Nessus Attack Scripting Language

From Wikipedia, the free encyclopedia

[Jump to navigation](https://en.wikipedia.org/wiki/Nessus_Attack_Scripting_Language#mw-head)[Jump to search](https://en.wikipedia.org/wiki/Nessus_Attack_Scripting_Language#searchInput)

The **Nessus Attack Scripting Language**, usually referred to as **NASL**, is a [scripting language](https://en.wikipedia.org/wiki/Scripting_language) that is used by [vulnerability scanners](https://en.wikipedia.org/wiki/Vulnerability_scanner) like [Nessus](https://en.wikipedia.org/wiki/Nessus_(software)) and [OpenVAS](https://en.wikipedia.org/wiki/OpenVAS). With NASL specific attacks can be automated, based on known [vulnerabilities](https://en.wikipedia.org/wiki/Vulnerability).

Tens of thousands of plugins have been written in NASL for Nessus and OpenVAS.[[1]](https://en.wikipedia.org/wiki/Nessus_Attack_Scripting_Language#cite_note-1) Files that are written in this language usually get the file extension .nasl. For the exploitation of a [zero day attack](https://en.wikipedia.org/wiki/Zero_day_attack) it is possible for an end user of Nessus or OpenVAS to write custom code in NASL which is executed by these vulnerability scanners.

In earlier versions of Nessus, a binary called nasl or nasl.exe was provided that could interpret NASL code to perform vulnerability scans. In later versions of Nessus, this should be done via an [API](https://en.wikipedia.org/wiki/API) that is provided by this software.

An example of executing a NASL plugin 'myzeroday.nasl' on Windows, a command such as the following could be invoked:

nasl.exe -t 127.0.0.1 "C:\temp\myzeroday.nasl"

An equivalent example of a Linux or UNIX command could look like this:

nasl -t 127.0.0.1 /tmp/myzeroday.nasl

If the plugin, in this example myzeroday.nasl, is placed in the same directory where other NASL plugins are located, it can also be included in standard scans by Nessus or OpenVAS, via the Web GUI or an API.

Many of the specifications of the formal language are similar to those of the programming language C and the scripting language Perl and those of other languages. [Control flow](https://en.wikipedia.org/wiki/Control_flow) such as the for loop, the if and if-else statements are part of the language and comments are preceded by a hash.

An example of "Hello World" in NASL is:[[2]](https://en.wikipedia.org/wiki/Nessus_Attack_Scripting_Language#cite_note-2)

display("Hello World\n");

In the release notes of Nessus 6.10.0 of 1/31/2017, a new NASL compiler for faster plugins was mentioned.[[3]](https://en.wikipedia.org/wiki/Nessus_Attack_Scripting_Language#cite_note-3)

References[[edit](https://en.wikipedia.org/w/index.php?title=Nessus_Attack_Scripting_Language&action=edit&section=1)]

* 1. [**^**](https://en.wikipedia.org/wiki/Nessus_Attack_Scripting_Language#cite_ref-1) [*"Fast comparison of Nessus and OpenVAS knowledge bases"*](https://avleonov.com/2016/11/27/fast-comparison-of-nessus-and-openvas-knowledge-bases/)*. 27 November 2016.*
  2. [**^**](https://en.wikipedia.org/wiki/Nessus_Attack_Scripting_Language#cite_ref-2) Network Security Tools, by Justin Clarke, Nitesh Dhanjani, O'Reilly Media, Inc., April 2005, [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [9780596007942](https://en.wikipedia.org/wiki/Special:BookSources/9780596007942)
  3. [**^**](https://en.wikipedia.org/wiki/Nessus_Attack_Scripting_Language#cite_ref-3) <https://docs.tenable.com/releasenotes/nessus/nessus50.htm> Nessus 6.10.0 Release Notes - 1/31/2017, New NASL compiler for faster plugins

Sources[[edit](https://en.wikipedia.org/w/index.php?title=Nessus_Attack_Scripting_Language&action=edit&section=2)]

