Global Information Assurance Certification Paper

Apache modules for rapid mitigation of security threats

GIAC Security Essentials Certification (GSEC) Practical Assignment Version 1.4c

Option 1 - Research on Topics in Information Security

Submitted by: Stephanie Sullivan
Location: Boston, Massachusetts

Abstract

Netcraft[1] reports that nearly 70% of web sites are being served by the Apache web server. The sheer numbers make these tempting targets for attackers. In many instances the time to respond to vulnerabilities by rebuilding a patched version of Apache may be longer than desirable – for a number of technical or procedural reasons.

Using of the modular nature of Apache and its API a number of threats may be mitigated until updated executable images may be made available and installed. This document will explore how the Apache API may be used to rapidly mitigate certain threats without rebuilding or patching the Apache binary. This technique may also be used to mitigate certain issues in other software embedded in Apache and web applications. This approach is intended to address the period from when a vulnerability becomes known and when a "real" fix may be applied.

[1] Netcraft: January 2005 Web Server Survey, 2005, Netcraft web server survey for from August 1995 through January 2005, Retreived January 2005 <http://news.netcraft.com/archives/2005/01/01/january_2005_web_server_survey.html >

Contents

List of Figures

Introduction

The time from the discovery of a vulnerability until exploits appear in the wild is shrinking. Rapid mitigation of vulnerabilities in Apache, extension modules, and down stream web application code is increasingly critical. Apache's modular structure may provide a solution in many cases.

The goal of this paper is to provide help to the interested security professionals to best leverage Apache modules within the context of detecting and mitigating security threats, including recognizing when a module approach may be most appropriate.

The intimate details of the Apache API in depth, the Perl language, or how to configure/install mod_perl are already well covered in available texts and many resources on the internet and are not covered in this paper.

The widely deployed Apache web server is a large and complex service running on multiple operating hardware and software platforms. With any large system there is the chance for error or oversight in the creation of the software creating the potential for a exploitable vulnerability. The vulnerabilities specific to Apache are chronicled over time in the web journal Apache Week[2, 3].

The scope of a threat is not limited to Apache itself. The Apache web server is modular in design, and many extensions may be added to the core web server to provide additional capabilities. These modules extend the scope of the code base accessible through Apache and may have vulnerabilities remotely exploitable via Apache. These include numerous applications and services such as databases, languages and encryption. Further, web applications that are not part of Apache itself may be exploited remotely though Apache such as (but not limited to) PHP or CGI based application interfaces.

In contrast to the need for rapid threat response many organizations have testing and change management processes that must be followed. They take great care to maintain a stable and well understood production environment. However, these processes can take precious time for re-qualification. This is especially so with Apache where many modules and web applications may be affected by rebuilding the executable image.

Another example is where a web server is built into a server appliance. These devices may depend upon vendor supplied binaries with proprietary extensions. In these

[2] Cox, Mark J., "Overview of security vulnerabilities in Apache httpd 1.3", Apache Week, 22nd October 2004. Retrieved February 2005, <http://www.apacheweek.com/features/security-13>

[3] Cox, Mark J., "Overview of security vulnerabilities in Apache httpd 2.0", Apache Week, 22nd October 2004. Retrieved February 2005, <http://www.apacheweek.com/features/security-20>

environments the appliance manager may be waiting for months for the vendor to supply a patch.

Where widely deployed web applications may be at fault, an ISP may need to mitigate a problem while minimizing impact upon customers. This approach may be used to provide a reasonable window for customers to upgrade affected applications to secure versions.

Apache Modules

Modular Structure

One of the key strengths of Apache is its modular structure. This enables the many extension modules that are available via open source and commercial sources. This also makes it possible to build extensions to mitigate a number of security threats.

The Apache web server is the base for many web applications, tools and languages used to enhance web site function and creation. Some of the best known of these include Java (Tomcat), Perl, PHP, Python, Cold Fusion, OpenSSL, and FrontPage.

The Apache extensions described earlier are implemented as modules or handlers using the Apache Application Programming Interface (API). This interface provides access to an Apache request at each phase of processing. The Apache API also provides the hooks needed to aid in the module-based mitigation of certain threats.

Module-Based Mitigation

The Apache API provides a means to integrate code dynamically into the web server. By creating a small amount of code in Perl or C as an Apache module, a range of threats may be mitigated without having to patch and rebuild Apache.

Apache modules can be small and quick to code; sometimes less than 14 lines in length. These small modules may be faster and easier to move through change control and testing processes than re-qualifying a patched Apache build. This helps reduce the time to mitigate a threat and therefore the potential exposure.

Apache Versions V1.3 vs. V2.0+

The Apache web server is widely deployed with many established sites in two supported versions: V1.3 and V2.0. V2.0 has been released for a few years and is growing in popularity.

Apache V1.3 has a single threaded coding model which spawns worker processes to handle web requests. The various V1.3 servers have been available for many years and are well understood workhorses that are very widely deployed.

Apache V2.0 has Multi-Processing support including an expanded API for multi-threaded server options. The new API is not completely backward compatible with the

V1.3 API. Information on porting Apache V1.3 modules to Apache V2.0[4] is available on the perl.Apache.org web site.

Given its very broad base and the focus of this paper on installations that are difficult to patch/rebuild this paper will focus Apache V1.3. The concepts and examples presented can be moved forward to the V2.0 API. The Apache web site provides developer documentation[5] to assist in doing this in addition to the mod_perl web site.

