HMAP: A Technique and Tool For Remote Identification of HTTP Servers

By

DUSTIN WILLIAM LEE

B.S. (Gonzaga University) 1990

THESIS
Submitted in partial satisfaction of the requirements for the degree of
MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

Professor Karl N. Levitt (Chair)

Associate Professor Matthew A. Bishop

Jeff Rowe, Ph.D.

Committee in Charge

2001

Contents

Acknowledgements

I would like to thank my Committee members, Karl Levitt, Matt Bishop, and Jeff Rowe for helpful discussions and comments. I also would like to thank Nick Puketza and Radu Teodorescu for their comments on early drafts.

In addition, I would like to thank the seclab crew for many interesting discussions (security related and otherwise). It might have taken less time without them but it would have been a lot less fun.

And finally, I would like to thank my lovely wife, Jodi Hedges, for occasionally feigning interest in my work, making significant writing suggestions and being an all around wonderful person to have in my life. I couldn't have done this without you.

Chapter 1

Introduction

"Note: Revealing the specific software version of the server might allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Server implementors are encouraged to make this field a configurable option." [RFC 2068: HTTP / 1.1 (section 14.38)]

"Eschew obfuscation." [Anonymous]

1.1 Overview

When an HTTP server[1] and an HTTP client[2] communicate with each other, they typically include a description of their vendor and version number in their messages. This information is found in the "Server" and "User-Agent" header fields respectively. Providing this information may seem unnecessary since one purpose of the HTTP protocol specification is to allow client and server to communicate even if they are developed by different vendors. Regardless, there are some benefits to including vendor specific information. For example, some client implementations have certain HTTP Response parsing bugs that can be worked around by appending "dummy" data to the Response's header section [18]. Since the client will not behave correctly if this work around is not implemented, it is obviously in the clients interest to give an accurate self-description.

A server advertising its vendor and version number is more problematic. Since vulnerability databases (e.g. SecurityFocus [13]) organize vulnerabilities and exploits by vendor and version number, an attacker can strike more efficiently if he has confidence in

[1]synonym for web server
[2]synonym for web browser

the identity of the target. This is why the RFC [2] cited above recommends providing the capability of obfuscating the server's identity.

Since HTTP servers do not generally provide a simple means for modifying or omitting the server's self-description (e.g. as a configuration file option) a server administrator can not easily hide this information. However, through modification of source code or even the binary (by employing some reverse engineering) it is possible to hide the server's identity. A small percentage of servers on the Internet do currently hide their identity by somehow omitting the Server header field or simply leaving it blank. This thesis will demonstrate that this method, while a small improvement, is not sufficient to hide a server's identity. In fact, it will be shown that it is a difficult endeavor to completely hide an HTTP server's identity. HTTP Responses contain information that provide clues to their identity. It is possible to compare the Responses of a target server against known server profiles and determine the target server's vendor and version number with a high degree of confidence.

1.2 Objectives and Contributions

The purpose of this thesis is twofold. The first objective is to describe a methodology for using HTTP message characteristics for determining a HTTP server's identity with a high degree of confidence. The proof of this concept will be in the form of a new tool called HMAP which automates this identity discovery. The second objective is to provide a systematic way that security personnel can use these results in a defensive manner to make their system less identifiable and to detect when outsiders are trying to probe for their server's identity.

1.3 Thesis Organization

First, relevant features of the HTTP protocol are discussed to provide a background for the rest of the thesis. An overview of related projects and tools follows. A methodology and the new tool HMAP that identifies web servers by analyzing HTTP transactions will be introduced. Finally, the security implications of this work will be discussed.

Chapter 2

HTTP Overview

To place this thesis into proper perspective one must first be familiar with some of the basic elements and structure of HTTP communication. Hypertext Transfer Protocol (HTTP) was developed by Tim Berners-Lee while he worked at CERN. In its initial incarnation HTTP/0.9 was used as a terse method of retrieving documents from a remote server without user authentication. The type of documents served were generally HTML (Hypertext Markup Language) or other simple text formats. The advent of different graphical browsers that allowed the user to easily view related pages quickly by way of hyperlinks added to the excitement of this new protocol. HTTP/1.0[1] was developed to address some of the short comings of the initial design and introduce new features. These included more sophisticated multimedia handling, support for less transient TCP/IP connections and web infrastructure elements such as proxies and gateways. In addition, cookies were introduced as a way to add some state to a stateless protocol[3]. Many of these additions were in common use in an *ad hoc* way before being added to the specification. Because features were constantly being added and adapted, many servers that claimed to be HTTP/1.0 compliant were so in name only. Within a few years, HTTP/1.1 was developed to add more features and correct other mistakes but conformance to this standard has also been problematic[5].

In the current Internet environment there is a fairly even mixture of clients and servers claiming to be HTTP/1.0 or HTTP/1.1 compliant and very few really being either. Despite this, the protocol is in heavy and growing use. Since most of the work done by HTTP is fairly simple, non-compliance to some of the more esoteric features does not interfere with day to day use. Even with the updates the fundamental idea of HTTP is straight forward. Each transaction consists of a Request initiated by a client and a reply by a server.

2.1 Request

A Request has the following format:

```
METHOD URL HTTP-Version
[Header: Field]*

[Body]
```

The METHOD is the command that the server will try to fulfill. For instance "GET" means retrieve the indicated resource. Common methods include: GET, HEAD, OPTIONS, TRACE and POST. The URL[1] (Uniform Resource Locator) designates which resource the method should act upon. The HTTP-Version indicates to the server what version of HTTP the client understands and therefore what type of protocol elements it can handle. This first line (method line) is followed by zero or more header lines and an optional body. The header lines provide information about the client software making the Request, including its preferences regarding the Response message and various other Request modifiers. The body generally won't be present in a Request unless a POST is being sent or some of the WebDAV[4] methods are being used. Here is an example Request for fetching a web page with a typical browser (Netscape)[21]:

```
GET /index.html HTTP/1.0
Referer: http://somewhere.org/
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.75 (Macintosh; U; PPC)
Host: www.noplace.org
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

The headers used above are only a small subset of the dozens of possible headers. Most of these fields are optional. One notable exception is the "Host" header which for HTTP/1.1 is a required field. These headers closely follow the well known MIME format[24]

[1]URI or Uniform Resource Indicator seems to be the preferred term these days.

2.2 Response

An HTTP Response has the following format:

```
HTTP-Version Response-Code Response-Message
[Header: Field]*

[Body]
```

The HTTP-Version informs the client what level of HTTP is being used so that it can interpret what follows in an appropriate manner. The Response-Code is a numeric value (currently in the range 1XX - 5XX) that describes the success or failure with respect to the Request handling. The Response-Message is a human readable description of the Response-Code. For instance “200” means that the Request was satisfied and is usually written as “OK”. “404” indicates that the Request could not be satisfied since the document was not found and is usually written as “Not Found”. Following this are a variety of headers (a few of which are required) and a body if appropriate. The following is a Response to a GET Request sent from an Apache server:

```
HTTP/1.1 200 OK
Date: Fri, 23 Feb 2001 22:06:03 GMT
Server: Apache/1.3.12 (Win32)
Content-Location: index.html.en
Vary: negotiate,accept-language
TCN: choice
Last-Modified: Thu, 18 Jan 2001 18:50:42 GMT
ETag: "0-574-3a673b02;3a6c9af5"
Accept-Ranges: bytes
Content-Length: 1396
Connection: close
Content-Type: text/html
Content-Language: en
Expires: Fri, 23 Feb 2001 22:06:03 GMT

