server could not include the ability to encrypt data. The result is that any information sent from the server to the client and vice versa can be read by anyone capable of watching the wire.

There is a related product named Apache-SSL which does provide the confidentiality service. It is legal to use within the United States providing the user has purchased a license for RSA encryption. With some exceptions, it is legal throughout the rest of the world without restriction. It can be downloaded from sites (none of which are located in the United States) that are listed on the Apache-SSL homepage: http://www.apache-ssl.org.

Nonrepudiation is the procedure by which one member of a transaction can prove that the other transaction member performed a given action. It is virtually never implemented in web servers and would provide no security advantage in most settings.

3.7 Common Gateway Interface (CGI) and Scripting

Scripting for the Web falls into two categories. These are client-side scripting and server-side scripting. Server-side scripting is the focus of this section. The Common Gateway Interface is one mechanism to pass server-side scripting tasks between the web server and scripting engines. Since client-side scripting does not happen at the server, it does not have a security impact on the server. The remainder of this section will focus on server-side scripting and specific issues with the Apache web server

3.7.1 Modules

The Apache Web Server Version 1.3.3 contains two modules that relate to the CGI. The first module, *mod_cgi*, provide the basic functionality for the web server to process files, pass these files to scripting engines, and return the output to the browser. *Mod_cgi* is automatically compiled into the default Apache Web Serve build. The second module, *mod_env*, is used to make environment variables available to CGI scripts and Server Side include commands. *Mod_env* is not complied into the default Apache Web Server build.

3.7.2 Default Configuration

The default access.conf file defines a **<Directory>** container directive for CGI scripts. This default path to this directory is /home/httpd/cgi-bin. The actual path is entirely dependent on the individual configuration for a specific web server. Within this <Directory> container directive the **Options** directive is set to None.

The default srm.conf file defines an **AddHandler** and **ScriptAlias** directive for CGI scripting. Both of these directives are commented out.

These conditions mean that the structure is present to support CGI scripting, but it is not operational.

3.7.3 Background Information

CGI has been the mechanism used, for a number of years, to pass server-side programming tasks between the web server and scripting engines.

3.7.3.1 Markup and Scripting Languages

There are two basic categories of languages that must be considered with regards to a web server. The first category is actually the markup languages that define the content and form of information provided to the web browser. The second category of languages is the scripting languages.

3.7.3.1.1 Markup Languages

Markup languages are derived from a parent language named SGML. The most familiar of these languages is HTML.

As with all markup languages, HTML is transmitted to the web browser as ASCII text; essentially, the HTML commands are human readable.

3.7.3.1.2 Scripting Languages

Scripting languages are interpreted languages. As such, the instructions are normally transmitted in one form (in this case ASCII text) and then parsed or read by a process that interprets these instructions. The parsing means that the process carries out a specific set of internal instructions to produce the results sought by the scripting language commands. Interpreted languages require the presence of this computer process, normally called the interpreter, to accomplish the work.

As with markup languages, the instructions created in these languages are human readable. These instructions are organized and stored as files and are referred to as source code. The source code can be transmitted from one computer to another; however, to carry out the commands prescribed in the source code, the interpreter must be present on the destination machine. The interpreter can also be referred to as the scripting engine.

Scripting languages are used both on the web browser and the web server to provide functionality to a web site that is beyond the capabilities of HTML. When used on the web browser, it is referred to as client-side scripting. When used on the server, it is referred to as server-side scripting.

3.7.3.2 Client-Side Scripting

In client-side scripting, the programming instructions are passed from the server to the client unexecuted. These instructions are commonly referred to as source code. When the source code reaches the client, it is acted upon in some manner.

JavaScript, developed by Netscape, is the most popular client-side scripting language. Other languages are JScript from Microsoft and PerlScript. Although not an interpreted language, Java applets can be used in client-side programming. With each of the scripting languages and Java, the client computer must have the scripting engine or a Java Virtual Machine (JVM). Most modern browsers include a JVM.

Client-side scripting adds functionality to the web browser. These functions include local processing like an order calculator in a shopping cart application, client-side form verification, visual effects like mouseover effects, and additional processing of data.

3.7.3.3 Server-Side Scripting

Server-side scripting is processing that occurs on the server. For network efficiency, this processing should be actions that cannot be performed on the client. Queries to a backend database and formatting of that data in HTML is an example of the type of processing that normally is done from the server.

3.7.4 Configuration Information

CGI scripting can be enabled or supported in any number of directories. If CGI script is explicitly enable for a parent directory, any child directories will inherit this condition. CGI scripting can be explicitly enabled or disabled on a per directory basis

3.7.4.1 Enabling CGI Scripting

Enabling CGI scripting requires mapping a CGI file extension and identifying the directories where CGI scripts are enabled. This mapping ensures that the server processes these files as CGI scripts. There are two directives needed to complete this step. The first is the **AddHandler** directive. Adding an **AddHandler** directive to a **<Directory>** container directive identifies to Apache Web Server that the server must process files with the mapped extension. The second directive is the **Options** directive. The **Options** directive is the directive that actual enables CGI scripting. Where the **Options** directive is placed is critical. If the directive is placed in the **<Directory>** container directive for the root directory, CGI scripting is enabled globally. If the intent is to enable CGI scripting on a directory by directory basis, the directive may be placed in the appropriate **<Directory>** container directives or in a directory .htaccess file. CGI scripting may also be control through virtual hosts. In this case, the directive would be placed in the <**VirtualHost>** container directive. An example using the two directives to allow CGI scripting in all directories follows. **<Directory>** container directives are typical found in the access.conf configuration file.

```
<Directory />
        AddHandler cgi-script .cgi
        Options ExecCGI
  </Directory>
```

In the example above, the **AddHandler** directive, instructs the web server to process all files with the .cgi extension as CGI scripts. The **Options** directive specifies ExecCGI as the option currently supported for this directory. Since the ExecCGI parameter is not preceded by a “+” symbol, the options are not cumulative; ExecCGI is the only option supported.

3.7.4.2 ScriptAlias

ScriptAlias is another directive which influences the processing of CGI scripts. This directive is used to specify a logical alias for a directory that supports CGI scripting. The directory will likely have a different physical path then the alias provided in the directive. In effect, the **ScriptAlias** directive provides a shortcut to the directory. **ScriptAlias** is normally found in the srm.conf configuration file.

3.7.5 Synopsis and Recommendations

3.7.5.1 Scripting Languages and Security

Scripting languages are more significant to security on both the server and client than markup languages. The reason is that scripting languages are interpreted or executed in some forms where markup languages are not. Generally, the concern is that the parent process executing the script may have wide-ranging permissions. If the script performs some inappropriate action, the security of the system may be compromised.

Execution of server-side source code can have a significant security impact. A simple example of this can be shown in a weak implementation of a **finger** service. The **finger** command is a simple UNIX utility to provide information about users on local and remote hosts. The format of a **finger** command run at the command line is:

```
finger [options] <username>
```

When this command is executed, it is done in the context of the current user. This means that the permissions and authority of the current user is used to determine if the command can be executed and the information requested is accessible to the current user. In most cases, the finger command and the data requested are accessible to everyone.