Applicable Threats

Most external threats to Apache and downstream code are based upon crafted web requests using the http protocol. Many of these may be handled by detecting their characteristic signatures as found in the http request.

Apache logs can often provide source for patterns of Universal Resource Identifier (URI) based attacks. Documentation of known exploits can be sources of information to quickly evaluate if a given threat is applicable for simple module based mitigation.

A recent example of an applicable threat is the PHPBB Viewtopic.PHP PHP Script Injection Vulnerability[6] exploit. This threat has an active exploit and may be mitigated with an Apache module that examines the request URI. By using knowledge of how the threat may present it may then be detected and rejected before reaching PHP and therefore PHPBB.

When evaluating threats based upon URI patterns or request characteristics it is important to recognize that care must be used in crafting detection so false positives are minimized while not missing actual exploit attempts.

Apache modules are one approach and as can be seen in many complex systems there may be other methods to mitigate a threat including modification of the Apache configuration to limit access to certain files that may be vulnerable or limiting request methods. When such configuration changes or other mechanisms meet the threat sufficiently without modifying Apache they should be used. The decision of whether to use Apache modules should be made in context of the other tools that may be available.

Module Mechanics

Apache modules can be implemented in many languages, but are most often written in

[4] A Reference to mod_perl 1.0 to mod_perl 2.0 Migration, mod_perl project of the Apache Software Foundation, 2005, Retreived February 2005, <http://perl.apache.org/docs/2.0/user/porting/compat.html>

[5] Apache HTTP Server Version 1.3 Documentation, The Apache Software Foundation, Retrieved February 2005, <http://httpd.apache.org/docs>

[6] "Apache PHPBB Viewtopic.PHP PHP Script Injection Vulnerability", SecurityFocus, December 30 2004, Retreived February 2005, <http://www.securityfocus.com/bid/10701/discussion/>

C or Perl. C executes faster but Perl is usually faster to write, an advantage in the context of threat mitigation. Other advantages of Perl are the built-in memory management, powerful string manipulation and regular expressions.

Modules written in Perl may be external or embedded as source code within Apache configuration files. Modules written in Perl can be very concise and take advantage of the powerful pattern matching and string manipulation capabilities of the Perl language. As such Perl may be more accessible to the security professional who is not a programmer.

As the goal is to provide rapid mitigation the focus will be on Perl that is embedded within the Apache configuration files. This minimizes the time to implementation and facilitates rapid testing. All of the Apache API principles discussed in Perl may be applied equally to a C module allowing for the differences in language constructs and that C can only make external or compiled-in modules.

Have mod_perl?

The use of Perl requires that Apache have mod_Perl[7] installed and configured to allow access to the Apache API. Although mod_perl is not one of the modules installed by the default Apache installation it is widely deployed[8]

To use modules in perl requires that mod_perl either be compiled into or dynamically added into the Apache configuration files. In a V1.3 implementation a compiled-in mod_perl may be found on a UNIX or Linux system by using the grep command to filter the output of the –L option of the httpd command for mod_perl. The following is the output of a V1.3 Apache with mod_perl compiled in:

```
# httpd -L |grep PerlScript
PerlScript (mod_perl.c)
```

To determine if mod_perl is loaded dynamically in Apache V1.3 use grep to search the httpd.conf file for the AddModule directive for mod_perl. In this case it is not loaded dynamically. If it were the AddModule directive below would have been uncommented.

```
# grep 'AddModule mod_perl.c' /etc/httpd/conf/httpd.conf
# AddModule mod_perl.c
```

Apache Requests

The Apache API is a window into the Apache request flow. The phases of the request are the points in processing where modules and handlers may plug into the flow.

[7] Welcome to the mod_perl world, mod_perl project of the Apache Software Foundation, 2005, Retreived February 2005, < http://perl.apache.org/>
[8] Netcraft's mod_perl statistics, mod_perl project of the Apache Software Foundation, 2005, Retreived February 2005, <http://perl.apache.org/outstanding/stats/netcraft.html>

Differing, and generally increasing, amounts of information are available to a module at each progressive phase because more processing has been accomplished on the request. It is important that careful consideration of the Apache request phases go into the decision of which phase should host a security module.

It is best to aim for the earliest possible phase of the request cycle where a threat may be detected. It is important to keep in mind that the detection and mitigation must happen in the request cycle before the threat may be realized. A good understanding of what processing takes place at each phase of the request flow is needed to make a good decision in module placement within the request cycle.