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
 <HEAD>
```

```
  <TITLE>Test Page for Apache Installation</TITLE>
 </HEAD>
<!--Background white,links blue(unvisited),navy (visited),red (active)-->
 <BODY
  BGCOLOR="#FFFFFF"

et cetera ......
```

The Response headers provide meta-information about the entity being returned as well as information about the server. For instance, since the client usually specifies a number of different acceptable formats for the return data, the headers identify the specific format of the document actually returned. Above we see the name of the file being sent (index.html.en), it's length in bytes (1396) and type (text/html) as well as the language ("en" for English). An example of a required field according to the HTTP/1.1 specification is the "Server" field. While many Responses return HTML type data, any encoding (including raw bytes) is allowed in the body. As long as the client knows how to handle it, the host can send it. In general, there are no specifications on the sizes of any of these elements (e.g. header fields, body) though most implementations tend to impose them.

Response codes come in several varieties and range from the 100's to the 500's. 1XX codes are the "informational" range (e.g. "100 Continue") and provide status on a connection. This range of codes is only available at the HTTP/1.1 level. 2XX codes are the "successful" range (e.g. "201 Created") and are used to indicate that a Request was understood and successfully complied with. 3XX codes are the "redirection" range (e.g. "301 Moved Permanently") and are used to indicate that further action must be taken by a client before the Request can be satisfied. For instance, the client might get redirected to a different URL. 4XX codes are the "client error" range (e.g. "404 Not Found") and are used to indicate that the client made an error of some sort. 5XX codes are the "internal server error" range (e.g. 501 Created) and are used to indicate that a Request could not be satisfied because the server believes it has made an error somehow.

2.3 Miscellaneous

The elements of HTTP Requests and Responses are not normally authenticated. For instance, a server could return an HTML document but incorrectly claim it is a GIF.

Obviously, it is in the interest of the client and server to provide accurate information to each other so they can successfully communicate. This lack of authentication applies as well to the “Server” and “User-Agent” fields which hold the self-descriptions of the server and client respectively. It is possible to put any information in these fields regardless of its veracity.

It is interesting to note that many of the documents retrieved from the Internet do not actually exist until they are requested. Instead they are created on the fly (e.g. a search engine result page, personalized news pages). One should be skeptical about reports regarding the number of pages that comprise the Internet.

Chapter 3

Related Work

HMAP uses fingerprinting and HTTP manipulation to determine web server identities. While there are no other utilities that achieve the same results as HMAP there are other fingerprinting products and projects that work by using HTTP in non-standard ways.

3.1 Web Server Surveys

Netcraft [14] and SecuritySpace [15] are two organizations that perform periodic surveys of the web to determine the market share of various web servers. One feature available from their sites allows the user to determine what HTTP server and OS a particular site is running. While monitoring the TCP/IP traffic of a workstation, this feature was used and the following HTTP Request was observed:

```
HEAD / HTTP/1.1\r\n
Connection: close\r\n
Referer: http://www.netcraft.com/survey/\r\n\r\n
User-agent: Mozilla/4.0 (compatible; Netcraft WebServer Survey)\r\n
Host:  \r\n\r\n
```

Since most server's Responses have a "Server" header including the vendor information it is a simple matter to extract this information. Presumably this same method is used to perform their web-wide survey. As discussed, server information determined this way is not very reliable. In fact, some simple changes to Apache/1.3.12 (Win32) source code fooled their HTTP server identity checker. Interestingly Netcraft is able to deduce the OS as well. This information is most likely required from the TCP/IP level using techniques similar to those of NMAP.

3.2 NMAP

NMAP [7] is a well-known network probing tool. In its default mode it performs a simple port scan to determine which ports are active on a target system. With the -o option it attempts to detect the OS and its version number. It does this by using known bugs and vagaries of different vendors' implementations of the TCP/IP stack. Most OSes can be uniquely characterized by comparing the target's behavior with known profiles.

For instance the correct behavior on receiving an unexpected FIN flag is to ignore it and not respond. Several operating systems incorrectly reply with a RESET. Using a decision tree method the specific operating systems can be identified by a series of "questions and answers". Depending on the behavior observed (like the FIN probe just described) movement through the tree continues until a leaf is reached. In most cases this answer is unique.

The results obtained by this method are not guaranteed. It is entirely possible that the functionality of NMAP could be reverse engineered to create a TCP/IP stack that spoofs the behavior of another OS or of none at all. Going to this extreme is fairly unlikely since it is always risky to alter software that is working correctly (or correctly enough).

3.3 PRO-COW

Krishnamurthy, et al use a home-grown unreleased tool [5] to check web servers for compliance with HTTP/1.1 specifications. While this tool was not designed for security or identity probing purposes, it has some functionality similar to that of HMAP. For instance, it spoofs various activities of a client in order to run a series of tests. These tests check both for appropriate default behaviors as well as appropriate responses to error conditions. As an example HTTP/1.1 Requests MUST[1] [2] include a Host header identifying the domain that the URI pertains to. If a client doesn't include this field a "400 Bad Request" error should be returned.

As the PRO-COW project has demonstrated many servers are at different levels of compliance. Since these differences tend to be constant for a given vendor/version

[1]words describing specification conformance such as MUST and SHOULD appear as all capitals in RFCs

number combination this information could be used for gleaning clues about server identity, though they do not report using it in this manner.

3.4 whisker

whisker [8] uses unusual HTTP Requests to probe web servers for known CGI based vulnerabilities. By sending specially crafted HTTP Requests, whisker can analyze the Responses to check for the presence of vulnerable software. The Request variations employed tend to be exclusively in the Request line URL. A highly relevant feature of this tool is that it has been tuned to elude typical IDS monitoring procedures. For example, there are many semantically equivalent ways to rewrite a Request such as:

```
GET /cgi-bin/vulnerable.cgi HTTP/1.0\n\n
```

A string of 1 or more '/'s can be substituted for '/'s in the path

```
e.g. /path/file.cgi becomes ////path////file.cgi
```

Spurious '.'s can be added to increase the length of the path

```
e.g. /path/file.cgi becomes /./././././././././path/file.cgi
```

URL encoding can also be used. There are many other IDS eluding techniques. In general these modifications rely on the fact that IDSes typically do simple string matching and can not afford the time and complexity of searching for semantically equivalent forms.

While whisker uses information about the specific OS and web server being probed to tailor its attack search it relies on the server's self-description of its identity.

3.5 LWP

LIBWWW-PERL (or, more concisely, LWP [16]) is a collection of Perl packages for creating programs that use Internet protocols. Of most relevance to this thesis, it allows for simple and intuitive web client programming. Many of LWP's class names and interfaces influenced the internal design of HMAP. Initially, making HMAP an extension of the LWP framework was considered. This was not possible for a number of reasons. First of all LWP tries to clean up (by modifying the Response) when the server doesn't respond or makes what LWP considers a mistake. Since many of HMAP's tests elicit just these sorts of Responses it was difficult to get accurate details on how the server was behaving. In addition, the APIs did not allow creation of Requests and evaluation of Responses at a sufficiently low detail. A new framework supporting these and other programming needs was warranted.

Chapter 4

HMAP

A methodology for fingerprinting an HTTP server for the purpose of identification is described in this section. As a proof of concept, a Perl language tool called HMAP was developed to automate this process. The name HMAP was, of course, inspired by NMAP. Whereas NMAP manipulates and analyzes the TCP / IP layer to gather clues about an operating system, HMAP performs its work at the HTTP level.

4.1 Motivation

Before we examine how an HTTP server fingerprinting method might work it is instructive to review some of the vulnerabilities that HTTP servers are subject to:

- May 2000, bugtraq id: 1284, “Apache HTTP Server (win32) Root Directory Access Vulnerability”
  - Description: Using a valid but unusually long path name for a URL, the web server incorrectly believes that the file was not found and returns a directory listing by default, even if configured not to do so.
  - Affected servers: Apache Group Apache 1.3.9 win32, Apache Group Apache 1.3.6 win32, Apache Group Apache 1.3.12 win32, Apache Group Apache 1.3.11 win32, IBM HTTP Server 1.3.6.2 win32, IBM HTTP Server 1.3.3 win32
- September 2000, bugtraq id: 1642, “Microsoft NT 4.0 and IIS 4.0 Invalid URL Request DoS Vulnerability”
  - Description: Using a specially formed URL, a bug in inetinfo.exe causes it to make an invalid memory request. Web server services are not available until the service is restarted.
  - Affected server: Microsoft IIS 4.0 for Microsoft Windows NT 4.0
- June 2000, bugtraq id: 1393, “Netscape Enterprise Server for Netware Buffer Overflow Vulnerability”

- Description: A buffer overflow exists that can allow the execution of arbitrary code by using specially formed URLs.
- Affected servers: Netscape Enterprise Server for NetWare 4/5 5.0, Netscape Enterprise Server for NetWare 4/5 4.1.1

This list is a small subset of available vulnerabilities but are examples of the sorts of problems found in web servers currently in use. Each of these is serious in its own way. A buffer overflow which allows execution of arbitrary code is clearly dangerous, but even something as seemingly innocuous as retrieving a directory listing can be part of a larger attack. If the directory listing includes the user directories of valid users for the system, this information could then be used when doing password guessing or social engineering attacks. Thus, awareness and attention to HTTP server vulnerabilities are an important part of the security landscape. Since the specific details of each vulnerability are dependent on the vendor and version number combination, it is evident that hiding the identity of a server would be desirable.

How do attackers typically acquire a server's identity? One simple method to accomplish this is as follows:

```
echo -e "GET / HTTP/1.0\n\n" | nc www.someserver.com 80 | grep Server
```

This command simply pipes the string "GET / HTTP/1.0\n\n" to the netcat command which opens a TCP/IP connection on the default HTTP server port of the target server. The reply from the server is filtered by the grep command and only lines containing the text "Server" are displayed. In most cases this will yield something like the following:

```
Server: Apache/1.3.12 (Win32)
```

Generally this provides accurate information. Of course there is no guarantee that the server is telling the truth. In fact, it is just as easy to cobble together another "netcat" script on the server end that responds with any bogus Server header.

A natural first attempt at hiding the identity is to simply not send it in the first place. No server, to my knowledge, has this as a configurable feature (despite the quote at the beginning of this thesis) but with access to source code it is a trivial matter to change this, either by clearing it out or setting it to a fictitious value. Using a variation of this above "netcat" technique I have conducted a survey of the top 100 most visited web sites [20]. The following results indicate that about 5% of sites[1] use this sort of hiding method

[1]This includes only sites that are leaving the Server header blank or just not sending it at all. Others could be lying about who they are but HMAP has not been used against the others for verification

(i.e. not sending back server headers or leaving them blank). Several sites (not in the top 100) are also known to hide server header information. As I will demonstrate, this tactic will only work against an uninformed attacker. By methodically fingerprinting a server, its true identity can be determined with a high degree of confidence.

4.2 Testing Methodology

To generate a fingerprint for a server it is necessary to find a set of characteristics that will discriminate its use of HTTP from that of other servers. These differences will arise from variations in how closely a vendor followed the HTTP specifications. The specification for the different HTTP variations and features are found in RFCs[2]. These specifications are not enforced, rather they are agreed upon conventions. If a vendor doesn't comply with the specifications it risks incompatibility with other's software. Of course, if others deviate from the specification in the same way then this deviation itself becomes a *de facto* specification (e.g. cookies developed in this way with Netscape taking the lead).

The HTTP RFCs describe the features of the HTTP protocol in terms of different levels of compliance. The words MUST, SHOULD and MAY (and each of these in combination with NOT) are used to indicate the importance the RFC's authors placed on the various features. Implementations that are deficient in the MUST category are considered to be non-compliant. A given web server could work perfectly well in most cases and still be non-compliant. It is this variability of compliance among other things, that will allow HTTP servers to be fingerprinted.

Keeping in mind this variation in compliance, we can compile a list of characteristics to use for fingerprinting and to generate tests (i.e. HTTP Requests) that will provoke Responses exemplifying these different characteristics. Primarily, the characteristics list will be derived by examining how a server responds to Requests under normal and abnormal conditions. The distinction between "normal" and "abnormal" is not precise. "Normal" roughly corresponds to client Requests that are well-formed with respect to the RFC and expected to succeed (i.e. return error codes in the 200 or 300 range). "Abnormal" Requests are then the Requests that are not "normal". After observing actual traffic from various servers, certain characteristics were found to be more useful for discriminating than others. The list of characteristics discussed below does not contain all

[2]Request For Comment

possible identity discriminators but was sufficient for the servers that were tested. Other variations may also be developed that contribute to the complete fingerprinting picture.

4.3 Fingerprint Characteristics

Using the analogy of programming language structure classifications, the types of characteristics that an HTTP fingerprinting methodology incorporates can be divided into the following categories: lexical, syntactic and semantic.

- Lexical: the specific words, phrases and punctuation that are used in Responses.
- Syntactic: the ordering and context of words, phrases, headers and other elements.
- Semantic: the server's specific interpretation of a Request from among the possible interpretations.

Rather than show all fingerprinting characteristics in detail, some of the more representative ones will be examined. Others are shown in the HMAP implementation code and are variations of those discussed here. The characteristics used in practice to discriminate and identify various servers will be a subset of all possible characteristics. Enough information for discrimination can be gathered without looking at all characteristics. Furthermore, the classification of the various characteristics within the three categories is not unique. Some ambiguity can be tolerated as the classification's primary aim is simply to guide and organize the search for identifying characteristics

4.3.1 Lexical

The lexical characteristics category covers variations in the actual words / phrases used, capitalizations and punctuation. These differences tend to be conspicuous and are quite useful for fingerprinting.

Response Code Message

As mentioned earlier, an HTTP Response includes a numeric value describing the success or failure of Request satisfaction. For each of these error codes there is also human readable text. For instance, for the error code 404, Apache [17] reports "Not Found" whereas Microsoft IIS / 5.0 reports "Object Not Found". While some messages are fairly uniform across the various implementations (e.g. 200 rarely deviates from "OK") there is generally a fair amount of variation between different servers in this regard. This variation is manifest both in the actual words used and in the capitalization pattern (all initial capitals versus only first word capitalized is a common distinction, e.g. “Not Found” vs. “Not found”). Many of these error codes can be elicited by sending a properly constructed Request.

Header Wording

Generally headers must match those in the specification or the client will not be able to identify them and correctly interpret the Response. Regardless of this constraint, variation still occurs in the form of capitalization patterns. For instance, the header “Content-Length” is returned by some servers and “Content-length” is returned by others.

Line Terminators

Some servers use only “\n” to separate elements of the header while the RFC [2] specified behavior is to use “\r\n”. This sort of deviation is less common in servers under current development but can be found in some earlier offerings.

Server’s Name

While the purpose of HMAP is to be suspicious of the server’s reported identity, the server’s claim of its identity can still be used as one of the characteristics.

4.3.2 Syntactic

HTTP messages are required to have a certain structure. It would be very difficult for servers and clients to understand each other if they didn’t follow this structure. Nonetheless, variation in the ordering and format of Request elements like the headers and their contents is expected.

Header Ordering

The HTTP specifications categorize headers as one of three types: “entity” (e.g. “Allow”), “general” (e.g. “Date”) and “response” (e.g. “ETag”). Furthermore they suggest that HTTP Response header fields should be ordered as follows: “general” then “response” then “entity”. Different vendors follow this suggestion to various degrees but will still frequently differ in the ordering within these categories. For instance, Apache servers consistently place "Date" before "Server" while Netscape-FastTrack / 4.1 have these reversed. We can examine the ordering of the entire set of headers sent by a server.

List Ordering

There are many instances where the contents of a header will be a list of items. For instance when an OPTIONS method is sent in an HTTP Request, a list of allowed methods for the given URI are returned in an "Allow" header. The order of these elements tend to vary between servers. This is also true for the "Vary" header. Not all lists are necessarily useful for discriminating identity, however. For instance, the Content-Language header can identify more than one language type when a document contains more than one human language. The ordering may imply what languages are used in the page in order of percentage of content. In this case the ordering is configurable and not an artifact of the inner-workings of the server.

Formatting

Some elements of the header have formats that are variable or unspecified by the specification. For instance, the "ETag" header provides a unique identifier (like a hash) for a given document that can be used to determine if the client has seen this document already (among other uses). For instance Apache / 1.3.11 returns an ETag header with the following format: "0-574-38379154;3a5b7811". Jigsaw / 2.1.2 server returns ETags like the following: "mvanct:s0jndthg". Since there are no official guidelines for how ETags should be constructed and presented these tend to be a good characteristic for fingerprinting.

4.3.3 Semantic

When a server receives a Request and attempts to satisfy it, it has to decide on some interpretation of the meaning of the Request. Furthermore, when the server constructs the Response and assigns the return code it has to give its interpretation of whether or not the Request was satisfied properly and for what reason. It also has to decide what information to send back to the requester both in the form of Response line, headers and body.

Headers Present

Some Requests, malformed and otherwise, will cause the server to believe the requester is an HTTP/0.9 based client. Legal Responses to HTTP/0.9 clients can only include a body (i.e. no headers or Response line). The existence or non-existence of a header can be used to classify how a server interpreted a client Request.

Specific Headers Used

A server has a choice of headers to include in a Response. While some headers are required by the specification, most headers (e.g. ETag) are optional. For instance, upon a "501 Method Not Implemented" error Apache servers send an "Allow" header with a list of the allowed methods for the designated URI, while Jigsaw/2.1.2 does not.

Response Codes for *Ad Hoc* Requests

Even when most servers agree that a certain Request is malformed, they often assign it a different type of error. For instance sending the text stream "hi" (with no headers or other HTTP trappings) to an Apache server provokes a headerless Response whose message body warns that the method "hi" is not implemented. Microsoft IIS/5.0 replies with "400 Bad Request". This implies that Apache interprets the Request as a bad reply from a HTTP/0.9 client whereas Microsoft takes it as a malformed Request from an HTTP/1.X client.

There are a large number of possible variations for developing malformed test Requests. Primarily, we can munge the method line, munge the headers and decide whether or not to include a body. For instance, the method line normally contains a method, a URI and a version. Each of these is separated by "whitespace", nothing preceding the method and a line terminator immediately following the version. Creating malformed variations is a simple matter of misspelling, switching positions, omitting elements, varying the quantity and type of "whitespace" and so on. The following is a partial list of some of the ways that the method line alone can be varied.