When a **finger** service is implemented through a web server, the command is executed in the context of the user that started the web service or a special user account created specifically for the web service. This user context is normally privileged. This is the first part of the problem.

The next part requires an explanation of how the finger service is implemented. Through the web, **finger** works by accepting text input from the user. Normally, this is a text input field in an HTML form. In this text field, the user would input the username, in the form of *johndoe* or *johndoe@acme.com*. The web server must execute the **finger** command using the inputted information as the username.

The syntax rules of commands allow multiple commands, delimited by a semicolon, to be submitted on one line. By appending additional commands to the end of the username string, a user can have these commands executed in the user context of the web server. If the web server has a privileged user context, these commands can undermine the security of the system.

To summarize, the risk of CGI scripting is in the content of the actual script. Web server administrators should take two precautions when dealing with CGI scripting:

- Inspect the code of scripts before allowing them on the system.
- Limit the permissions of the user ID that the scripts execute under.

3.8 Server Side Includes

The SSI file is an HTML or text file that contains additional instructions that are processed by the server before the file is transmitted to the client.

3.8.1 Modules

The Apache Web Server Version 1.3.3 contains two modules that relate to Server Side Include files. The first module, *mod_include*, provides the basic functionality for the web server to parse files and execute instructions contained within those files. *Mod_include* is automatically compiled into the default Apache Web Serve build. The second module, *mod_env*, is used to make environment variables available to CGI scripts and Server Side Include commands. *Mod_env* is not complied into the default Apache Web Server build.

3.8.2 Default Configuration

Server Side Includes requires three directives to activate which will be described in detail in the next section. There are no active **AddType** or **AddHandler** directives in the three configuration files for Apache. These directives are most likely found in the srm.conf file. In the three **<Directory>** container directives in access.conf, the **Options** directive does not allow included files. For the cgi-bin directory, the **Options** directive is set to None, which explicitly disallows included files.

3.8.3 Background Information

Server Side Includes were devised to give a limited processing ability on the server with having to resort to a CGI script or an API. This limited ability allows two basic tasks—execute, a small set of commands, and manipulate a small set of variables. The following sections describe each of these tasks.

SSI commands are identified in the base document by their unique format. The following is an example of a SSI command within a conventional HTML file. The format is very close to an HTML comment. If SSI parsing is not permitted, any SSI commands will be sent to the client as is and handled as HTML comments by the browser.

```
<HTML>
…
<BODY>
…
<!-#include file="footer.htm" -->
</BODY>
</HTML>
```

The base document is the HTML file or script requested by the client. The server does not parse normal HTML or text files. This is done to increase the response speed of web requests. Supporting SSI requires that the file be identified in some manner so that the server is aware that it must parse the file. The customary means to identify documents that must be parsed is to use a different file extension. The conventional SSL file extension is .shtml.

3.8.3.1 SSI Commands

There is a list of eight SSI commands that can be used in the SSI statement.

- config—Allows the administrator to customize an error message for parsing error and customize the format for displaying file size and date/time values. This command requires one of three argument values dependent on the purpose.
  - errmsg—Used to specify the text of the parsing error message.
  - sizefmt—Used to configure the display of date/time values.
  - timefmt—Used to configure the display of date/time values.
- echo—Prints the contents of an Include or server variable.
- exec—Allows the execution of an external binary file or script. This command requires one of two argument values.
  - cmd—Spawns a shell for programs/commands other than CGI scripts.
  - cgi—Invokes a CGI script.
- fsize—Prints the size of the named file. The format adheres to the sizefmt parameter in the config command. This command requires one of two argument values dependent on whether the path given is physical or logical.

  - file—Used for a path that is specified from the file system. A "file" path is relative. Absolute paths and directory traversing operation (../) may not be used.
  - virtual—Used for a path that is specified as a URL. The path can be absolute and begin with a "/" or relative. Absolute paths originate at the ServerRoot. Relative paths originate from the path of the base document.

- flastmod—Prints the time. The format adheres to the timefmt parameter in the config command. include—specifies the path of the file to be incorporated into the base document. This command requires one of two argument values dependent on whether the path given is physical or logical. Executable content can be part of the included file, if permitted by the parameters to the Options directive.
  - file—Used for a path that is specified from the file system. A "file" path is relative. Absolute paths and directory traversing operation (../) may not be used.
  - virtual—Used for a path that is specified as a URL. The path can be absolute and begin with a "/" or relative. Absolute paths originate at the ServerRoot. Relative paths originate from the path of the base document.
- printenv—Prints a list of all Include and environment variable and their values.
- set—Assigns the value of an Include variable.

3.8.3.2 Include and Server Variables

3.8.4 Configuration Information

3.8.4.1 Enabling SSI on Apache Web Server

There are two steps required to enable SSI on the Apache Web Server. The first is to verify that the module for SSI, mod_include, is compiled into the Apache executable. This module is compiled in the default configuration. The list of compiled modules can be checked with the `httpd –l` command.

The second step is to map an SSI file extension and to identity the directories where SSI is allowed. This mapping ensures that the server parses these files for SSI commands. There are three directives needed to complete this step. The first is the **AddHandler** directive. Adding an **AddHandler** directive to a Directory container identifies to Apache Web Server that the server must parse files with the mapped extension. The second is the **AddType** directive. The **AddType** directive defines the MIME type the web server will use when formatting responses to requests for files with the mapped extension. The third is the Options directive. The Options directive is the directive that actual enables SSI parsing.

Where the Options directive is placed is critical. If the directive is placed in one of the global configuration files, like access.conf, SSI parsing is enabled globally. If the intent is to enable SSI parsing on a directory by directory basis, the directive may be placed in the appropriate Directory containers or in the .htaccess files. SSI parsing may also be control through virtual hosts. In this case, the directive would be placed in the <**VirtualHost>** container directive. An example using the three directives to allow SSI files to be placed in all directories follows:

```
AddHandler server-parsed .shtml
AddType text/html .shtml
Options +Include
```

3.8.4.2 Options Directive Parameters Relevant to SSI

The **Options** directive has a wide range of parameters. Many of these parameters do not have any impact on SSI parsing. The ExecCGI, Include, and IncludeNOEXEC parameters have direct impact on SSI parsing. The ExecCGI parameter allows the execution of CGI scripts. The Include parameter behaves like an on/off switch for SSI parsing. If Include is not part of the parameter list, SSI files will not be parsed and any enclosed SSI commands are treated as HTML comments. The IncludeNOEXEC parameter allows SSI parsing, but if the included file is executable or has executable content, that file or content is not processed.

3.8.5 Synopsis and Recommendations

SSI is a double-edged sword. It is very useful for minimizing the duplication of content that appears across a number of web pages, such as headers, footers, and copyright information. However, the fact that it included files, it can contain executable content make these files a security risk.

It is prudent to use the IncludeNOEXEC parameter on directories where the web administrator does not have direct control of the content. It is very easy for someone to use an SSI command to incorporate executable content that is harmful.