Request Processing Phases

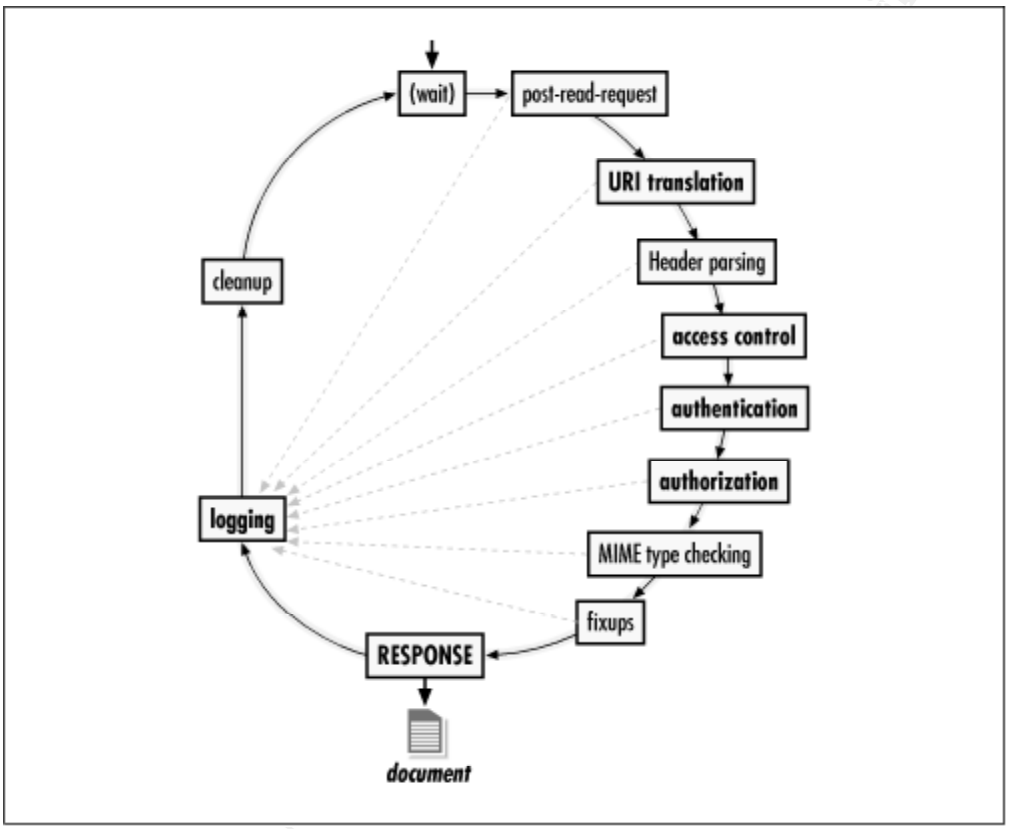

Figure 1: Apache request phases diagram[9].

There are 11 phases of request processing as summarized below. These are the same for Apache 1.3.* and Apache 2.0 and later.

Post-Read-Request

In this phase, all information in the request has been read and the header fields have been parsed into easily accessed fields. As it is the earliest point in the request cycle a client request is available, many exploits detectable via request pattern matching of

[9] Stein, Page 60. Illustration used with permission of O'Reilly and Associates

request elements may be addressed here.

URI translation
The URI (Universal Resource Identifier), which is also commonly referred to as a URL (Universal Resource Locator), gets translated at an actual file or a virtual file. Actual files exist as files in the files system and a virtual file is created programmatically. An example of a virtual file would be the a URI that passes a request to mod_status[10]

Note that enabling mod_status on a live server should be discouraged or done with great care as it may provide information about server configuration and/or usage patterns to a potential attacker.

Header parsing
During this phase request header fields may be examined and modified. Since this is after the URI translation and the file and location are known, processing beginning at this phase may be embedded within <directory> and <file> virtual containers or invoked from .htaccess files.

Incoming request headers may also be examined earlier during the Post-Read-Request phase. In this phase, however in the header parsing phase a second pass at the headers is available with the file and location information then available.

Access control
Limiting access based upon IP address or other factors than user identity occur during this phase. A module here has the option of forbidding access and ending the request, passing on processing to the next module in this phase (or to the next phase), or OK'ing the request to the next phase.

Authentication
This phase is where Apache modules handle authentication of users. This includes the issuing of challenges and handling of responses. There are multiple core and third party authentication modules for Apache.

Authorization
Authorization to access restricted resources such as protected directories is handled in this phase. Information gathered from the authentication phase is available for processing here.

MIME Type Checking
How a file is handled is determined in this phase. Different types of files are handled differently by Apache. For example, a CGI script would be processed in this phase. Other handlers may also be available here. How a given request is processed at this

[10] Module mod_status, The Apache Software Foundation, Retrieved February 2005, < http://httpd.apache.org/docs/mod/mod_status.html>

point is determined by a combination of MIME type definitions and configuration directives.

Fixups
This is where modules may hook into the transaction flow to perform final request adjustment before the request handler is invoked.

Response
This is where the actual request gets processed including invoking any handlers for the request.

Logging
This is the point where the log entry for the request is generated. Any phase may trigger a log entry. By triggering a log entry within a security module, selective additional information may be logged to aid in analysis of attack/intrusion attempts.

Cleanup
In this phase modules can register to be called to release allocated and or locked resources. Mod_perl takes care of memory it allocates transparently for Perl modules. Explicitly allocated resources or locked structures should be explicitly freed.

The Request Object

The request object is central to programming Apache modules. It gives us the context of the current request including all Apache knows about it at the current point in the request phase. The request object also provides a plethora of mechanisms for many different kinds of operations including:

- Client request methods provide the means for retrieving and/or changing information from/in the client request. These include
- Access control methods provide the means to get information based on the per-directory type of control directives such as allow, deny, type, requires, etc. In addition it provides the means for determining if authentication is required and informing Apache that attempted authentication failed.
- Server response methods control header fields of the server response as well as other functionality including compression and language
- Content handling methods are the functions that control sending information to a client such as the content of web pages.
- Logging methods provide the ability to generate log entries of various types and severity.
- Apache configuration methods allow inspection and, in many cases, modification of the Apache configuration dynamically
- Apache core methods which include redirection and handling of sub-requests
- mod_perl specific methods provide functions that enable the management of handlers and miscellaneous functions that do not fit elsewhere such as the Apache::exit() function to terminate the current Perl module without terminating

the Apache child process

Client Request Information

Handling threats may require methods from any of groupings described above. In most instances early detection may be accomplished using Client request methods.