* [Information on NASL](http://searchsecurity.techtarget.com/tip/Using-Nessus-Attack-Scripting-Language-NASL-to-find-application-vulnerabilities)
* Nessus Network Auditing, 2nd Edition by Russ Rogers, Publisher: Syngress Release Date: October 2011 [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [9780080558653](https://en.wikipedia.org/wiki/Special:BookSources/9780080558653) Chapter 11 NASL
* [Using the 'nasl' Nessus Command Line Tool](https://www.tenable.com/blog/using-the-nasl-nessus-command-line-tool)
* [NASL API Documentation](http://www.openvas.org/compendium/nasl-api-documentation.html)
* [Basic Structure of NASL Scripts](http://www.openvas.org/compendium/basic-structure-of-nasl-scripts.html)
* [Adding third party nasl plugins to OpenVAS by Alexander V. Leonov Information Security Automation](https://avleonov.com/2017/06/30/adding-third-party-nasl-plugins-to-openvas/)
* [FreeBSD Manual Pages, Manual Reference Pages - OPENVAS-NASL (1)](https://www.gsp.com/cgi-bin/man.cgi?section=1&topic=openvas-nasl)
* Nessus, Snort, and Ethereal Power Tools, by Noam Rathaus, Gilbert Ramirez, Jay Beale, Brian Caswell, Publisher: Syngress 2005 [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [9780080489421](https://en.wikipedia.org/wiki/Special:BookSources/9780080489421) Chapter 1. The Inner Workings of NASL (Nessus Attack Scripting Language)

**The Nessus Attack Scripting Language Reference Guide  
(incomplete)**

**Renaud Deraison < deraison@cvs.nessus.org >**

**Version 0.98.2**

**1  Introduction**

**1.1  What is NASL ?**

NASL is a scripting language designed for the Nessus security scanner. Its aim is to allow anyone to write a test for a given security hole in a few minutes, to allow people to share their tests without having to worry about their operating system, and to garantee everyone that a NASL script can not do anything nasty except performing a given security test against a given target.  
Thus, NASL allows you to easily forge IP packets, or to send regular packets. It provides you some convenient functions that will make the test of web and ftp server more easy to write. NASL garantees you that a NASL script :

* will not send any packet to a host other than the target host
* will not execute any commands on your local system

**1.2  What NASL is not ?**

NASL is not a powerful scripting language. Its purpose is to make scripts that are security tests. So, do not expect to write a third generation web server in this language, nor a file conversion utility. Use perl, python or whatever scripting language to do this.

NASL was designed rather quickly, so you may spot some inconstencies in its syntax. Please, let me know if you find some.

**1.3  Why not using Perl/Python/tcl/*whatever you like* for Nessus ?**

I know that there is a lot of very good scripting languages around here, and that NASL is really weak compared to them. But none of these languages is secure, in the sense that you can easily write a test that will be a trojan and will indeed open a connection to a third party host - letting it know that you are a Nessus user, and even eventually send the name of your targets to this evil third party host. Or worse, it could send your passwd file, or whatever.

Another problem with many of these scripting language : a lot of them are memory hungry. It can also be an headache if you want to configure them for Nessus. Just think about Perl. Perl is good. Perl is beautiful (according to some). But how much time will you have to spend to install all the modules that may be necessary for writing efficient Nessus tests ? Net::RawIP is only one of them.

NASL, on the other hand, does not take a huge amount of memory. This way, you can launch 20 threads of nessusd at the same time, without the need of having 256Mb of RAM. NASL is also self-sufficient. That is, you will not have to install a dozen of packages for each new security test.

**1.4  Why should you write your tests in NASL ?**

You may already wonder whether it is worth or not to learn yet another scripting language to write your tests, rather than coding them in C or Perl, or whatever. What you must know is that :

* NASL is optimized for Nessus. Writing a Nessus test in this language is fast
* NASL has a lot of things in common with C, so you should not be afraid of it
* NASL produces secure and easily sharable security tests
* NASL produces portable and easily modifiable security tests. When the Windows NT version of Nessus will be released, you will use the same functions to do the same things (such as sending raw IP packets)

**1.5  What this guide will teach you**

This guide teaches you how to write your own Nessus tests in NASL. This is my first attempt to write a comprehensive document, so I may have written complicated things.

**1.6  NASL limitations : what to not expect**

As I stated before, NASL is not a powerful language. The biggest limitations as of now are :

* *The lack of users defined functions*. This is planned for the 0.99.x version of the NASL package, but this is currently unsupported.
* *The poor string operations*. NASL does not support regular expressions, so you are somewhat limited in the way you can work with strings.

**1.7  Thanks**

I would like to thank the following persons for their advices regarding the design of NASL. Without them, NASL would be more akward than it is already :

* Denis DUCAMP (denis@hsc.fr)
* FYODOR (fyodor@dhp.com)
* Noam RATHAUS (noamr@securiteam.com)

I always appreciate suggestions and complaints about the language. Do not hesitate to share your opinion (be it good or bad) with me.