3.9 Redirection and Aliasing

Redirection and aliasing are processes by which the Apache server receives a request and translates it into a reference to a path other than the one that would appear to be the target. For example, if a client requests a resource with the path A, if that path is really an alias or is redirected, the actual resource being requested would have path B, which is not necessarily deducible from A. This section described the modules and directives that allow the Apache server to implement this functionality.

3.9.1 Modules

The Apache Web Server Version 1.3.3 contains three modules that contain functionality that falls under the general category of aliasing and redirection. The modules are:

- *core*—The main Apache module. This contains directives concerning the document root that serves as the base within the file system of all client requests for URLs.
- *mod_alias*—The primary source of directives involving aliasing and redirection.
- *mod_userdir*—A special case of aliasing and redirection; this module implements a user directory mechanism that will dynamically calculate where to look if the Apache server receives a URL containing ~<*username*>.

The *mod_alias* module contains several directives all of which implement aliasing or redirection in some form. The directives differ in how they calculate the URL to be matched and in how the redirection is presented to the client. There are also special directives for when the aliased resource is a CGI executable.

The *mod_userdir* module provides special functionality that allows individual users to have directories that can be easily referenced in URLs. The Apache server checks the URL to see if it contains a tilde (~) followed by a username. The path of the requested resource will by calculated based on the parameter of the **UserDir** directive. Depending on the format of this parameter, this will either result in a simple aliased lookup or an actual redirect. The *mod_userdir* module can also specify that certain users should not have their directories calculated in this way.

3.9.2 Default Configuration

Both *mod_alias* and *mod_userdir* are part of the default Apache build. The *core* module is available in all builds of the Apache server. The default Apache configuration contains at least one directive relating to aliasing or redirection from each of these modules in the srm.conf file. The lines from the configuration file are shown below with additional description:

```
UserDir public_html
```

This directive, from the *mod_userdir* module, defines how the Apache server will calculate the location of a user's publishing directory. In this case, if the Apache server receives a request for a resource in a directory path that contains a tilde (~) followed by a username, the Apache server will look for that resource in the "public_html" folder in that user's home directory. For example, if a client requests "http://*servername*/~schmidtc/projects/index.html" then the Apache server will attempt to return the file "~schmidtc/public_html/projects/index.html" where ~schmidtc is resolved by the Linux operating system to by the home directory of the specified user.

```
DocumentRoot "/usr/local/apache/share/htdocs"
```

This directive, part of the *core* module, is used to specify the base directory in which the Apache server will search for URLs. This directory serves as the root directory in URL specifications. For example, in this case, if a client requests "http://*servername*/directory/file.html," then the Apache server will attempt to return the file "/usr/local/apache/share/htdocs/directory/file.html".

```
Alias /icons/ "/usr/local/apache/share/icons/"
```

This directive establishes an alias. It states that any time a client requests a resource in the directory "icons," the server should look for that resource in the directory "/usr/local/apache/share/icons." For example, if the client requests "http://*servername*/icons/bullets/redbullet.gif," the Apache server will attempt to return the file "/usr/local/apache/share/icons/bullets/redbullet.gif." This directive should not be used if the requested resource may be executable, as it will not execute a target file but will return the executables' contents as if it was a normal file.

```
#ScriptAlias /cgi-bin/ "/usr/local/apache/share/cgi-bin/"
```

This directive establishes an alias to a CGI executable. It states that any time a client requests an executable file in the directory "cgi-bin" the server should look for that executable in the directory "/usr/local/apache/share/cgi-bin." For example, if the client requests "http://*servername*/icons/bullets/script.pl," the Apache server will attempt to return the results from executing "/usr/local/apache/share/icons/bullets/script.pl." This directive should not be used if the requested resource may be a file, as this will result in an error.

3.9.3 Background Information

This section discusses information which is not a straightforward part of the configuration process, but which is necessary for knowledgeable use of the aliasing and redirection mechanisms.

3.9.3.1 General Use

There are two general issues that all users of aliasing, redirection, and **UserDir** directives must be aware of. First, with the exception of the **AliasMatch**, **ScriptAliasMatch**, and **RedirectMatch** directives, the directives will only be triggered if the string immediately following the server name in the URL matches the target string. There cannot be any intermediate characters. For example:

```
Alias /fake_name/ "/usr/local/real_name/"
```

The above directive would be matched on the URL "http://www.server.com/fake_name/…", but not on the URL "http://www.server.com/stuf/fake_name/…". Notice that after the string to be matched, there can be any number of characters extending the path.

The **AliasMatch**, **ScriptAliasMatch**, and **RedirectMatch** directives can match to any portion of the URL. For this reason, they must be used with great care to prevent accidental matching.

The second major issue involving aliasing and redirection directives is the use of trailing slashes (/) in the directives' parameters. These directives behave differently depending on whether or not the parameter to be matched with the URL and/or the parameter representing the true path contains a trailing backslash.

Both the aliasing and redirection directives behave in the same way. If the first parameter does not contain a trailing backslash then it will match both URLs with or without a trailing backslash. If, on the other hand, the first parameter has a trailing backslash, no match will be performed if the requested URL does not have a backslash after the matching string. For example:

```
Alias /fake_name/ "/usr/local/real_name/"
```

The above directive would match requests for "http://www.server.com/fake_name/" and "http://www.server.com/fake_name/directory/file.html", but will not match a request for "http://www.server.com/fake_name".

The second parameter should match the first parameter in that they should both possess or lack a trailing backslash. If this is not observed, the actual file or URL requested will either lack or have an extra backslash when calculated.

There is no difference in the behavior of the **UserDir** directive based on the presence or absence of a trailing slash.

3.9.3.2 Aliasing vs. Redirection

This paper has been referring to aliasing and redirection as if they are two different mechanisms, which is, in fact, the case. The primary difference between the two features is how the process is seen from the perspective of the client. Redirection involves the server sending one of a number of HTTP redirects back to the client. The different types of redirects inform the client of the status of the original document. The four types of redirect messages the Apache server can be configured to send are:

- **Moved Permanently** (HTTP code 301)—Indicates that the resource has been permanently moved to a new location and the client should update its references. The message should include the new path of the resource.
- **Found** (HTTP code 302) —Also known as "Moved Temporarily," this messages indicates that the requested resource is temporarily residing in a different location, but

that it will return to its original path at some time in the future. The message should include the temporary path of the resource.

- **See Other** (HTTP code 303) —Indicates that the resource has been replaced or updated with a different resource at the specified location. The message should include the path of the new resource.
- **Gone** (HTTP code 410) —The requested resource has been removed from the server and no forwarding URL has been provided. This message does not provide a new URL to look up but simply exist to inform the user that the URL was once valid, but is no longer so.

Most modern browsers, upon reception of a redirect message, will automatically create a request to the provided URL making the entire process transparent to the user. (The obvious exception to this being the "Gone" message that does not contain a URL to request.) The *mod_alias* module contains directives that can respond to client requests with any HTTP message, the aforementioned four messages being the most likely in the case of redirection. Although it is most common to redirect to another HTTP server, if the parameter specifies a different protocol that the browser is capable of requesting, this protocol is used. For example, **Redirect** ***"ftp://ftp.domain.com/folder"*** would be parsed by most modern browsers and result in a request to the given server using the ftp protocol.