```
[GET]
[GET /]
[GET / HTTP/999.99]
[GET / hhtp/999.99]
[GET / http/999.99]
```

```
[GET / HTTP/Q.9]
[GET / HTTP/9.Q]
[GET / HTTP/Q.Q]
[GET / HTTP/]
[GET/HTTP/1.0]
[HEAD /.\ HTTP/1.0]
[HEAD /asdfasdfasdfasdfasdf/../ HTTP/1.0]
[HEAD /asdfasdfasdfasdfasdf/.. HTTP/1.0]
[HEAD /./././././././././././././././ HTTP/1.0]
[HEAD    /       HTTP/1.0]
[HEAD ///////////// HTTP/1.0]
[Head / HTTP/1.0]
```

Different servers have very distinctive reactions to the full set. A fairly distinctive fingerprint can be developed simply by issuing these "naked" method lines one at a time against a target server.

Error Ranges

For Requests and Request objects of unusually large length (unusual being defined by the server) the HTTP / 1.1 specification provides the errors "413 Request Entity Too Large" and "414 Request-URI Too Long" respectively. If a binary search is performed to identify the length of the URL that initially provokes this error for a given server, it is clear that different servers have unique patterns of errors that they pass through for increasing sizes. For instance, for varied URL lengths:

Server	URL Length	Response
Apache / 1.3.12 (Win)	1-216	404 Not Found
	217-8176	403 Forbidden
	8177-up	414 Request-URI Too Large
Netscape-FastTrack / 4.1	1-4089	404 Not found
	4090-8123	500 Server Error
	8124-8176	413 Request Entity Too Large
	8177-up	400 Bad request

Not only do these two servers report that a length limit has been passed for different URL lengths but they also report them as different problems. Apache indicates that the URI itself is too large whereas Netscape reports that the entire Request is too large. There are many different tests that provide a similar type of behavior for monotonically increasing parameters. For instance, the number of headers or the length of individual headers can be varied to achieve similar identifying patterns.

4.4 Tool Usage

HMAP is a Perl language program that automatically performs a wide variety of tests like those described above. It records all of its interactions with the target server and compares all of the Responses with a list of known server characteristics. It's comparison method is fairly simple. For each test it tries to provoke a Response from the target. If it can't get a response (for instance some errors aren't even implemented in some servers) then it notes that it can't compare this characteristic. If it does get a Response then it determines if it was an exact match or not and scores accordingly. For each server that HMAP has a profile for it displays a final result of how many exact matches, misses and "don't knows" that were found. For instance, one line of the output might be:

```
Apache/1.3.11 (Win)        21:9:5
```

where the score in this case indicates that 21 characteristics were an exact match for Apache / 1.3.11 (Win), 9 were definitely different and 5 couldn't be determined. One score line would appear for each server that has been previously profiled. Whichever server (or set of servers) matched most closely by having the highest number of exact matches will be noted as such. Obviously if one attempts to fingerprint a server for which there is no pre-existing profile then we wouldn't expect to find a good match (unless the server happens to behave like another known server). The result of this test is that we can determine the closest matching server type for a target. This is not proof of identity but is a good indicator of such.

It should be obvious that it is not possible to discriminate between two different server instances of the same type. For instance, two Apache 1.3.12 servers that are configured exactly the same will appear as identical targets to HMAP. Appendices A B contain respectively a more detailed discussion of HMAP design and output interpretation.

4.5 Assumptions/Limitations

HMAP was written in Perl [10][11][12] to make use of its pattern-matching facilities and portability. In its current incarnation it can parse the known profiles (about 20) and complete a fingerprint in well under a minute (several hundred Requests). For the number of profiles and tests used so far the interpreted nature of Perl has not been a performance bottleneck.

HMAP makes the reasonable assumption that the target server behaves deterministicly. Perhaps this assumption appears obvious, but one technique that will be suggested for frustrating identity probing is to randomize Responses to some extent to make it more difficult to compose a reliable description of the server. In general, this assumption has been reliable for all of the tested servers. HMAP also assumes that that the features it tests for are not easily configurable to other settings or if they are, it is rarely done. This assumption affects which tests can be reliably used. In practice only tests that produced consistent fingerprinting discriminators are retained in the testing set so the tests used are by definition the most reliable ones. If development of HMAP continues, a wider test bed of servers and configurations will undoubtedly result in changes in the list of tests used.

HMAP performs all tests it knows about even if all tests completed so far match only one server. In theory a decision tree technique might seem more efficient. This method has been avoided since the server is not known *a priori* and potential matches shouldn't be disregarded. In addition, one change to a server could easily frustrate a decision tree method. By testing all characteristics a more accurate picture develops.

It might be argued that the availability of a tool like HMAP does not impact web server security since even if a site is actively trying to hide it's identity an attacker can simply try all attacks they know of and observe which if any succeed. Note, however, that without a method similar to HMAP that the attacker is forced to increase both time and bandwidth used for an attack which generates more data for an Intrusion Detection System to use. On the other hand using an HMAP like method for gaining information about a system allows an attacker to narrow his attack space. While probing for system identity will also generate traffic, probing behaviors are frequently less likely to generate alarms and have a more ambiguous legal status than an actual attack.

Chapter 5

Security Implications

Web servers are vulnerable to a wide variety of attacks most of which are targeted by vendor and version number. A major contribution of this thesis is to underscore the difficulty that might be expected in thoroughly hiding a server's identity in an attempt to avoid such attacks. HMAP demonstrates this by implementing a methodology that automatically discovers a server's identity. In this section the security implications of this insight and the availability of HMAP-like tools will be discussed. At first glance HMAP may seem to be of primary benefit to attackers. However, attackers do not currently need this tool at all. At present very few sites hide their identity so this sort of tool will not be a critical member of a hacker's tool kit. It is invariably the case that the attackers usually have more information on vulnerabilities than the defenders, so making the information available is more likely to help more hurt.

5.1 Attackers

Assuming that an attacker is using this method of fingerprinting, what advantage does this give him? If an attacker didn't know what server was being used he would be forced to systematically try all of his attacks, many of which will be irrelevant to the target's specific vendor and version. This will force him to expend more time and bandwidth. By increasing confidence in the server's identity the attacker can winnow down to vulnerabilities that will more likely work for a target. Fewer tests will leave less of a footprint for security personnel to notice[1].

[1]As this thesis was being written a "worm" called Code Red was released. One of the reasons it was detected so quickly is that it "noisily" attacked randomly selected machines whether they used vulnerable HTTP servers or not.

If server logs and/or network traffic are being monitored for HTTP server fingerprinting activities, running the full suite of fingerprinting tests is fairly easy to detect. A stealthy hacker may try some of the following techniques to mask their activity:

- run subsets of tests from several computers and correlate data - an Intrusion Detection System (IDS) might not see a pattern of behavior if it analyzes on a host by host basis
- run tests over a long period of time (if the IDS notices one unusual Request then it might be just random error - earlier transactions will be flushed over time)
- only do short forms of large Request tests (e.g. first half of long URL list may provide enough information for discrimination)
- make a search tree of characteristics and only do the necessary ones. (find minimal subset of characteristics that identify the specific server)
- mask contents of Requests with URL encoding
- change the contents of long URL type Requests so they vary in more than just the length

To really develop HMAP into an attack tool known attacks could be programmed into it to take advantage of the exploit once the server type is known.

It must be remembered that ascertaining a server's identity this way is not necessarily against the law or even truly an attack. Even if a system administrator were to detect that someone was running HMAP against their site it is not clear what recourse they could take. Information gathering and probing is currently a gray area within the security community and likely to remain so for some time.

5.2 Defenders

Different aspects of computer security generally fit into one of the following categories: protection, detection and reaction. System Administrators and HTTP server developers work at the protection level, while Intrusion Detection falls in the detection level. Reaction is beyond the scope of this work. Defenders can improve their system security based on the fingerprinting methodology outlined above. The advice in each of these sections is complementary. The most successful protection will include elements from each category.

5.2.1 System Administrators

There are already many resources related to basic security configurations for web servers (e.g. [19]). Issues that arise in particular from an attacker using an HMAP-like technique to learn a server's identity are now discussed.

The most obvious use a sysadmin could have for the HMAP tool would be to use it against his own servers to determine if their web server is readily externally identifiable in its current state. In addition, HMAP could be used iteratively to determine if steps taken to obfuscate have had the intended effect. Examples of identity hiding steps are given in appendix C. Unfortunately the ability to hide one's identity will depend greatly on access to the source code and configuration options made available by the vendor.

It is very easy to fill up log files at an abnormal rate with HMAP. If there is no mechanism in place to control the lengths of the logs, then using long URL testing can quickly consume disk space which might be part of a denial of service attack. Even if there are such mechanisms, forcing large amounts of data through log files can obscure other attacks that might have occurred (needle in a hay stack). If the log files do roll over automatically then forcing large amounts of data through could flush out other data, perhaps hiding other signs of previous wrong doing.

Finally, since HMAP performs a large number of tests using non-standard Requests, it would be useful to run it against a server to see if it is susceptible to any simple denial of service attacks. At least one server was identified that crashes consistently when HMAP was run against it.

5.2.2 HTTP Server Developers

HTTP server developers can assist with hiding identity by providing options to make server fingerprinting more difficult. For instance, configuration parameters or even compilation settings for source code could be made available. Developers can use details of the server fingerprinting methodology to identify the types of characteristics and behaviors that need to be modifiable in order to hide a server's identity. The following are some strategies that could be employed to make identity hiding effective.

Common Interface and Behavior

The reason that fingerprinting works is that each vendor has a slightly different interpretation of how the server should appear and react. If the server development community could agree on a common interface, then fingerprinting would be a more difficult process. The HTTP specification is a step in this direction but to really succeed different fingerprinting cases would have to be specifically addressed to guarantee uniformity. This is unlikely considering the wildly varying compliance with the much more lax HTTP/1.X standards. As web server attacks increase in severity and frequency this sort of strategy may become more attractive.

Configuration Options

It is rare to find any server configuration options that allow identity hiding. Apache has a server directive “ServerTokens” that allows you to truncate the Server header’s level of detail but not to hide it entirely. It would be beneficial if users could remove this line. This will not be a fool proof hiding method but it is a step in the right direction. Even better would be the ability to address the many fingerprinting characteristics. By using configuration or build options the user should be able to have more control over the lexical, syntactic and semantic elements of Responses. Using these configuration options the users could masquerade as another server or act like no known server at all. In addition they could choose to match a generic standard as mentioned above.

Variable Output

To make things even more difficult for a fingerprinter, variable Responses could be allowed. For instance, “File Not Found”, “Not Found”, “Not found” could all be used randomly in appropriate Responses. Fingerprinting becomes more difficult if a static picture of the target can not be developed. This would not affect the ability of a client and server to converse since each of the variations would be semantically equivalent.

5.2.3 Intrusion Detectors - Misuse Detection

Since the problem of fingerprinting of web server identity is not yet well publicized, there are currently no IDSes that look for this exact behavior. IDSes that look for CGI attack probes do exist and presumably could be extended to include awareness of identity probing. Like server developers, IDS developers can use details of the fingerprinting methodology to identify characteristics and behavior that indicate identity probing is occurring.

As with many probe type activities (e.g. port scans) it is not necessarily the case that HMAP type behavior is a guarantee that illicit activity will follow, but it is certainly suspicious. Therefore an IDS wouldn't necessarily issue an alert upon detection of this behavior. Instead it should be thought of as providing sensors that show what is occurring. It will be the responsibility of a more comprehensive system to determine what to do with this information (e.g. decide that a probe might be part of a larger picture indicating attack preparations).

At a high level, an IDS's sensors can watch both the TCP traffic and the logfiles to acquire information about the system. Clues that fingerprinting is occurring can be found in both Requests and Responses. The suggestions below are geared towards misuse detection techniques but anomaly and specification detection are also possible. Many of the techniques used by HMAP in its current form are not very subtle and would be relatively easy to detect, but only if system administrators are aware of the goals of this type of unusual activity.

Request

Request Element Size Many of the tests used to provoke server Responses use very large elements. For instance large URIs and large numbers of headers are used to determine the Request sizes at which a server starts reporting certain errors. An IDS should look for these sorts of aberrations especially when the message changes in size over a wide range and contains headers / URLs that are not typical.

Unknown and Unusual Elements Unknown methods (e.g. "QWERTY") or methods that normal browsers rarely or never send (e.g. "TRACE") should be detected. The same observation applies to unknown or unusual header fields.

Unusual Constructions Most Requests have a fairly simple format as described above in the HTTP section. Unusual constructions, such as a Request including an inappropriate body or the use of incorrect line terminators should be examined.

Method Line Syntax Most browsers are fairly well behaved regarding how they issue a Request. Unusual spacing or corrupted version information is highly suspect.

Browser Behavior In the same way that a client can fingerprint a server, a server can check the structure and content of a client's Requests and determine if the client is correctly specifying it's own identity. If a client's User-Agent field indicates one type of browser but it behaves like another browser, this should raise suspicion. On the other hand, some privacy “sanitizing” packages [22] purposely hide the name of the browser so this method might raise too many false alarms. At this time few users seem to employ this sort of obfuscation so it should be a useful technique.

General IDS Eluding Techniques There are quite a few standard techniques that CGI vulnerability scanners use to thwart IDS detection. An important example of these is URL encoding which allows for a Request URL to be rewritten using a hexadecimal format making pattern matching more difficult and CPU intensive. A good list of these sorts of techniques can be found at the whisker site [9]. Other IDS eluding techniques include sending Requests from multiple servers and correlating the results later. Since IDS systems do not have infinite capacity for keeping state, allowing long time spans to pass between Requests can also be effective.

Response

Unusual and Repeated Errors Many of the tests are attempts to provoke non-“200 OK” Responses. While some of these errors like “404 File Not Found” are fairly common, others like “413 Request Entity Too Large” are rare enough to raise suspicion. Even common errors like “404 File Not Found” seen far out of proportion to their norm should also raise some flags.

Headerless Responses Since HTTP / 0.9 type clients are fairly rare these days, if a server sends back a Response that doesn’t have a header it is possible that someone is masquerading as such a client or a Request that has confused a server was sent.

Miscellaneous Techniques

Several other techniques can be used together with traditional IDS sensors to augment system security. An IDS could also help prevent the fingerprinting by detecting “bad” Requests (too long, malformed etc.) and converting them before they are received by the server so that the interrogator can not correlate Requests with Responses correctly. This conversion could be done by the IDS system itself or by some sort of proxy. A similar technique for preventing TCP / IP fingerprinting is described in [23]

Honeypots are another useful resource for tracking attacker behavior. With respect to web server fingerprinting a honeypot could detect a fingerprinting sort of behavior and then return purposely misleading Responses. Later if an attack is launched against the type of server that was falsely advertised, the deception could be continued, perhaps letting the attacker believe that their attack was successful. Only when there is a strong need to learn about an attacker should this method be employed.

As with all intrusion detection techniques additional resources (time, code complexity, memory) will be required to monitor, translate and analyze HTTP traffic. It is important to balance the cost of the security with the actual value of the resources being protected.

5.3 Miscellaneous Uses

As discussed earlier, Netcraft (as an example) performs world wide surveys of the web to determine statistics on the usage of various servers [14]. With respect to server identity they use the traditional simple HEAD method and check the Server line of the returned header. Most likely their current results are fairly accurate. If for security reasons the Server field is hidden then fingerprinting methods could allow them to continue to discover or verify their results. One method for reducing the intrusive nature of these tests is to use a strategic subset based only on trying to confirm the claim made about the server vendor.

Another more esoteric use relates to determining a “family tree” of sorts between servers. Some commercial servers are descendants of other server implementations. For instance Yahoo! is based on a version of Apache. Depending on the changes, it might be possible to determine from which version a server branched. This in turn might point to existing security problems in that product if they are also known to exist in its parent.

Chapter 6

Conclusion

Hiding the identity of an HTTP server from a sophisticated and knowledgeable attacker is a non-trivial endeavor. The content and organization of specifically provoked Responses carry enough information, in most cases, to uniquely fingerprint a web server's vendor and version information. A tool, HMAP, was developed that mimics the behavior of a client to elicit such Responses. These Responses are then automatically analyzed to create a fingerprint and identify a server with a high degree of confidence. Like many security utilities (e.g. SATAN) HMAP can be used by both attackers and defenders. Attackers can use increased confidence of a server's vendor and version information to help determine possible vulnerabilities to exploit. Alternatively, defenders can use information on how HTTP server identity probing works to help them strategically augment server security. Software developers can make HTTP servers more easily configured to leak less information about their identity. Server administrators can thoroughly test their servers to learn how easily its identity can be ascertained by probing. Intrusion Detection Systems can be augmented to detect HTTP server probing activities.

6.1 Future Work

There are a number of directions to be explored and functionality that could be added:

- A formal theory for protocol fingerprinting (ftp, smtp, telnet, ssl, etc.) should be developed. This could be used to analyze a protocol for its susceptibility to fingerprinting. A formal theory could build upon the lexical, syntactic, and semantic classification system used in this thesis.

- A generic tool/architecture for analyzing different protocols should be developed.
- Testing for proxies, gateways and other HTTP entities should be added. In addition, web site configurations such as virtual hosts should be accounted for.
- The battery of tests should be added to and refined. The most efficient discriminators should be determined and tests that give redundant information should be removed. It should be determined statistically from large survey how commonly certain characteristics are varied (e.g. is long error ranges stable in different Apache configurations?).
- The number of known server profiles should be increased. Tests using large test beds and wide arrays of servers/configurations should be employed.
- A number of useful related functions should be added to HMAP (in an NMAP sort of vein). For instance, install pages could be tested for existence. This might give clues as to how careful the system administration is attended to. Other functions could be added as learning tools. For instance, one could determine if a certain server is configured to respect a client's language preferences or the format of its cookies.
- Tests that check for specific configurations/add-on packages should be performed. Many vulnerabilities in web servers are really due to 3rd party packages. It would be useful to determine if these are being advertised.
- A traffic sniffing mode should be created that listens to HTTP traffic and makes a best guess off of normal traffic (100% stealthy).
- A better scoring/ranking method for matches should be devised. In particular it would be interesting to give different characteristics different weights depending on the ease of changing underlying server attributes (i.e. if a characteristic is easy to change in the server it is less valuable as a fingerprinting clue).

Appendix A

HMAP Internals

This section gives a brief overview of how HMAP works. The format of the initialization file, an examination of the different classes used and an explanation of the program flow are each discussed.

A.1 Server Profiles Input Format

HMAP starts by reading in a “server.profiles” file which contains the characteristics of all known servers. Here is a subsection of one of these files which shows a profile for one server:

```
SERVER_PROFILE_START
   SERVER_NAME=Apache/1.3.12 (Win32)
   MESSAGE_TEXTS
      HTTP_200=OK
      HTTP_400=Bad Request
      HTTP_403=Forbidden
      HTTP_404=Not Found
      HTTP_405=Method Not Allowed
      HTTP_406=Not Acceptable
      HTTP_412=Precondition Failed
      HTTP_413=Request Entity Too Large
      HTTP_414=Request-URI Too Large
      HTTP_501=Method Not Implemented
      HEADER_SERVER=^Apache/1\.3\.12 \(Win32\)
   ODD_REQUESTS
      BAD_REQUEST_RC_001=???
      BAD_REQUEST_RC_002=???
      BAD_REQUEST_RC_003=400
      BAD_REQUEST_RC_004=200
      BAD_REQUEST_RC_005=200
      BAD_REQUEST_RC_006=200
      BAD_REQUEST_RC_007=200
      BAD_REQUEST_RC_008=200
      BAD_REQUEST_RC_009=200
      BAD_REQUEST_RC_010=???
      BAD_REQUEST_RC_011=200
      BAD_REQUEST_RC_012=200
      BAD_REQUEST_RC_013=200
      BAD_REQUEST_RC_014=200
      BAD_REQUEST_RC_015=200
   ORDERING
      ORDER_OPTIONS=GET, HEAD, OPTIONS, TRACE;
      ORDER_HEADERS=Date, Server, Content-Location, Vary, TCN,
    Last-Modified, ETag, Accept-Ranges, Content-Length,
    Connection, Content-Type, Content-Language, Expires;
   FORMATTING
      FORMAT_ETAG=\"\w{1}\-\w{1,3}\-\w{1,8};\w{1,8}\"
   ERROR_CONDITIONS
      ERROR_LONG_URLS=216:404 Not Found,
                      217:403 Forbidden,
      8176:403 Forbidden,
      8177:414 Request-URI Too Large;
SERVER_PROFILE_END
```

This profile was generated automatically by HMAP while communicating with a known/trusted host. Each term that is followed by a "=" is a characteristic. The data for a characteristic is either a single item or a list of items. A list of items is terminated by a ";". Every item is treated as a string for simplicity. An individual server profile is initiated and terminated with SERVER_PROFILE_START and SERVER_PROFILE_END respectively. Any characteristic's name that is not recognized by HMAP is simply ignored so each of the "headings" above (e.g. MESSAGE_TEXTS) is only presented for ease of reading. Characteristic values that will be used for matching against a family of closely related values (e.g. FORMAT_ETAG) use Perl regular expression syntax.

A.2 Program Structure and Design

This section describes the different structural elements of HMAP (e.g. classes, program flow) and how they work together to generate a fingerprint.

A.2.1 HMAP

HMAP is the starting point for HMAP processing. HMAP is invoked by calling this file directly with a Perl interpreter. From this file the command line options are acted on and the "server.profiles" file is parsed. After this the real work is done by creating and using the various objects.

A.2.2 HMFingerprint

This class is the workhorse of HMAP. It keeps a list of all characteristics and the tests for eliciting each. It also keeps track of previous Responses (in some cases) and detects if the server Responses vary for the same characteristic. This could be an indication that more than one server exists for the same domain name. A fingerprint object interrogates the server for each characteristic. In the "getresult" method the fingerprint object uses the name of the test to decide which actual method will be called to generate an HTTP Request and analyze the reply. In order to populate as many of these characteristics as possible before issuing a report, the method "prefetch_characteristics" can be used to search for all characteristics. This is the typical mode of operation since some tests do not provoke the desired Responses from some servers but other tests do. HMFingerprint has the ability to print an easily readable output of all its values which can also be used directly in the "server.profiles" file.

A.2.3 HMProfile

HMProfile stores the characteristics for each server found in the "server.profiles" file. It is also used to perform the comparisons with each of the candidate characteristics whose values are stored in the HMFingerprint object. The specific type of comparisons is determined individually for each characteristic. For instance, it must know the difference between doing an exact string matching and the comparison of the order of elements in two lists.

A.2.4 HMUtils

A convenience package containing a few utilities such as list comparisons.

A.2.5 HMCookie

A simple wrapper for Cookie data.

A.2.6 HMResponse

HMResponse provide a wrapper for HTTP Responses. It knows how to parse a Response into its individual components: Message Line, Headers and Body. It can also report if a Response is malformed and give access to the various sub-components.

A.2.7 HMRequest

Just as HMResponse wraps an HTTP Response, and HMRequest wraps an HTTP Request. This class can generate a number of basic default Requests but also allows very low level manipulation and creation of Requests. Unlike most libraries available this class allows you to make malformed Requests.

A.2.8 HMHeader

A class that stores and allows access to groups of header lines. Most importantly it preserves the headers exactly as they were received and allows comparison of headers with each other. This class is used by both HMRequest and HMResponse.

A.2.9 HMUserAgent

This class's main responsibility is to create and manage the socket level connections and send and receive the Requests and Responses. The socket code is complicated by the fact that irregular Responses are frequently received and it is difficult to handle the returned data correctly for every type of server. Furthermore, many servers will end communications abruptly. This code tends to be somewhat fragile.

A.3 Program Flow

Once the basic function of each of the packages and classes is understood the execution of the program is relatively easy to follow. HMAP begins by reading the command line parameters to decide what its specific behavior will be. In the default mode it reads in the server profiles and creates a HMProfile object for each. Following this a HMFingerprint object is created. The HMFingerprint behaves somewhat like a HMProfile except that all of its characteristics are blank to begin with and it knows how to generate Requests that will fill in data for many of these characteristics. After the HMFingerprint object obtains as many characteristics as possible, comparisons are made between every characteristic it has with the corresponding characteristic in each profile. These comparisons continue until every profile and characteristic have been compared. Comparisons get one point for matching and zero for not matching. The scores are then sorted by the number of matches. These results are then displayed.

Appendix B

HMAP Session Example

The following demonstrates an example session with the HMAP tool.