Threats presenting themselves through requests to Apache may be able to be detected through identifiable signatures within their requests. The following are some examples of different types of detectable threats:

The Apache Chunked Encoding[11] threat is avoidable by examining the "Transfer-Encoding" client request header field and refusing to serve requests that requested "Chunked" encoding.

The more recent Apache Mod_Proxy Remote Negative Content-Length Buffer Overflow Vulnerability[12] may be detected by checking the "Content-length" client request header field for a negative value and rejecting such requests.

The PHPBB Viewtopic.PHP PHP Script Injection Vulnerability is detectable using Perl regular expressions to examine the URI of requests. Because of specific characteristics of the URI needed to exploit this vulnerability it may be mitigated with a low probability of false positives.

Client request methods provide access to the critical parts of an http request during the early post-read-request phase as used in the examples above. Not all fields contain information during the post-read-request and are setup in later request processing phases.

The methods available from the Client request object are described in detail[13] on the mod_perl web site. Below a useful subset is summarized.

- **method** – this method provides the type of http request being performed such as GET, POST, PUT, etc.
- **the_request** - provides the unparsed request line sent by the client. This may be convienient for logging the entire request for later analysis
- **proxyreq** – this method is true if the request is a proxy http request

[11] "Apache Chunked-Encoding Memory Corruption Vulnerability", SecurityFocus, September 22, 2004, Retreived February 2005, <http://www.securityfocus.com/bid/5033/discussion>

[12] "Apache Mod_Proxy Remote Negative Content-Length Buffer Overflow Vulnerability", SecurityFocus, January 29 2005, Retreived February 2005, <http://www.securityfocus.com/bid/10508/discussion>

[13] Apache - Perl interface to the Apache server API, mod_perl project of the Apache Software Foundation, 2005, Retreived February 2005, <http://perl.apache.org/docs/1.0/api/Client_Request_Parameters>

- **protocol** – identifies the http protocol version. Typically "HTTP 1.0" or "HTTP 1.1"
- **hostname** – identifies the host being sought by the request. The HTTP 1.1 protocol allows for multiple web hosts on a single IP address and this method provides the targeted domain to be identified. This information is also available with the headers_in and the header_in methods.This is not available with an HTTP 1.0 request
- **uri** – provides the request URI. It does not include any query string
- **filename** – In the URI translation phase this becomes the filename to which the URI refers
- **path_info** – after the URI translation phase this method returns the part of the path left over after the file is located
- **args** – returns the optional arguments of a GET request – the portion that comes after the "?" in the request. The results are different depending on the calling context. In scalar context the unparsed query string is returned. In hash context a set of key/value pairs is returned. Note that if multiple input fields of the same name are present only the last will be present in hash context
- **headers_in** – returns a hash of all the headers in the client request.
- **header_in** – this method takes a case-insensitive argument of a header field name and returns the content for that field if set. The case of header names may vary between browsers and this method makes specific header checks simpler than walking a hash of all headers as returned by headers_in.
- **content** – returns the POST method content as is typical for form input in a web page. In hash context, when the content-type is application/x-www-form-urlencoded, a hash of the key/value pairs representing the POST information in the request. For other content types the method may be used in a scalar context to obtain the unparsed post data as a string. In this case the module code is responsible for parsing this data. This method will return information with the first call, but nothing after.
- **get_remote_host** – tries to obtain the remote host name through DNS lookup and if not available it returns the IP address in dotted format. Optional arguments (defined in Apache::Constants) are available to modify how lookups are performed:
    - REMOTE_HOST – if the DNS lookup of the domain name fails or the HostNameLookups Apache configuration directive is off the result will be the unset value. Otherwise the host name will be returned
    - REMOTE_NAME – this is the default lookup as described above
    - REMOTE_NOLOOKUP – if Apache has cached the host name from a previous lookup it is returned, otherwise the dotted IP address is returned. No DNS lookup is performed
    - REMOTE_DOUBLE_KEY – this argument causes a DNS lookup of the IP followed by a lookup of the resulting domain name. If the final IP address matches the original the host name is returned. If it does not the undef value is returned.

Error Logging

An important part of threat detection and mitigation is leaving information about the threat, its source, time and circumstances appropriate for later analysis. Mod_perl and the Apache API have a number of methods to create differing severities of error log entries.

In generating error log entries it is important to note that there are log methods defined for the request object ($r->log;) and the server object ($r->server->log;). The request object log method logs the requesting IP address but the server log does not.

The three methods of the request log object are:

- **log_error** – creates an error log entry with timestamp, source IP, and as an argument takes a string as the error message.
- **warn** – this method is like the log_error, but comes with a severity of warning. As there are relatively few warnings in Apache error logs this can help in sorting out the log entries.
- **log_reason** – makes a log entry like log_error but with expanded information
- **as_string** – takes the Client request and Server response headers and converts them into a multi-line string. When used with the above logging methods it provides a convenient way place all request header information into a string: both incoming header and any outgoing headers that have been set when invoked.