Aliasing, unlike redirection, takes place entirely on the server. If the Apache server receives a request for a URL that is aliased it internally recalculates the path of the requested resource based on the aliasing directive used and returns the resource with the new path. No messages are sent to the client during this process. As such, aliasing is primarily used to re-map the URL file structure in much the same way that a link re-maps the file structure in the operating system. (For more on links, consult the **`man`** pages for the **`ln`** command.) The purpose of such a re-mapping is generally either to shorten what would otherwise be inconveniently long URLs or to hide the file system structure from probing using the HTTP server. The latter is a security advantage in, should an attacker gain access to the file system , it will be structured differently then how it appears to be structured when using the HTTP server.

If the same URL is the subject of both an alisaing and a redirection directive, the redirection directive will always take precedence. As a result, the client with receive a HTTP redirect message and the aliasing directive will not be considered.

3.9.3.3 Aliasing, Redirection, and Container Directives

Because they affect the path the Apache server will use to look up a requested resource, the interaction of aliasing and redirection directives with container directives, which are linked to the path and name of a resource, may seem to be somewhat confusing. The rule to remember is that the **<Location>/<LocationMatch>** directives are matched against the URL while the **<Directory>**/**<DirectoryMatch>** and **<Files>**/**<FilesMatch>** directives are matched against the file system . The result is that the former directives are matched against the alias name that the client requests while the latter directives are matched against the actual path and file on the file system . For example, consider:

```
    Alias /aliaspath "/usr/local/apache/share/another_path"
```

If a user requests “http://*servername*/aliaspath,” container directives will have the following behaviors:

- **<Location *"aliaspath"*>** will apply all contained directives to the request
- **<Location *"/usr/local/apache/share/another_path"*>** will not be considered in this request
- **<Directory *"aliaspath"*>** will not be considered in this request
- **<Directory *"/usr/local/apache/share/another_path"*>** will apply all contained directives to the request
- **<Files *"file.html"*>** will apply all contained directives to the request only if this is the name of the file requested. (In the above example, only the directory name is aliased.)

Following up on the last bullet, the **Alias** directive may be used to alias individual files. For example, consider:

```
    Alias /aliaspath2/file.html
"/usr/local/apache/share/another_path2/real_file.html"
```

In this case, requests for “http://*servername*/aliaspath2/file.html” would cause the server to return the file “/usr/local/apache/share/another_path2/real_file.html”. (In fact, an alias with a filename can refer to a directory and vice versa. Aliasing can simply be viewed as string replacement when the server looks up the resource.) If the recent **Alias** directive were used, **<Files>** container directives would behave as follows:

- **<Files *"file.html"*>** would not be considered in this request
- **<Files *"real_file.html"*>** will apply all contained directives to the request

As can be seen in the previous two examples, the **<Location>**/**<LocationMatch>** directives are matched against the alias while both the **<Directory>**/**<DirectoryMatch>** and **<Files>**/**<FilesMatch>** directives are matched against the actual path and name of the requested resource.

Redirection does not consider any container directives no matter the type. When a client requests a target that is the subject of a redirection, the Apache server will immediately send the redirect message without any further consideration.

3.9.3.4 The *mod_userdir* Module

The **UserDir** directive in *mod_userdir* can behave either as an aliasing or redirection call depending on how it is configured. The calculations by which a username in a URL is translated is somewhat complicated and allows for virtually any configuration of user publish directories providing that the publishing directories of all users have the same structure.

If the parameter of the **UserDir** directive does not contain a colon (which would indicate a protocol specification), it behaves as an aliasing directive. Specifically, it behaves like the **Alias** directive in that requests for a cgi will return the text of the executable rather than execute the file and return its result. The directory in which the Apache server looks for the requested resource depends on the format of the directive's parameter:

- **UserDir *"directory"***—If the parameter is not led with a backslash (/) then it represents the path under the users home directory. The given path may be of any depth. In the above example, Apache would look in the "~*username*/directory/" directory. Note that, for the purposes of **<Directory>**/**<DirectoryMatch>** matching, the actual path of the users home directory is used rather than the tilde (~) representation.
- **UserDir *"/directory"***—If the parameter is led with a backslash (/) then the given path, followed by a directory named after the given username will be checked. Again, the given path may have any depth. In the above example, Apache would look in the "/directory/*username*/" directory.
- **UserDir *"/directory1/*/directory2"***—If the parameter is led with a backslash (/) and contains an asterisk (*) then the given path will be checked for the requested resource where the asterisk is replaced with the requested username. In the above example, Apache would look in the "/directory1/*username*/directory2/" directory.

If the parameter string of **UserDir** contains a colon, it is assumed that a protocol is being specified and that the directive is specifying that a redirection be performed. Normally, such strings would begin "http://," but any protocol could be used, just as with normal redirection. Upon reception of a request for the given users directory, an HTTP 302 (Found) message is returned to the client with the new URL. The construction of the URL depends on the parameter of **UserDir**:

- **UserDir *"http://www.server.com/directory"***—If the parameter does not contain an asterisk (*) then the users name is appended onto the end of the URL (with a backslash inserted just before it). The URL given in the parameter may have any depth. In the above example, the client would be redirected to "http://www.server.com/directory/*username*/".
- **UserDir *"http://www.server.com/*/directory"***—If the parameter contains an asterisk (*) then the client is redirected to the URL given in the parameter with the asterisk

replaced with the given username. In the above example, the client would be redirected to "http://www.server.com/*username*/directory/". Notice that the username in the redirected URL is not led with a tilde (~). See the next bullet on how to do that.

- **UserDir** ***"http://www.server.com/~*"***—To redirect the client such that the requested directory will be ~*username*, simply place a tilde (~) in front of the asterisk (*) in the parameter. In the above example, the client would be redirected to "http://www.server.com/~*username*/".

In all cases, if the user requests a path longer than the username (for example, "~*username*/dir1/dir2/file.html") the path after the username is simply appended to the end of the path being looked up locally or being redirected.

Notice that in all the above examples, no trailing slash (/) is placed in **UserDir**'s parameter. The Apache server automatically adds this to all requests. If the configuration file contains a trailing slash this will result in two trailing slashes which may cause an error.

When used in conjunction with container directives, the **UserDir** directive behaves exactly as an aliasing or redirection directive would, depending on how the **UserDir** directive is used. If **UserDir** is used in its aliasing capacity, the **<Location>**/**<LocationMatch>** directives will be compared against the URL (with the ~*username*), while the **<Directory>**/**<DirectoryMatch>** and **<Files>**/**<FilesMatch>** directives will be matched against the actual path of the resource requested. If **UserDir** is used in its redirection capacity, no container directives will be consulted.

If a **UserDir** directive is used along with an aliasing or redirection directive that is set to match on a specific ~*username*, the **UserDir** directive will always take precedence and the aliasing or redirection directive will not be considered. This occurs no matter whether **UserDir** is being used in its aliasing or redirection capacity.