**2  The basics : NASL syntax**

NASL syntax is very similar to C, except that a lot of boring stuff has been removed. You do not have to care about the type of your objects, nor do you have to allow memory for them or free it. You do not need to declare your variables before you use them. You just have to focus on the security test you want to perform.

If you do not know C, then you will have a hard time reading this manual has it is currently intended for C programmers. Just complain and this guide will be made more readable in the future.

**2.1  Comments**

The comment char is '#'. It only comments out the current line.

Examples :

Valid comments are :

a = 1; # let a = 1

# Set b to 2 :

b = 2;

Invalid comments would be :

#

Set a to 1 :

#

a = 1;

a = # set a to 1 # 1;

**2.2  Variables, variables types, memory allocation, includes**

You do not need to declare variables before you use them. You do not have to care about the variable types. The NASL interpretor will yell at you if you try to do bogus things, such as adding an IP packet to a number. And you do not have to care about memory allocation nor do you have to care about the includes. There is no include. Memory is allocated when needed.

**2.3  Numbers and strings**

Numbers can be entered in three bases : decimal, hexadecimal, or binary.

All these lines are correct :

a = 1204;

b = 0x0A;

c = 0b001010110110;

d = 123 + 0xFF;

The strings must be quoted. Note that, unlike C, the characters are not interpolated unless you explicitly ask to interpolate them using the string() function.

a = "Hello\nI'm Renaud"; # a equals to "Hello\nI'm Renaud"

b = string("Hello\nI'm Renaud"); # b equals to "Hello

# I'm renaud"

c = string(a); # c equals to b

The string() function will be dealt with in the ``String Manipulation'' section.

**2.4  Anonymous / Non Anonymous arguments**

**2.4.1  Non Anonymous functions**

One thing which is different with C is the way NASL handles the arguments of a function. In C, you must know by heart which argument must be at which place. And this quickly becomes an headache when a function you call has more than 10 arguments. For instance, imagine a C function which will forge an IP packet for you. This function requires a dozen of arguments. If you want to use it, then you will have to remember their exact order or read the the documentation of this function. This is a waste of time, and this is what NASL attempts to avoid.

So, when the order of the arguments of a function is important, and when the different arguments of the function have different types, then the function is a non anonymous function. That is, you have to give the name of the elements. If you forget some elements, then you will be prompted for them at runtime.

* Example :

The function forge\_ip\_packet() has a lot of elements. These two calls are valid and perform the exact same thing :

forge\_ip\_packet(ip\_hl : 5, ip\_v : 4,

ip\_p : IPPROTO\_TCP);

forge\_ip\_packet(ip\_p : IPPROTO\_TCP,

ip\_v : 4, ip\_hl : 5);

The user will be prompted at runtime for the missing arguments (ip\_len, and so on...). Of course, a security test must not directly interact with the user, but this is handy for debugging and quick coding.

**2.4.2  Anonymous functions**

The anonymous functions are functions that take only one argument, or arguments of the same type.

Examples :

send\_packet(my\_packet);

send\_packet(packet1, packet2, packet3);

These functions may have options. For instance, the send\_packet() function waits for an answer. If you feel there is no need to read the host's answer, then you can deactivate the pcap, and speed up the test :

send\_packet(packet, use\_pcap:FALSE);

. The repeat operators

- For and while

The for and while work like in C :

for(instruction\_start;condition;end\_loop\_instruction)

{

#

# Some instructions here

#

}

Or

for(instruction\_start;condition;end\_loop\_instruction)function();

while(condition)

{

#

# Some instructions here

#

}

or

while(condition)function();

Examples :

# Count from 1 to 10

for(i=1;i<=10;i=i+1)display("i : ", i, "\n");

# Count from 1 to 9, and say the type

# of each number (even or odd)

for(j=1;j<10;j=j+1){

if(j & 1)display(j, " is odd\n");

else display(j, " is even\n");

}

# Do something completely useless :

i = 0;

while(i < 10)

{

i = i+1;

}

**2.5  Operators**

The standard C operators work in NASL. That is, +,-, \*, / and % work. At this time, the operators priority is not taken in account, but this will change. In addition to this operators, the binary operators | and & are implemented.