  When logged the newlines in the string may be replaced with a non-line-breaking character such as a bang (“!”) as in the example below (logged using with the warn method).

```
[Wed Feb 16 19:08:57 2005] [warn] [client 999.999.999.999]
Header Dump: !GET /zzzz.html HTTP/1.1!Accept: image/gif,
image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-
excel, application/vnd.ms-powerpoint, application/msword,
application/x-shockwave-flash, */*!Accept-Encoding: gzip,
deflate!Accept-Language: en-us!Connection: keep-alive!Host:
www.zzzzzzzz.zzz!User-Agent: Mozilla/4.0 (compatible; MSIE
6.0; Windows NT 5.1; .NET CLR 1.0.3705; .NET CLR 1.1.4322)!X-
Forwarded-For: 999.999.999.999!!HTTP/1.1 (null)!!
```

Returning Status to Apache

When processing within a module is completed status must be returned to Apache. The status returned determines the next actions Apache will take with the request. One of 4 scenarios will ensue:

1) The request is terminated with a specific HTTP status code. This ends the processing and moves directly to the Response phase to return the specified error to the requestor. The codes are the 200, 300, 400, and 500 series codes used in http processing and defined by the RFC2616[14].

Commonly Used HTTP Error Returns		
Numeric	**Symbolic**	**Meaning to the Client**
200	HTTP_OK	The document will follow, all OK
301	HTTP_MOVED_PERMANENTLY	The document has been relocated permanently. The location response header field will contain the new location
302	HTTP_MOVED_TEMPORARILY	The document has been relocated temporarily. The location response header field will contain the new location
400	HTTP_BAD_REQUEST	The request sent by the client was malformed
401	HTTP_UNAUTHORIZED	The requested resource requires authentication
403	HTTP_FORBIDDEN	The server refuses to fulfill the request
404	HTTP_NOT_FOUND	The requested document was not located
405	HTTP_METHOD_NOT_ALLOWED	The method (GET, PUT, POST, etc.) used for this request is not allowed.
406	HTTP_NOT_ACCEPTABLE	The possible server response to the user request are not within the types the client is able to accept
500	HTTP_INTERNAL_SERVER_ERROR	The server encountered an internal error condition
503	HTTP_SERVICE_UNAVAILABLE	The server is unable to handle this request temporarily

2) Continue processing the request normally with the next module of the current phase or move to the next phase by returning a success (symbolically “OK”) status.
3) Decline processing of the request, and let the request continue normally with the next module of the current phase or move to the next phase (symbolically “DECLINED”).

[14] Fielding, R., et al., “Hypertext Transfer Protocol -- HTTP/1.1”, The Internet Society, June 1999, Retrieved February 2005, <ftp://ftp.isi.edu/in-notes/rfc2616.txt>

4) Terminate further processing and the TCP connection of the request without generating an error. Symbolically “DONE”.

These status codes are defined in the Apache::Constants Perl module.

Examples

Many programming exercises leverage a (hopefully) good piece of example code as the starting point. In the following examples the clear and concise blowchunks module, written by Cris Bailiff[15], and used with his permission is the basis for the examples in this paper. The full source code of the Perl and C versions are included in the appendix.

Anatomy of BlowChunks

The following example is an analysis of the BlowChunks module code. This module will be examined in detail. It forms the basis of the following two examples which follow and with small modification address different vulnerabilities.

The BlowChunks code consists of 11 lines of Perl and 3 lines of Apache configuration. The source below is meant to be inserted into the Apache V1.3 httpd.conf configuration file.

The following has been reformatted for readability and line numbers added for reference.

```
1.    <perl>
2.    use Apache::Log;
3.    sub Awayweb::BlowChunks::handler {
4.      my $r = shift;
5.      if (join('',$r->headers_in->get('Transfer-Encoding')) =~
      m/chunked/i)
6.           {
7.           $r->log->warn('Transfer-Encoding: chunked - denied and
      logged');
8.           return 400
9.           }
10.   return 0
11.   }
12.   </perl>
13.   PerlPostReadRequestHandler Awayweb::BlowChunks
```

Lines 1 and 12 are the Apache configuration container for the module’s Perl code which is inserted between these tags.

Line 13 is an Apache configuration directive that places the Awayweb::BlowChunks module in the request processing loop at the Post Read Request phase – the earliest point in the cycle of request processing. The argument of the directive is the name of

[15] Bailiff, Cris, Source code to the BlowChunks Apache Module, July 2002, Retreived February 2005, <http://www.awayweb.com/pub/src/>

the module – the first 2 elements of the name of the subroutine that makes up the body.

Now to the Perl code. Line 2 is the "use" statement. This brings in a Perl module and makes its functions and definitions available to the calling module. In this case the Apache::Log module brings in the logging methods to extend the request object.

Line 2 is the beginning of the Perl code. It brings in a Perl module which is part of mod_perl. The Apache::Log module make logging routines available to the module. Another useful module to include in many instances is Apache::Constants which defines many symbolic values making code more maintainable and readable.

Line 3 is the beginning of the module subroutine. The Awayweb::BlowChunks:: part is the name defines the name of the module. The subroutine name is handler – the default name for the main routine of the module.

The subroutine contents are enclosed within matched curly braces opening at the end of this line and closing on line 11. The code between the brackets is where the work of the module is performed.

Line 4 is where the request object is retrieved.

Line 5 is an 'if' statement that uses the headers_in and get methods to retrieve the Transfer-Encoding header field contents and compare it with the word 'chunked' using a case-insensitive regular expression.

If line 5 matches a chunked encoded transfer has been attempted and lines 6 through 9 are executed logging the event on line 7 using the log->warn method of the request object. On line 8 a 400 status (bad request) is returned to Apache ending the module execution and causing the request to be rejected.