In addition to providing how to calculate the target of a user name lookup, the **UserDir** directive can be used to define which usernames will be converted. This can be done through the inclusion of either ***enabled*** or ***disabled*** as the first parameter of the **UserDir** directive. The use of these parameters will have one of three results:

- **UserDir** ***disabled*** prevents any user directories from being translated with the exception of those specifically allowed using the ***enabled*** parameter.
- **UserDir** ***disabled username username…*** prevents the listed usernames from being translated. The server will attempt to service all other user directory requests.
- **UserDir** ***enabled username username…*** specifically allows the listed usernames to be serviced by the Apache server. When used in conjunction with **UserDir** ***disabled***, the listed usernames will be the only ones serviced.

By default, the Apache server will attempt to translate and service all requests for user directories.

3.9.3.5 Aliasing and Security

Aliasing, and in this instance aliasing refers to the **UserDir** directive when used in its aliasing capacity, can be beneficial in a number of ways. Long paths can be aliased preventing clients from requesting long and unwieldy URLs. Mnemonic names can be assigned to resources that would otherwise require the memorization of a less intuitive path. Additionally, the use of aliasing directives can hide the true structure of the servers' file system from clients. The latter feature can be a great benefit to security.

Many attacks require a foreknowledge of at least some portion of the server's directory tree. Without the use of aliasing directives, the directory tree seen in URLs will be exactly the same as the directory tree of the server itself (minus the path to the server's publish directory). By using aliasing directives the URL directory tree can be broken up and placed throughout the servers' file system , thus foiling mapping attempts.

There is one danger to using aliasing directives in the form of links. A link is a file that connects to a directory at some other location in the file system . Links are used to alter paths on the file system in much the same way that aliasing alters the lookup path in the Apache web server. As far as the user, and Apache, are concerned, the directory the link connects to is simply a sub-directory of the directory that contains the link. This occurs regardless of the target directory's original position in the file system . In effect, a link creates a second path to a directory.

This has two ramifications. First, if a link exists in a directory that is the target of an aliasing directive, the client may be able to follow that link wherever it may lead. For example, if the link "`root`" was linked to the directory "`/`" then the Apache server would attempt to service a request for a resource in the directory "root" by looking in the /, or root directory of the file system . It is unlikely an administrator would wish clients to be granted this ability.

The second ramification of the use of links relates to a link's ability to create alternative paths. The **<Directory>**/**<DirectoryMatch>** directives match against the file system path. However, they are only compared against the path being requested. A link can create a second, different path. As such, a second set of **<Directory>**/**<DirectoryMatch>** directives would need to be added in order to match this new path. Otherwise, the contained directives might be applied to requests using one path, but not to requests using another, resulting in an uneven security policy.

The danger of links can be mitigated in several ways. First, an administrator should always check for links in any directory that can be published. This is the best method for detecting unwanted links. In addition, the **Options** directive can be used to prevent the

Apache server from following links. This can be used in cases wherein the administrator may not be able to control links placed in a given directory, such as when that directory is controlled by an individual user. Finally, the administrator should set up an access control policy such that directories that are not intended for publication can never be accessed. This should be done for all directories regardless of whether they seem reachable through the web server. There is always the possibility that a rogue link may exist connecting a published directory to the other.

3.9.4 Configuration Information

This section presents examples of the use of the various aliasing and redirection directives. Examples include a brief description of the intention of the directive and an explanation of its arguments. Due to the lack of interaction between the various directives, each example is independent. The examples do not reflect any unified attempt to make significant changes to the settings of the server, but were simply contrived to demonstrate the use of their respective directives. Each example also describes directives that are similar to the one demonstrated.

3.9.4.1 Aliasing

This example will show the use of the **Alias** directive. Use of the **ScriptAlias** directive can be easily extrapolated where the target directory contains executables. The displayed directive simply intercepts requests for the "project_1" directory in what appears to be to root directory of the server and links it to the project's directory. The other directives enable clients to access the project directory since, by default, this directory would not allow access.

```
Alias /project_1 "/home/schmidtc/project_1/html"
```

As a result of this directive, whenever the Apache server receives a request for a resource whose URL begins with "/project_1" (after the protocol and server designation) it will look for this resource in the file system directory tree under the "/home/schmidtc/project_1/html" directory.

```
<Directory "/home/schmidtc/project_1/html">
```

Specify the directory being referenced in the alias. By default, no access is allowed to this directory. In order to allow external users to access it, permissions must be enabled.

```
order allow,deny
```

Specify the access control order (see Section *****).

```
allow from all
```

Specify that access is allowed from all hosts (see Section *****).

```
Options None
```

Specify that no special options are permitted. Among other things, this setting prevents the Apache server from following symbolic links while in this directory and prevents the execution of CGI and Server Side Includes (see Section 3.1.4).

```
</Directory>
```

Close off the above container block.

Behavior: Once the above directives are implemented, the server will attempt to service all request for resources within the "project_1" directory in the URL by looking in the "/home/schmidtc/project_1/html" directory. For example, if a user requested "http://www.server.com/project_1/reports/august.html" the server would attempt to find and service the file "/home/schmidtc/project_1/html/reports/august.html". The disabling of symbolic links and executable content using the **Options** directive enhances security.

Note that, while the **order** and **allow** directives allow access to the given file system directory from the Apache server's perspective, it must still be the case that the user "nobody" (or whichever user is specified for the server child processes in the **User** directive) must able to reach the directory within the file system . If the access control of directories in the operating system prevented such access, Apache would be unable to serve the directory.

3.9.4.2 Redirection

Redirection is a relatively simple feature to implement. Apache provides four directives to implement this feature: **Redirect**, **RedirectMatch**, **RedirectTemp**, and **RedirectPermanent**. Of these, **RedirectTemp** is equivalent to **Redirect** with a first argument of ***temp*** while **RedirectPermanent** is equivalent to **Redirect** with a first argument of ***permanent***. This example will demonstrate the use of the **Redirect** directive.

```
Redirect permanent /research "http://research.server.com"
```

When the client sends a request for a resource in the "/research" directory of the URL, the server sends a "Redirect Permanent" message (HTTP code 301) indicating that all requests for this directory should be directed to "http://research.server.com".

```
Redirect seeother /help_files "http://www.server.com/help2"
```

When the client sends a request for a resource in the "/help_files" directory, the server sends a "See Other" message (HTTP code 303) indicating that new versions of the requested files are located in "http://www.server.com/help2".

Behavior: In either of the above cases, client requests are met with redirect errors of the specified type. Most client browsers will parse these errors and transparently request the indicated resource. There are really no other issues with using this directive since it is processed before container directives or aliases are considered.