```
C:\some\path\HMAP>hmap.pl xyzzy.cs.ucdavis.edu
uri is [xyzzy.cs.ucdavis.edu]

[miscellaneous debugging info .... ]

Known servers:
==============
1 : Apache/1.2.4
2 : Apache/1.3.3 (Unix)
3 : Apache/1.3.9 (Win32)
4 : Apache/1.3.9 (Unix)
5 : Apache/1.3.11 (Win32)
6 : Apache/1.3.12 (Win32)
7 : Apache/1.3.12 (Unix)
8 : Apache/1.3.14 (Win32)
9 : Apache/1.3.14 (Unix)
10 : Apache/1.3.19 (Unix)
11 : Apache/1.3.20 (Win32)
12 : Jigsaw/2.1.2
13 : Microsoft-IIS/5.0
14 : NCSA/1.3
15 : NCSA/1.5.2
```

```
16 : Netscape-Enterprise/2.0a
17 : Netscape-Enterprise/3.5.1
18 : Netscape-FastTrack/4.1
19 : thttpd/2.20b
20 : thttpd/2.21b 23apr2001


Matching server types:
   Apache/1.3.14 (Unix) -  46:  0:  9

Scoring: (# of matches:# of misses:# of not found)
   Apache/1.2.4:                 30: 15: 10
   Apache/1.3.11 (Win32):        42:  4:  9
   Apache/1.3.12 (Unix):         42:  6:  9
   Apache/1.3.12 (Win32):        42:  4:  9
   Apache/1.3.14 (Unix):         46:  0:  9
   Apache/1.3.14 (Win32):        42:  4:  9
   Apache/1.3.19 (Unix):         43:  3:  9
   Apache/1.3.20 (Win32):        42:  4:  9
   Apache/1.3.3 (Unix):          43:  3:  9
   Apache/1.3.9 (Unix):          24:  3: 28
   Apache/1.3.9 (Win32):         42:  4:  9
   Jigsaw/2.1.2:                 13: 30: 12
   Microsoft-IIS/5.0:            17: 28: 10
   NCSA/1.3:                     17:  7: 31
   NCSA/1.5.2:                   15:  9: 31
   Netscape-Enterprise/2.0a:     14: 28: 13
   Netscape-Enterprise/3.5.1:    12:  5: 38
   Netscape-FastTrack/4.1:       19: 25: 11
   thttpd/2.20b:                  9: 14: 32
   thttpd/2.21b 23apr2001:       16: 26: 13

C:\some\path\HMAP>
```

The first section (“Known servers”) lists all of the servers for which profiles exist. Next the best matches for the target server are shown. This section can have more than one best match. In this case the confidence is very high since there were no mismatches. Finally the full results are shown for all server profiles.

Another common use for HMAP is to fetch a page or issue a specific Request method against a server. The following shows the result of sending a “HEAD” method to a server:

```
D:\practice\perl\HMAP>hmap.pl -mHEAD www.slashdot.org
uri is [www.slashdot.com]

*** Sending request:
HEAD / HTTP/1.0


.
*** Received response:
HTTP/1.1 200 OK
Date: Fri, 27 Jul 2001 21:42:31 GMT
Server: Apache/1.3.12 (Unix) mod_perl/1.24
Connection: close
Content-Type: text/html