If line 5 does not match execution passes to line 10 where zero is returned indicating DECLINED status and allowing processing of the request to continue normally.

Module NegXfer

This example builds upon Blowchunks explained above to handle another threat detectable in a similar manner.

```
1.    <perl>
2.    use Apache::Log;
3.    sub Awayweb::NegXfer::handler {
4.      my $r = shift;
5.      if ($r->header_in->(' Content-Length') < 0)
6.          {
7.          $r->log->warn(' Request with negative Content-Length:
   Rejected');
8.          return 400
9.          }
10.   return 0
11.   }
12.   </perl>
13.   PerlPostReadRequestHandler Awayweb::NegXfer
```

In the above module lines 3, 5, 7 and 13 were modified from BlowChunks for this example. Lines 3 and 13 changes modify the name of the module so it is unique.

Line 5 uses the header_in method (fetches a single header field using a case-independent name) and checks for a length value less than zero. If this is the case lines 6 through 9 are performed.

Line 7 logs the event. The text of the message is modified to accurately reflect the event in the error log.

The remainder of the module operation is the same as BlowChunks.

PHPBBViewtopic module

This module again modifies the BlowChunks module to detect the PHPBB Viewtopic.PHP PHP Script Injection Vulnerability . This is detected using regular expressions on the URI and the get mode arguments based upon the exploit information from Security Focus vulnerabilities database.

```
1.    <perl>
2.    use Apache::Log;
3.    sub Awayweb::PHPBBViewtopic::handler {
4.       my $r = shift;
5.       my $u = $r->uri;
6.       my $a = $r->args;
7.       my $t = $r->the_request;
8.       if ($u =~ m/viewtopic.php$/ &&
9.           ($a =~ m/highlight.*%2527/ || $a =~ m/highlight.*SELECT%20/i))
10.         {
11.         $r->log->warn("Request:$t, PHPBB Viewtopic.PHP",
12.                       " vulnerability: Rejected");
13.           return 400
14.         }
15.      return 0
16.   }
17.
18.   </perl>
19.   PerlPostReadRequestHandler Awayweb::PHPBBViewtopic
```

In this example lines 3, 5 – 9, 11-12 and 19 are the new and modified lines. As in the previous example changes to 3 and 19 reflect the new name of the modified module.

Lines 5, 6 and 7 respectively get the URI, the GET method arguments, and the unparsed request line all from the request object. Note that since the arguments are assigned to a scalar they are stored as an unparsed string in the local variable $a.

Lines 8 and 9 are the 'if' statement testing the URI and args for patterns indicating a possible attack. They do two sets of regular expression checks: one on the URI for the vulnerable file (on line 8) and then a set of two checks on the GET method arguments for known exploit patterns (on line 9). If the args and URI checks both match lines 10 – 14 are executed.

Lines 11 and 12 log the event. The unparsed request collected in line 7 is embedded in the first string being logged. The second string is appended to the first in the log and is only separate to improve readability and avoid overly long lines.

As in the previous examples a 400 status (bad request) is returned on line 13. If there is no match the module returns the DECLINED value and processing of the request continues normally.

Below are two example exploits. Each are adapted from the Security Focus web site[16]. The resulting log entries that are generated follow each showing the triggering request

in detail.

Exploit 1

```
http://www.nnoweb.com/forums/viewtopic.php?t=[12343323]&highlight=Bug,SELEC
T * FROM $table
```

Log Entry 1

```
[Sun Feb 20 15:44:05 2005] [warn] [client 999.999.999.999] Request:GET
/forums/viewtopic.php?t=[12343323]&highlight=Bug,SELECT%20*%20FROM%20$table
HTTP/1.1, PHPBB Viewtopic.PHP vulnerability: Rejected
```

Exploit 2

```
http://www.example.com/viewtopic.php?t=29040&highlight=%2527%252esystem(chr
(108)%252echr(115))%252e%2527
```

Log Entry 2

```
 [Sun Feb 20 15:46:41 2005] [warn] [client 999.999.999.999] Request:GET
/viewtopic.php?t=29040&highlight=%2527%252esystem(chr(108)%252echr(115))%25
2e%2527 HTTP/1.1, PHPBB Viewtopic.PHP vulnerability: Rejected
```

It is very important that pattern matching be performed carefully and be thoroughly tested as false positives and negatives are a risk. This is especially important with the URI, GET or Put method arguments or user agent header field. The ability to quickly make changes and test them is another virtue of using mod_perl based modules.

Conclusions

Apache modules can aid in the timely mitigation of certain security threats, especially when Apache or the downstream code may not be patched or updated in a timely manner. The approaches presented in this paper are not intended to be the ultimate response for a given threat. Instead, they are intended to provide additional options to running exposed or shutting down until a more proper solution may be implemented.

Using mod_perl and with good example solutions a variety of threat mitigation may be constructed quickly. These can provide customized handling of a variety of threats. By using the flexible logging mechanism detected exploit attempts may be monitored for reference or further action.

Apache mod_perl modules are a valuable tool for the security professional. By reducing threat exposure security is improved. By adding a method to handle otherwise difficult situations quickly with minimal impact otherwise exposed systems may be better protected until more traditional patches may be applied minimizing threat impact.