3.9.4.3 The UserDir Directive

The **UserDir** directive can be useful to site administrators who wish to allow specific users to be able to publish content that they are able to control themselves. The following example describes one possible secure configuration for such a mechanism.

```
UserDir disabled
```

Causes all requests for user directories to be denied by default.

```
UserDir enabled schmidtc rmcquaid kjones
```

Allows the server to look up user directories for the three users listed only. Requests for any other user's user directory will not be serviced.

```
UserDir /home/user_publishing/*/http
```

Specify where the user directories are located. In this case, schmidtc's user directory would be located in "/home/user_publishing/schmidtc/http", rmcquaid's would be in "/home/user_publishing/rmquaid/http" and kjones's would be in "/home/user_publishing/kjones/http".

```
<Directory "/home/user_publishing/*/http">
```

By using the wildcard character in the above specification, this container block applies to the "http" directory within each individual users' publishing directory.

```
order allow,deny
```

Set the order of the access control.

```
allow from all
```

Allow access to these files from all hosts.

```
Options None
```

Specify that no special options are permitted. Among other things, this setting prevents the Apache server from following symbolic links while in this directory and prevents the execution of CGI and Server Side Includes (see Section 3.1.4).

```
AllowOverride AuthConfig Limit
```

Specifies which directives the Apache server will read from .htaccess files, should there be one present in the requested directory. (See Section 3.1.5 regarding .htaccess files.) The parameters allow users to control authentication and access control to their directories through .htaccess files, but will prevent them from changing other properties of their directory (such as giving themselves the ability to follow links).

```
</Directory>
```

Closes off the container block.

Behavior: The above configuration will cause the server to process requests for ~schmidtc, ~rmquaid, and ~kjones. All other requests for user directories will be met with a "Not Found" (HTTP code 404) message from the server. For users that are allowed, Apache will calculate the user's publishing directory and make a request there.

The **<Directory>** container block is used to enable access to the user publishing directories since, but default, access would not be allowed. In addition to granting access to all hosts, the block also contains directives that limit the activities that may be taken within the user directories. Specifically, it prevents Apache from following symbolic links or executing code and limits the directives that will be read from .htaccess files. These directives are particularly important when dealing with user directories since individual users are likely to be able to control the content of these directories without administrative supervision. As such, it is important to configure Apache such that hostile users will not be able to violate security from their user directories. By preventing links from being followed, the administrator is eliminating the possibility that the user might create a link to sensitive information on the server. Preventing the execution of CGI and Server Side Includes prevents users from running executables without supervision. Finally, and most importantly, limiting the directives read from .htaccess files as described allows users to restrict access to their user directories as they wish, but prevents them from granting themselves more capabilities than the administrator has specified in the primary Apache configuration files.

3.9.5 Synopsis and Recommendations

Aliasing and redirection directives can be very useful to the functioning of a web server in general and its security in particular. Specifically, careful use of aliasing directives can hide the internal structure of the file system on the web server from mapping attempts while, at the same time, making it easier to use. The only risk that must be taken into consideration is the danger of links within the file system . However, through careful inspection of published directories, the preventing Apache from following links, and/or denying access to all directories except those the administrator wishes to be publicly viewed.

The **UserDir** directive is a special case of aliasing and redirection and simplifies the task of granting individual users directories from which they can publish content. If the administrator wishes to use this feature, they should follow the example given in Section 3.9.4.3 regarding the secure configuration of the mechanism: allowing user directory lookups only to specific users and restricting the features of those directories. If the administrator does not wish to use this feature, it should be disabled. This can either be done by placing a single **UserDir** ***disabled*** directive in the configuration file, or by disabling the module when building the Apache server by adding the argument "`--disable-module=userdir`". In the latter case, the administrator will need to remove the default **UserDir** ***public_html*** directive from the srm.conf configuration file.

The redirection directives, while not providing much additional security, can be very useful especially in situations where a web site actually consists of several individual web servers. The flexibility of the associated directives allow the administrator complete control of the type of message returned to the client in the event of a redirection.

Directives that allow regular expressions to be used in aliasing and redirection are available through the **AliasMatch**, **ScriptAliasMatch**, and **RedirectMatch** directives. It is very easy to have unintended matches occur when using these. For this reason, it is recommended these directives not be used unless absolutely necessary, and then only with the utmost care to match only the desired URLs.

3.9.6 Additional Topics

There are no other topics relating to aliasing and redirection that have not been addressed in the above section.

3.10 Virtual Hosting

An Apache Web Server can support more than one web site. This is done through virtual web sites or virtual hosts. A virtual web site is a set of content directories that are managed and accessed as if it was a separate, unique web server. Virtual web site is a more descriptive term; however, virtual host is the Apache directive used to implement a virtual web site. This feature was designed to manage the problem commercial Internet service companies face when trying to host multiple web sites.

3.10.1 Modules

The **<VirtualHost>** container directive is part of the core Apache module; therefore, there are no additional modules required to implement virtual hosts.

3.10.2 Default Configuration

There are no active **<VirtualHost>** container directives in the three configuration files for Apache. The httpd.conf does contain an example of the **<VirtualHost>** container directive that is commented out.

3.10.3 Background Information

Before virtual hosts were available, the possible solutions were to have a separate machine for each web site or to have several network interface cards in a single machine. Each of these interface cards would be bound to a different IP address. Virtual web sites can be hosted on the same Apache Web Server by uniquely identifying each site using one of three methods:

- By assigning each site a nonstandard port (e.g., other than port 80) to listen for HTTP traffic
- By assigning each site a unique Internet Protocol (IP) address
- By providing additional information (e.g., Host Header information) in the header portion of each HTTP packet which uniquely identifies the virtual web site to which the packet should be directed

3.10.3.1 Nonstandard Ports

Nonstandard ports are the easiest way to direct HTTP requests to a virtual web site. No modifications to the DNS database are required. The use of a nonstandard port number requires that the port number be explicitly included in the URL (e.g., http://www.mysite.com:8080). This technique is well established and provides the means to host more than one Web site from a single IP address. Requiring a port number in the URL is a drawback for this technique. If the port number is not provided, the web server will assume the default port (port 80) for an HTTP transaction and access the appropriate directory. The request is directed to the wrong Web site. Another use of this technique is to support special services; one example is secure connections through SSL. The server listens for incoming requests on a designated port and then directs the request to the service based on the port information. Some of these port numbers are well-known and others, especially those for proprietary services, may be less well known. By default, secure connections using SSL are made over port 443.

3.10.3.2 Unique IP Addresses

The next technique involves unique IP addresses. In this technique, the DNS name for each virtual web site (including the default web site) is paired to a unique IP address and registered with the Domain Naming System (DNS) server. When an HTTP request for that DNS name is resolved, to the associated IP address, the request will be directed to the correct Web server through the IP address. The server hosting Apache Web Server must be configured to respond to each of the associated IP addresses. More significantly, each virtual web sites must be mapped to a specific IP address.

Virtual web sites using port numbers or unique IP addresses are referred to as IP-based virtual hosts.

3.10.3.3 Host Headers

The final technique is through Host Headers. With Host Headers, a single IP address can be used to support multiple virtual web sites. In the past, the HTTP protocol provided only the IP address of the requested web site and not the DNS name. In the TCP connection supporting an HTTP file request, the DNS name was resolved to an IP address and only the IP address was used for the HTTP file request. Even though multiple DNS names for web sites could be mapped to one IP address on the DNS server, when the HTTP request was sent to that IP address there was insufficient data in the HTTP packet to pass the request to the correct virtual Web site. Version 1.1 of the HTTP protocol added support for host headers.