In addition to this, there are two operators that do not exist in C :

**2.5.1  The 'x' operator**

for and while are great and handy. But because the condition has to be evaluated at each iteration, then there is a loss of performance, which can be of some trouble if you want to send a SYN storm or whatever. The 'x' operator will repeat the same function N times, and will go really fast (at native C speed actually).

Example :

send\_packet(udp) x 10;

Will send the same udp packet ten times.

**2.5.2  The ' > < ' operator**

The >< operator is a boolean operator which returns true if a string of chars A is contained in a string B.

Example :

a = "Nessus";

b = "I use Nessus";

if(a >< b){

# This will be executed since

# a is in B

display(a, " is contained in ", b, "\n");

}

**3  The NASL Network related functions**

NASL will not let you open a socket to another host that the host that nessusd wants to test.

**3.1  Sockets manipulation**

A socket is a way to communicate with another host using TCP or UDP. It is like a pipe, designed to send data on a given port of a given protocol.

**3.1.1  How to open a socket**

The functions open\_sock\_tcp() and open\_sock\_udp() will open a TCP or UDP socket. These two functions are using anonymous arguments. You can currently open a socket on only one port at once, but this will eventually change in the future.  
Example :

# Open a socket on TCP port 80 :

soc1 = open\_sock\_tcp(80);

# Open a socket on UDP port 123 :

soc2 = open\_sock\_udp(123);

The open\_sock functions will return 0 if the connection could not be established on the remote host. Usually, open\_sock\_udp() will never fail, since there is no way to determine whether the remote UDP port is open or not, whereas the open\_sock\_tcp() function will return 0 if the remote port is closed.  
A trivial TCP port scanner would be like this :

start = prompt("First port to scan ? ");

end = prompt("Last port to scan ? ");

for(i=start;i<end;i=i+1)

{

soc = open\_sock\_tcp(i);

if(soc) {

display("Port ", i, " is open\n");

close(soc);

}

}

**3.1.2  Closing a socket**

The function close() is used to close a socket. It will internally perform a shutdown() before actually closing the socket.

**3.1.3  Writing to a socket, and reading from it**

Reading and writing to a socket is done using one of these functions :

* recv(socket:<socketname>,
* length:<length>
* [,timeout : <timeout>)

Reads <length> bytes from the socket <socketname>. This function can be used for TCP and UDP. The timeout option is in seconds.

* recv\_line(socket:<socketname>,
* length:<length>
* [, timeout: <timeout>])

This function works the same way as recv(), except that it will stop reading data as soon as the \n character is read. This function only works with TCP sockets.

* send(socket:<socket>, data:<data> [, length:<length>]) : send the data <data> on the socket <socket>. The optional argument length tells the function to only send <length> bytes on the socket. If it is not set, then the data will be sent until a NULL character is met.

The functions that are used to read data from a socket have an internal timeout value of five seconds. If the timeout is reached, then they will return FALSE.  
Example :

# This Example displays the FTP banner of the remote host :

soc = open\_sock\_tcp(21);

if(soc)

{

data = recv\_line(socket:soc, length:1024);

if(data)

{

display("The remote FTP banner is : \n", data, "\n");

}

else

{

display("The remote FTP server seems to be tcp-wrapped\n");

}

close(soc);

}

**3.1.4  Higher level operations**

NASL has a set of high level functions, regarding FTP and WWW.

* ftp\_log\_in(socket:<soc>, user:<login>, pass:<pass>) will attempt to log into the FTP server connected to the freshly open socket <soc>. This function returns TRUE if it was possible to log in as <login> with password <pass>. It returns FALSE if an error occured.
* ftp\_get\_pasv\_port(socket:<soc>) issues a PASV command on the FTP server, and returns the port to open a connection onto. This allows NASL scripts to retrieve data via FTP. This function returns FALSE if an error occurred.
* is\_cgi\_installed(<name>) returns TRUE if the cgi <name> is installed on the remote web server. This function performs a GET request on the remote web server. If <name> does not start by a slash (/), then /cgi-bin/ is appended in the front of it. This function can also be used to determine the existence of a given file.