[16] “Apache PHPBB Viewtopic.PHP PHP Script Injection Vulnerability”, SecurityFocus, December 30 2004, Retreived February 2005, <http://www.securityfocus.com/bid/10701/exploit/>

Appendix A: BlowChunks – a real-life example

The Apache chunked-encoding vulnerability[17] showed up in 2002. Chunked encoding is used to submit certain types of http requests as described in RFC2616[18] in section 3.6.1. These are usually incremental transfers of data from the client to the web server.

This vulnerability within Apache is a buffer overflow due to an error in buffer size arithmetic. The following workaround was written by Cris Bailiff and is used in this paper with his permission.

Approach to mitigation

For this vulnerability the approach taken was to prevent the use of chunked transfer encoding by issuing a HTTP_BAD_REQUEST (400) response to the client. As the use of chunked transfer is relatively limited this resulted in an effective workaround until the problem could be corrected within the core Apache image.

Perl Module Full Source

```
# $Id: BlowChunks.pl,v 1.1 2002/07/31 05:34:36 awxfer Exp $
#
# Reject chunked requests before vulnerable chunking routines can read
them.
# (mod_perl version)
#
# Cris Bailiff, c.bailiff+blowchunks@devsecure.com - http://www.awayweb.com
# http://www.devsecure.com/pub/src/BlowChunks.pl
#
# Copyright 2002 Cris Bailiff.  All rights reserved.
#
# Permission is granted to anyone to use this software for any purpose on
# any computer system, and to alter it and redistribute it, subject
# to the following restrictions:
#
# 1. The author is not responsible for the consequences of use of this
#  software, no matter how awful, even if they arise from flaws in it.
#
# 2. The origin of this software must not be misrepresented, either by
#  explicit claim or by omission.
#
# 3. Altered versions must be plainly marked as such, and must not be
#  misrepresented as being the original software.
#
# 4. This notice may not be removed or altered.
#
# To install in your mod_perl enabled server, copy the code below into
# your httpd.conf file (at the end is best), or read this file into
# your configuration using an 'Include' statement, and restart httpd.
```

[17] "Apache Chunked-Encoding Memory Corruption Vulnerability", SecurityFocus, September 22, 2004, Retreived February 2005, <http://www.securityfocus.com/bid/5033/discussion>

[18] Fielding, R., et al., "Hypertext Transfer Protocol -- HTTP/1.1", The Internet Society, June 1999, Retrieved February 2005, <ftp://ftp.isi.edu/in-notes/rfc2616.txt>

```
#
# You need mod_perl with support for PerlPostReadRequestHandler
# and <perl> sections. You have these if your mod_perl was configured
# using EVERYTHING=1, which is typical.
#
# (Permission is granted to leave these comments out of your httpd.conf
# file :-)
# but please use this original version if passing along...)
#
# --cut-here---

<perl>
# blowchunks for mod_perl
# $Id: BlowChunks.pl,v 1.1 2002/07/31 05:34:36 awxfer Exp $
# Deny requests using Transfer-Encoding: chunked
#
use Apache::Log;
sub Awayweb::BlowChunks::handler {
  my $r = shift;
  if (join('',$r->headers_in->get('Transfer-Encoding'))
        =~ m/chunked/i)
  {
      $r->log->warn('Transfer-Encoding: chunked - denied and logged');
      return 400
  }
  return 0
}
</perl>
PerlPostReadRequestHandler Awayweb::BlowChunks
```

C Module Full Source

```
/*
 * $Id: mod_blowchunks.c,v 1.1 2002/07/31 05:34:36 awxfer Exp $
 *
 * Reject chunked requests before vulnerable chunking routines can read
them.
 * (apache module version)
 *
 * Cris Bailiff, c.bailiff+blowchunks@devsecure.com -
http://www.awayweb.com
 * http://www.devsecure.com/pub/src/mod_blowchunks.c
 *
 * Copyright 2002 Cris Bailiff.  All rights reserved.
 *
 * Permission is granted to anyone to use this software for any purpose on
 * any computer system, and to alter it and redistribute it, subject
 * to the following restrictions:
 *
 * 1. The author is not responsible for the consequences of use of this
 *  software, no matter how awful, even if they arise from flaws in it.
 *
 * 2. The origin of this software must not be misrepresented, either by
 *  explicit claim or by omission.
 *
 * 3. Altered versions must be plainly marked as such, and must not be
 *  misrepresented as being the original software.
 *
```

```
 * 4. This notice may not be removed or altered.
 *
 * To compile & install in your apache (using apxs):
 *
 *     # /usr/sbin/apxs -i -a -c mod_blowchunks.c
 *
 * and restart. Read the apxs(8) man page for more info on compiling apache
 * modules.
 */

#include "httpd.h"
#include "http_config.h"
#include "http_core.h"
#include "http_log.h"
#include "http_main.h"
#include "http_protocol.h"

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

module MODULE_VAR_EXPORT blowchunks_module;

static int blowchunks_check_one_header(void *data, const char *key, const
char *val)
{
    if (ap_find_last_token(NULL, val, "chunked")) {
      *((int *)data)=TRUE;
      return FALSE;
    }
    return TRUE;
}

static int blowchunks_post_read_request(request_rec *r)
{
    int found=FALSE;
    ap_table_do(blowchunks_check_one_header,&found,r->headers_in,
          "Transfer-Encoding",NULL);
    if (found==TRUE) {
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
          "Transfer-Encoding: chunked - denied and logged");
      return HTTP_BAD_REQUEST;
    }
    return DECLINED;
}

module MODULE_VAR_EXPORT blowchunks_module =
{
    STANDARD_MODULE_STUFF,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
#if MODULE_MAGIC_NUMBER >= 19970902
    blowchunks_post_read_request
#else
#error Your apache is too old to have the post_read_request module hook
#endif
};
```