D:\practice\perl\HMAP>
```

Appendix C

HTTP Server Identity Obfuscation Example: Apache/1.3.12

The modifications required to systematically obfuscate the identity of an Apache / 1.3.12 server are described. These changes are made by a combination of configuration and code changes. Implementation of these changes exclusively by means of the configuration file would be ideal. The changes demonstrated here will obfuscate many of characteristics for which the current version of HMAP tests. The ability to hide the identity in the case of Apache is probably better than what could be achieved for other servers since the source code was available. In addition, more configurable elements and better granularity of control were available. For example, many of the elements of IIS/5.0 on Windows 2000 could not be obfuscated. The key to each of these changes is to alter the information returned in a response without changing its semantics. Most of these modifications can be used independently or in combination with others.

C.1 Configuration Changes

As mentioned it would be preferable if all of the changes could be done via the configuration file. Currently very few of Apache's ServerDirectives allow this sort of change. Some configuration options that have been found to help hide server identity are described.

C.1.1 Request Size Limits

The directives LimitRequestBody, LimitRequestFields, LimitRequestFieldsize and LimitRequestLine each give the system administrator the ability set upper limits on the size of Request elements. Several HMAP tests check for the ranges of different errors and at what size these errors are elicited. By changing these limits to values different from the defaults (which does not appear to be common) these errors will then occur at different ranges. It is not clear if the sequence of errors encountered will change however and another version of HMAP may look only for this sequence irrespective of what values they occur at.

C.1.2 Server Name Reduction

It would be preferable to have the option to completely remove the Server header from a request or to substitute it with a fictitious value. This is not possible without code modification. There are several ServerDirectives that approximate this ability.

```
ServerTokens Prod[uctOnly]
     Server sends (e.g.): Server: Apache
ServerTokens Min[imal]
     Server sends (e.g.): Server: Apache/1.3.0
ServerTokens OS
     Server sends (e.g.): Server: Apache/1.3.0 (Unix)
ServerTokens Full (or not specified)
     Server sends (e.g.): Server: Apache/1.3.0 (Unix) PHP/3.0
```

C.1.3 Header

The Header directive provides the ability to add, remove and append to header entries in a Response. Unfortunately these do not apply to the Server header. It still may be of value to add or remove headers. Experiments with adding and removing headers have been attempted. Unless the header is one that you have created yourself, they can have erratic results (appending instead of replacing, replacing in addition to appending, and so on). But this method is worth investigating. Adding headers will at least add some subterfuge.

C.2 Code Changes

The most effective way to hide a server's identity is by modification of the source code. We can use the categories: lexical, syntactic, and semantic to classify the code changes by their effect.

C.2.1 Lexical

We can change many of the words used in a response without affecting how it will be interpreted by the client.

Wording of Human Readable Messages

The file http_protocol.c contains an array of strings called status_lines [RESPONSE _CODES] that hold the human readable messages for each return code. Each of these can be changed synonymously to other phrases without changing the meaning of the response.

Capitalization of Header Fields

The header fields must retain the same wording so that the client will understand it, but most clients seem to understand variations in capitalization style. This technique will require some hunting around the code since the headers tend to be hardcoded in many different places through out. One good concentration of them is in http_protocol.c in the function ap_send_http_header(request_rec *r). Here one could change "Content-Type" to "Content-type" for instance.

Remove or Change Server Name

A simple technique that will frustrate the naive attacker is to remove or alter the server's name. There are several locations where this could be accomplished. The simplest is to simply use the SERVER_BASEVERSION key word in httpd.h and change this to the desired value (or simply leave as blank).

Date Format

There are several different date formats that are permitted by the HTTP/1.1 specification. The format of the date returned from the header could be modified to another of these legal formats. A convenient place to make this change would be in the function ap_gm_timestr_822(pool *p, time_t sec). It's possible that some clients may be confused by alternate formats for time since the common format (i.e. Date: Fri, 23 Feb 2001 22:06:03 GMT) is by far the most common.

C.2.2 Syntactic

Allow Header Options Order

In response to an OPTIONS method a server typically responds with a message that includes an "Allow" header listing the methods that can be used against the designated URI. Examining the function make_allow(request_rec *r) in http_protocol.c, the order that these methods are listed can be rearranged.

Header Order

Changing the order of headers can help obfuscate server identity since this order is often unique to a specific server. There are many different functions where headers are constructed, so one can identify these locations (such as (http_protocol.c: ap_basic_http _header(request_rec *r), ap_send_http_header(request_rec *r), ap_send_error_response(request _rec *r, int recursive_error), ap_send_http_options(request_rec *r), etc.) and make changes there. Alternatively, one could work with the routine where the headers are pooled together and ordered (alloc.c: ap_table_setn(table *t, const char *key, const char *val)).

ETags

Since the main function of the ETag is to provide a unique identifier of a document one could easily change the format and still provide distinct values. These changes should be made in ap_make_etag(request_rec *r, int force_weak)

C.2.3 Semantic

Finally one can change the server's request interpretation. This sort of change is the most complicated since more substantial changes than just textual substitution need to be made. The changes must preserve the semantics of "normal" request while at the same time changing these semantics to some degree.

Request Regularizer

Changing the semantics of Apache would require a significant amount of work and so is beyond the scope of this thesis. Nonetheless we can hypothesize the sort of translation function that would be needed to do this. A straight forward way to achieve this would be to create a “regularizing” function that pre-processes the request and alters the content of any unusual requests so they match a more generic format. For instance, long URIs could be truncated to some acceptable length before normal processing. Another example would be to check if the request line contains unknown methods or bad version numbers. A default “bad request” could be substituted for this before processing. In both of these scenarios a valid fingerprint could not be generated since the core Apache engine never directly interacts with the probing requests.

Bibliography

[1] T. Berners-Lee, R. Fielding, H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*, RFC 1945

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2068

[3] D. Kristol, L. Montulli. *HTTP State Management Mechanism*, RFC 2109

[4] Y. Goland, E. Whitehead, A. Faizi, S. Carter, D. Jensen. *HTTP Extensions for Distributed Authoring – WEBDAV*, RFC 2518

[5] Balachander Krishnamurthy, Martin Arlitt. *PRO-COW: Protocol Compliance on the Web—A Longitudinal Study*, 2001

[6] Balachander Krishnamurthy, Jeffrey C. Mogul, David M. Kristol. *Key Differences between HTTP/1.0 and HTTP/1.1*, 1999

[7] fyodor. *Remote OS detection via TCP/IP Stack FingerPrinting*, http://www.insecure.org/nmap/nmap-fingerprinting-article.html

[8] Rain Forest Puppy. *whisker*, http://www.wiretrip.net/rfp/bins/whisker/

[9] Rain Forest Puppy. *A look at whisker's anti-IDS tactics*, http://www.wiretrip.net/rfp/pages/whitepapers/whiskerids.html

[10] Larry Wall, Tom Christiansen, Randal L. Schwartz. *Programming Perl*, 1996

[11] Sriram Srinivasan. *Advanced Perl Programming*, 1997

[12] Joseph N. Hall, Randal L. Schwartz. *Effective Perl Programming*, 1998

[13] Various. *http://www.securityfocus.com*, web

[14] Netcraft Surveys. *http://www.netcraft.com*, web

[15] Security Space. *http://www.securityspace.com/s_survey/data/index.html*, web

[16] Gisle Aas *http://www.linpro.no/lwp/*, web

[17] Apache Group *http://www.apache.org*, web

[18] Apache Documentation *http://www.apache.org/docs-2.0/misc/known_client_problems.html*, web

[19] Security Tips for Server Configuration *http://httpd.apache.org/docs/misc/security_tips.html*, web

[20] 100hot Web Rankings *http://www.100hot.com/directory/100hot/*, web

[21] Netscape *http://home.netscape.com/download/index.html?cp=djudep2*, web

[22] Internet Junkbuster Proxy *http://www.junkbusters.com/ijb.html*, web

[23] Defeating TCP/IP Stack Fingerprinting *http://www.usenix.org/events/sec2000/smart.html*, web

[24] Multipurpose Internet Mail Extensions *http://www.hunnysoft.com/mime/rfc2045.txt*, web