Examples :

#

# WWW

#

if(is\_cgi\_installed("/robots.txt")){

display("The file /robots.txt is present\n");

}

if(is\_cgi\_installed("php.cgi")){

display("The CGI php.cgi is installed in /cgi-bin\n");

}

if(!is\_cgi\_installed("/php.cgi")){

display("There is no 'php.cgi' in the remote web root\n");

}

#

# FTP

#

# open a connection to the remote host

soc = open\_sock\_tcp(21);

# Log in as the anonymous user

if(ftp\_log\_in(socket:soc, user:"ftp", pass:"joe@"))

{

# Get a passive port

port = ftp\_get\_pasv\_port(socket:soc);

if(port)

{

soc2 = open\_sock\_tcp(port);

data = string("RETR /etc/passwd\r\n");

send(socket:soc, data:data);

password\_file = recv(socket:soc2, length:10000);

display(password\_file);

close(soc2);

}

close(soc);

}

**3.2  Raw packets manipulation**

NASL allows you to forge your own IP packets, and will attempt to behave in an intelligent way with the packet forged. For instance, if you change a parameter in a TCP packet, then the TCP checksum will be recomputed silently. If you append a layer to an IP packet, then the ip\_len element of the IP packet will be updated - unless you deliberately say to not do it.

All the raw packets functions use non-anonymous arguments. Their names comes straight from the BSD include files. So, the 'length' element of an ip packet is called ip\_len and not 'length'.

**3.2.1  Forging an IP packet**

The function forge\_ip\_packet() will forge a new IP packet. The function get\_ip\_element() will return an element of a packet, whereas the function set\_ip\_elements() will change the elements of an existing IP packet.

<return\_value> = forge\_ip\_packet(

ip\_hl : <ip\_hl>,

ip\_v : <ip\_v>,

ip\_tos : <ip\_tos>,

ip\_len : <ip\_len>,

ip\_id : <ip\_id>,

ip\_off : <ip\_off>,

ip\_ttl : <ip\_ttl>,

ip\_p : <ip\_p>,

ip\_src : <ip\_src>,

ip\_dst : <ip\_dst>,

[ip\_sum : <ip\_sum>] );

The ip\_sum argument of this function is optional. If it is not set, it will be automatically computed. The field ip\_p may be a numeric value, or one of the constants IPPROTO\_TCP, IPPROTO\_UDP, IPPROTO\_ICMP, IPPROTO\_IGMP or IPPROTO\_IP.

<element> = get\_ip\_element(

ip : <ip\_variable>,

element : "ip\_hl"|"ip\_v"|"ip\_tos"|"ip\_len"|

"ip\_id"|"ip\_off"|"ip\_ttl"|"ip\_p"|

"ip\_sum"|"ip\_src"|"ip\_dst");

The function get\_ip\_element() will return one element of a packet. The element must be one of "ip\_hl", "ip\_v", "ip\_tos", "ip\_len", "ip\_id", "ip\_off", "ip\_ttl", "ip\_p", "ip\_sum", "ip\_src" or "ip\_dst". Note that the quotes have their importance.

set\_ip\_elements( ip : <ip\_variable>,

[ip\_hl : <ip\_hl>, ]

[ip\_v : <ip\_v>, ]

[ip\_tos : <ip\_tos>,]

[ip\_len : <ip\_len>,]

[ip\_id : <ip\_id>, ]

[ip\_off : <ip\_off>,]

[ip\_ttl : <ip\_ttl>,]

[ip\_p : <ip\_p>, ]

[ip\_src : <ip\_src>,]

[ip\_dst : <ip\_dst>,]

[ip\_sum : <ip\_sum> ]

);

The function set\_ip\_elements() change the value of the IP packet <ip\_variable> and recomputes the checksum if the element ip\_sum is not altered. Since this function will not create a new packet in memory, you should prefer it to forge\_ip\_packet() when you have to send multiple, nearly similar, IP packets.

Last but not least, there is a function dump\_ip\_packet(<packet>) which will print the IP packet in human readable form on screen. You should only use this for debugging purpose.

**3.2.2  Forging a TCP packet**

The function forge\_tcp\_packet() is used to forge a TCP packet. Its syntax is :

tcppacket = forge\_tcp\_packet(ip : <ip\_packet>,

th\_sport : <source\_port>,

th\_dport : <destination\_port>,

th\_flags : <tcp\_flags>,

th\_seq : <sequence\_number>,

th\_ack : <acknowledgement\_number>,

[th\_x2 : <unused>],

th\_off : <offset>,

th\_win : <window>,

th\_urp : <urgent\_pointer>,

[th\_sum : <checkum>],

[data : <data>]);

The option th\_flags must be one of TH\_SYN, TH\_ACK, TH\_FIN, TH\_PUSH or TH\_RST. Flags can be combined using the | operator. th\_flags may also be a numeric value. ip\_packet must have been generated with forge\_ip\_packet() or must have be a packet read using send\_packet() or pcap\_next().