Appendix B:

ModSecurity for Apache[19]

"ModSecurity is an open source intrusion detection and prevention engine for web applications" - http://www.modsecurity.org/index.php . This project provides a comprehensive web application firewall. Where it is appropriate or practical to incorporate into an Apache environment many of the filters and actions discussed in this paper may be performed. As it is not so widely deployed as Apache or mod_perl it may not be available or may not be suitable for a particular environment.

mod_dosevasive[20]

This module provides a degree of protection from denial of service attacks. This is performed by maintaining metrics on how often a request from a given IP occurs for a specific page or resource and blocking IP addresses that have violated policy

This sort of defense is a kind not easily programmed in the kind of simple Apache modules described in this paper.

[19] Ristic , Ivan, "ModSecurity for Apache", ModSecurity, 2005, Retreived February 2005, <http://www.modsecurity.org/projects/modsecurity/apache/index.html>

[20] Zdziarski, Jonathan A., "mod_dosevasive v1.10", Nuclear Elephant.com 2003, Retreived February 2005, <http://www.nuclearelephant.com/projects/dosevasive/>

References

Stein, Lincoln, Mac Eachern, Doug, Writing Apache Modules with Perl and C. Sebastopol; O'Reilly and Associates,1999

Bekman, Stas, Cholet, Eric. Practical mod_perl, Sebastopol; O'Reilly and Associates, 2003

Mobily, Tim, Hardening Apache, Berkeley, Apress, 2004

Wall, Larry, Christiansen, Tom, Schwartz, Randal L., Programming Perl, 2nd edition, Sebastopol; O'Reilly and Associates, 1996

Bailiff, Cris, Source code to the BlowChunks Apache Module, July 2002, Retreived February 2005, <http://www.awayweb.com/pub/src/>

Netcraft: January 2005 Web Server Survey, 2005, Netcraft web server survey for from August 1995 through January 2005, Retreived January 2005 <http://news.netcraft.com/archives/2005/01/01/january_2005_web_server_survey.html>

A Reference to mod_perl 1.0 to mod_perl 2.0 Migration, mod_perl project of the Apache Software Foundation, 2005, Retreived February 2005, <http://perl.apache.org/docs/2.0/user/porting/compat.html>

Welcome to the mod_perl world, mod_perl project of the Apache Software Foundation, 2005, Retreived February 2005, < http://perl.apache.org/>

Apache - Perl interface to the Apache server API, mod_perl project of the Apache Software Foundation, 2005, Retreived February 2005, <http://perl.apache.org/docs/1.0/api/Client_Request_Parameters>

Netcraft's mod_perl statistics, mod_perl project of the Apache Software Foundation, 2005, Retreived February 2005, <http://perl.apache.org/outstanding/stats/netcraft.html>

Apache HTTP Server Version 1.3 Documentation, The Apache Software Foundation, Retrieved February 2005, <http://httpd.apache.org/docs>

Apache HTTP Server Version 2.0 Documentation, The Apache Software Foundation, Retrieved February 2005, <http://httpd.apache.org/docs-2.0/>

Module mod_status, The Apache Software Foundation, Retrieved February 2005, < http://httpd.apache.org/docs/mod/mod_status.html>

Cox, Mark J., "Overview of security vulnerabilities in Apache httpd 1.3", Apache Week, 22nd October 2004. Retrieved February 2005,

<http://www.apacheweek.com/features/security-13>

Cox, Mark J., "Overview of security vulnerabilities in Apache httpd 2.0", Apache Week, 22nd October 2004. Retrieved February 2005, <http://www.apacheweek.com/features/security-20>

Fielding, R., et al., "Hypertext Transfer Protocol -- HTTP/1.1", The Internet Society, June 1999, Retrieved February 2005, <ftp://ftp.isi.edu/in-notes/rfc2616.txt>

"Apache Chunked-Encoding Memory Corruption Vulnerability", SecurityFocus, September 22, 2004, Retreived February 2005, <http://www.securityfocus.com/bid/5033/discussion>

"Apache Mod_Proxy Remote Negative Content-Length Buffer Overflow Vulnerability", SecurityFocus, January 29 2005, Retreived February 2005, <http://www.securityfocus.com/bid/10508/discussion>

"Apache PHPBB Viewtopic.PHP PHP Script Injection Vulnerability", SecurityFocus, December 30 2004, Retreived February 2005, <http://www.securityfocus.com/bid/10701/discussion/>

"Suggested filters against PHP Attacking Worms", CastleCops, December 27, 2004, Retrieved January 2005, <http://castlecops.com/article5642.html>

Ristic , Ivan, "ModSecurity for Apache", ModSecurity, 2005, Retreived February 2005, <http://www.modsecurity.org/projects/modsecurity/apache/index.html

Stein, Lincoln, "Apache Security from A-Z", Open Source Conference, Version 3, July 16, 2000, Retreived February 2005, <http://stein.cshl.org/~lstein/talks/perl_conference/apache_security/>

Zdziarski, Jonathan A., "mod_dosevasive v1.10", Nuclear Elephant.com 2003, Retreived February 2005, <http://www.nuclearelephant.com/projects/dosevasive/>