Operationally, the parent web server process must be able to distinguish packets directed at one virtual host from packets meant for another virtual host. In detail, a Host Header is a reference, in the header data of the HTTP packet, to the DNS name of the Web site. Specifically, the line in the HTTP header will take the form:

HTTP: Host = www.myserver.com

The browser provides the host header information. The server reads this information and directs the request to the correct virtual web site. The following requirements must be met for the host header method to be used:

- The DNS names for these virtual web sites must be registered with the DNS server that supports the organization.
- The web browser must support HTTP 1.1 to pass the DNS name of the virtual Web site in the HTTP header data. New browsers (Internet Explorer 4.0 and Netscape 4.0) support this feature.
- The web server must support this feature; Apache Web Server is compliant.

Virtual web sites using host headers are referred to as named-based virtual hosts.

3.10.3.4 Single versus Multiple Daemons

Each instance of the Apache Web Server is a daemon process. Apache Web Server can support virtual web sites using a single daemon process or multiple daemon processes. With a single daemon process, that process must be configured to listen and accept requests for all of the virtual web sites. With multiple daemon processes, there is more than one daemon process. The maximum would be one daemon process for each virtual web site.

In most situations, a single daemon process is preferred. If the volume of traffic to a specific virtual web site is high, an additional Apache Web Server instance can be started to distribute the load. Resource limitations must be considered when supporting virtual web sites. Each Apache Web Server instance will require a set of resources.

3.10.4 Configuration Information

3.10.4.1 Apache Virtual Host Directive

The **<VirtualHost>** container directive is the mechanism in the Apache Web Server to implement virtual web sites. The general form of the container is:

```
<VirtualHost>
     [Statement Block]
</VirtualHost>
```

Within the statement block, the administrator can use most Apache directives to define the functioning of the virtual web site. Directives within the container apply only to that virtual web site. The following directives may not be used with the <**VirtualHost>** container:

```
BindAddress
Listen
MaxRequestPerChild
MaxSpareServers
MinSpareServers
NamedVirtualHost
PidFile
ServerRoot
ServerType
StartServers
TypesConfig
```

The two important directives are **ServerName** and **DocumentRoot**. These directives are needed for each instance of the web server and each virtual web site. The **ServerName** directive provides the web site's DNS name, for example www.mysite.com. The **DocumentRoot** directive points to the directory that is the root directory of the web site. For example, if the root directory of www.mysite.com is /www/mysite/htdocs, the file index.html is accessed with the URL, http://www.mysite.com/.

The opening **<VirtualHost>** tag contains the IP address or DNS name of the virtual web site. Use of the IP address is recommended. Presently, if a DNS name is used and the DNS lookup fails, the web service will not start. A solution to this problem is anticipated in a future Apache Web Server release.

With Apache Web Server Version 1.3 or later, any virtual host containers for name-based virtual host must be preceded by the **NameVirtualHost** directive. This is a core directive that provides the IP address to resolve any subsequent name-based virtual host containers. DNS names may be used in the **NameVirtualHost** directive, but IP addresses are recommended for the same reason as mentioned earlier. For example:

```
NameVirtualHost 10.0.0.5
     <VirtualHost 10.0.0.5>
          ServerName www.mysite.com
          DocumentRoot /www/mysite/htdocs
     </VirtualHost>
     <VirtualHost 10.0.0.5>
          ServerName www.yoursite.com
          DocumentRoot /www/yoursite/htdocs
     </VirtualHost>
```

If there is more than one IP address associated with the Apache Web Server, each IP address requires a **NameVirtualHost** directive. For example:

```
NameVirtualHost 10.0.0.5
     …
NameVirtualHost 10.0.0.6
     <VirtualHost 10.0.0.6>
          ServerName www.theirsite.com
          DocumentRoot /www/theirsite/htdocs
     </VirtualHost>
```

Port numbers can be specified in the **NameVirtualHost** directive.

3.10.5 Synopsis and Recommendations

While the primary purpose of virtual hosts is to improve the overall utility of the web server platform, they can simplify access to a web resource for the end user. The requirement to recall long and difficult URLs to access a resource is alleviated. From the security perspective, this ease of use helps protect the internal structure of the web server from unauthorized access. This is the same benefit provided by aliasing described in the next section.

3.11 Other Security Issues

The Apache server is a full-featured Web Server in every respect. As such, it contains a great many more services than have been described so far. A complete description of all the features of the Apache Web Server would be a subject of many hundreds of pages. (Trust us on this.) Fortunately, most of the advanced features the Apache server are not necessarily for secure functioning of the server, or even particularly necessary for most general purpose web publishing. The authors of this paper decided the reader would be better served by a concise description of the features of the Apache Web Server which were both in common use and of security relevance. Hence, this document does not describe many features of the Apache server. Readers who wish to become more familiar with the other features of Apache are recommended to the Apache documentation distributed with the server, or to commercial books printed on the subject.

The previous disclaimer notwithstanding, the authors felt that a very brief introduction and description of the major additional features of the Apache server not covered thus far would be helpful. This section is intended to familiarize the reader with these additional features and to indicate what security issues are brought into the picture by there use. This is not a configuration guide to these features; it is merely a survey. If the reader wishes to implement any of them, the details of the implementation and the specific security considerations associated with the implementation are left to the reader.

3.11.1 Proxying

In addition to its normal functionality as a Web server, the Apache server also has the capability of performing the functions of a caching proxy server. A proxy server is a computer located at the intersection between one’s internal intranet and the Internet as a whole. All connections that internal machine (in the intranet) wishes to make with an external machine (in the Internet), must be accomplished by connecting to the proxy server and asking it to make the external connection on behalf of the internal machine. The idea behind this is that, for all intents and purposes, the internal network has only one computer exposed to the Internet, and therefore vulnerable to attack. Moreover, all connection requests from one’s network should appear, from the perspective of someone listening to the Internet, to have come from the proxy server regardless of the original requestor. This possibly prevents hostile outsiders from gaining any information about the configuration of one’s internal domain.

The Apache server is capable of serving as a proxy server when the *mod_proxy* module is compiled in when Apache is built. (The *mod_proxy* module is not part of the default Apache build.) With it, the Apache server is capable of receiving requests from internal hosts and forwarding them on to either internal or external servers as appropriate. The *mod_proxy* module contains a large number of directives that can be used to control specific aspects of how the Apache server handles given requests. For example, Apache can control whether to forward request directly to their target or pass them off to another proxy server, whether certain target sites should be blocked entirely, and which protocols the Apache server will handle.

Beyond the standard ability to proxy internal requests to the outside world, the Apache server can store documents that have been recently requested and serve the local copies the next time a client requests the resource. This saves the proxy server from needing to establish a connection to the target host. This, in turn, reduces load on the Apache server while increasing the speed with which a client's request is serviced.

The implementation of a proxy server is a security feature that should be undertaken very carefully. A well configured proxy server is a great asset to network security. A poorly configured proxy server, at best, provides a false sense of security and may even provide more access to one's internal network than would have otherwise been possible. As such it is important that implementation of the *mod_proxy* module only be done with a complete understanding of its features and with an eye on ones entire security policy.

3.11.2 The *mod_rewrite* module

The *mod_rewrite* module is used to dynamically change (rewrite) URLs' requested by a client causing them to request a different resource. This is done entirely on the server without the knowledge of the client. This functionality is similar to that provided by the aliasing directives, but is much more powerful. The module allows the administrator to define conditions and rules using regular expressions that define how a given URL is to be rewritten. The module will check each URL requested against its list of conditions and applies the associated rules of any conditions that are met.

This functionality gives the administrator a huge degree of control over what a client sees. For example, if a web site services both Intranet and Extranet clients, *mod_rewrite* could be used to examine the environment variable, REMOTE_ADDR for the IP address of the client. With this information, the URL could be re-written to one sub-directory for Intranet requests and a parallel subdirectory for Extranet requests. From the users' perspectives, the URL requested is the same.

The security ramifications of this module are wide ranging. As a word of caution, *mod_rewrite* is a very complex and confusing module. It is best to refer to the Apache documentation for reference and examples of how to employ this module.

3.11.3 Indexing, Server Status, and Other Ways to Remotely Learn About a Server

There are several ways in which the Apache server can end up publishing information about its settings and configuration. This is not always good from a security standpoint. This section covers a few of the more obvious ways in which a client might be able to discover more information about the server than would be strictly necessary for its own legitimate purposes, and what the server can do to prevent this.

This first matter is directory indexing. Directory indexing refers to the dynamically created index pages to be returned to the client, if the client requests a directory and the directory does not contain an index file as defined by the **DirectoryIndex** directive in *mod_dir* (see Section 3.1.3). The formatting of these pages is controlled by directives in the *mod_autoindex* module, which is part of the default Apache build. Some administrators are not comfortable with clients being able to learn the entire contents of a directory. For such circumstances, the administrator can use the **Options** directive from the Apache *core* module. If the directive is used and neither ***All*** nor ***Indexes*** are given as parameters, then requesting a directory which does not contain an index file will not return an index page and instead a "Forbidden" error (403) it will be returned. See Section 3.1.4 for more on the **Options** directive.

The second issue is the ability of the Apache server to display server status information dynamically to administrators. This is done so that an administrator may determine the system load, current connections, recent history statistics, and related information. This may be done even if the administrator is at a remote machine and cannot actually view the server's log files. Unfortunately, unless carefully configured, this information can also be available to any client on the Internet. This functionality is provided by the *mod_status* module. The *mod_status* module is part of the default Apache build. The default Apache configuration files do not contain directives that enable this functionality. As a result, initially, clients will still not be able to acquire this information. However, since the module is part of the Apache build, any users who can create .htaccess files on the server will be able to create configurations causing server status information to be displayed. If the administrator is uncomfortable about letting this information be distributed in an uncontrolled fashion, they should disable the module by adding "**`--disable-module=status`**" as an argument to the **`configure`** command.

The final issue is the ability of the Apache server to display its modules and configuration directives upon request. This functionality is provided by the *mod_info* status. This module is not part of the default Apache build. When this module is added and directives added to the configuration files, the server can be requested for the current configuration of the Apache server. This information includes a list of all the modules that were part of the build which created the server, as well as all the directives used from each of these modules and any parameters these directives have. This information displays the security settings, gives away any aliasing and redirection the administrator may have enabled, and provides the location of all important files on the server. This practically provides a roadmap for anyone who might wish to perform hostile actions on the server. Administrators should be extremely wary of enabling this module since, even if its directives are not added to the main configuration files, users can still enable it in individual .htaccss files. As such, the authors recommend that *mod_info* never be enabled on any server which stands a chance of being attacked (i.e., one that is attached to a network).

Section 4

Recommendations and Summary

The Apache server is robust and resistant to most exploits. It contains a wide range of features that allow for a high level of customization. This degree of flexibility makes becoming an Apache expert a long term endeavor. This allows administrators to modify the version of the Apache server being run to reflect the precise needs of their own system.

The Apache Web Server also contains several features that can be used to make it more secure. The server can implement access control based on either host identity or user authentication. It is capable of detailed auditing of events on the server and allows the auditing mechanism to be customized to meet specific needs of a web site's administrator. The Apache server also is capable of a high degree of control over how it handles user requests. This control can be used to ensure accessibility and to conserve server resources as appropriate. The server is capable of hiding its own internal structure from users in order to prevent the internal system from being mapped out by attackers.

Another advantage of the Apache Web Server is that the source code is available to all users. Due to this, the server\version is updated frequently; the server has been updated seven times since this assessment began a year ago and any serious bugs in the system are usually found quickly and fixed. The rapidity of new features and patches makes it unlikely that any security holes would remain a problem for an extended period of time.

During this study no security problems were detected with the Apache Web Server. However, the server lacks particular features that prevent it from being usable on secure systems. Specifically, it lacks encryption, which is necessary to create a truly secure web server. When the Apache Web Server was being designed, its authors made the decision that they wished the software to be as widely usable as possible. To avoid the difficulties associated with export controls, the Apache creators decided that the standard Apache server would not include encryption. The result of this decision is that secure information cannot be protected by the Apache Web Server. Users authenticating to the server can have their usernames and passwords acquired by an attacker under most circumstances. There is no guarantee that the server's response will not be altered by an attacker before reaching its intended recipient. Anyone with the ability to listen to the wire will be able to read every document returned by the server no matter how tightly the server controls its access.

Programmers have developed an extension of the Apache web server called Apache SSL. Apache SSL, which is distributed from outside the United States, contains confidentiality mechanisms that allow for encrypted connections between clients and servers. In addition to protecting client requests and server responses, the added functionality can secure authentication attempts against eavesdropping and replay. Apache SSL is legal for commercial use within the United States providing one has acquired a license for RSA encryption from the RSA Corporation. Apache SSL is based on the standard Apache web server and, while this project has not looked at Apache SSL, it appears that all functionality of the original Apache server is present in Apache SSL using the same configuration directives.

Currently planned for the next fiscal year is a task to extend this year's assessment of the Apache Web Server to include an assessment of the Apache SSL Web Server. It is hoped that, the latter application will prove as robust and full of features as the original Apache Web Server, while the addition of encryption technology will make the server appropriate for use in handling secure documents.

Glossary

API	Application Programming Interface
CGI	Common Gateway Interface
DNS	Domain Naming System
FTP	File Transfer Protocol
HTMP	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JVM	Java Virtual Machine
SGML	Standard Generalized Markup Language
SMTP	Simple Mail Transfer Protocol
SSI	Server Side Includes
TCP	Transmission Control Protocol
URL	Universal Resource Locator
WWW	World Wide Web

Changes

Rev 1.1. Revised to include the fact that the latest versions of Internet Explorer and Netscape Communication support digest authentication. Corrected a few typos which had no effect on technical content.

Rev 1.11 Added a statement reminding the reader that digest authentication has some security shortcomings as identified in the RFC.

Rev 1.12 Added a warnings page.

Distribution List

Internal

G020

D. J. Bodeau
H. W. Neugent
W. R. Gerhart
P. S. Tasker

G021

J. L. Connolly
J. D. Guttman
R. M. McQuaid
M. C. Michaud
J. Picciotto
C. M. Schmidt

External

National Security Agency
9800 Savage Road, STE 6704
Ft. George G. Meade, MD 20755

M. Pittelli, NSA/C43