The function used to change TCP elements is set\_tcp\_elements(). It's syntax is similar to forge\_tcp\_packet() :

set\_tcp\_elements(tcp : <tcp\_packet>,

[th\_sport : <source\_port>,]

[th\_dport : <destination\_port>,]

[th\_flags : <tcp\_flags>,]

[th\_seq : <sequence\_number>,]

[th\_ack : <acknowledgement\_number>,]

[th\_x2 : <unused>,]

[th\_off : <offset>,]

[th\_win : <window>,]

[th\_urp : <urgent\_pointer>,]

[th\_sum : <checkum>],

[data : <data>] );

This function will automatically recompute the checksum of the packet, unless you explicitly set the th\_sum element.

The function used to get one element of a TCP packet is get\_tcp\_element(). Its syntax is :

element = get\_tcp\_elements(tcp: <tcp\_packet>,

element: <element\_name>);

element\_name must be one of "tcp\_sport", ""th\_dport", "th\_flags", "th\_seq", "th\_ack", "th\_x2", "th\_off", "th\_win", "th\_urp", "th\_sum". Note the quotes !

**3.2.3  Forging a UDP packet**

The UDP functions are nearly the same as for TCP functions :

udp = forge\_udp\_packet(ip:<ip\_packet>,

uh\_sport : <source\_port>,

uh\_dport : <destination\_port>,

uh\_ulen : <length>,

[uh\_sum : <checksum>],

[data : <data>]);

The functions set\_udp\_elements() and get\_udp\_elements() work the same way as for the TCP functions.

**3.2.4  Forging an ICMP packet**

**3.2.5  Forging an IGMP packet**

**3.2.6  Sending a raw packet**

Once you have set up a packet using forge\_\*\_packet(), you can send it using the send\_packet() function.

This function syntax is :

reply = send\_packet(packet1, packet2, ...., packetN,

pcap\_active: <TRUE|FALSE>,

pcap\_filter: <pcap\_filter>);

If the argument pcap\_active is set to TRUE (the default), then this function will wait for a reply from the host the packet was sent to. You can set up the argument pcap\_filter to define what kind of packet you want. See the pcap (or tcpdump) manual to learn more from pcap filters.

**3.2.7  Reading raw packets**

You can read a packet using the pcap\_next() function, the syntax of which is :

reply = pcap\_next();

This function will read a packet from the last interface you used, with the last pcap filter you used on this interface.

**3.3  Utilities**

NASL provides several handy functions that usually makes your coding easier.

* The function this\_host() takes no argument an returns the IP address of the host the script is running on.
* The function get\_host\_name() takes no argument and returns the name of the currently tested host.
* The function get\_host\_ip() takes no argument and returns the IP adress of the currently tested host.
* The function get\_host\_open\_port() takes no argument and returns the number of the first open TCP port of the remote host. This is useful for some scripts such as land or a TCP sequence analyzing program which need to work against an open port.
* The function get\_port\_state(<portnum>) returns TRUE if the TCP port <portnum> is open, or if its state is unknown (for instance, if it was not scanned, or if it is outside the scanned range).
* The function telnet\_init(<soc>) initialize a telnet session on the freshly opened socket <soc> and returns the first line of telnet data.

Example :

soc = open\_sock\_tcp(23);

buffer = telnet\_init(soc);

display("The remote telnet banner is : ", buffer, "\n");

* The function tcp\_ping() takes no argument and returns TRUE if the remote host answered to a TCP ping request (send a TCP packet with the ACK flag set).
* The function getrpcport() is the same as the standard function of the same name. Its syntax is :
* result = getrpcport(program : <program\_number>,
* protocol: IPPROTO\_TCP|IPPROTO\_UDP,
* [version: <version>]);

This function returns 0 if an error occured (if the program <program\_number> is not registered in the remote rpc portmapper for instance).

**4  String manipulation functions**

NASL handles strings as numbers. So, you can play with the ==, <, and > operators safely.

Example :

a = "version 1.2.3";

b = "version 1.4.1";

if(a < b){

#

# Will be executed, since version 1.2.3 is lower

# than version 1.4.1

}

c = "version 1.2.3";

if(a==c) {

# Will also be evaluated

}

It is also possible to get the n-th character of a string, the same way as in C :

a = "test";

b = a[1]; # b equals to "e"

You can also add and substract strings :

a = "version 1.2.3";

b = a - "version "; # b equals "1.2.3"

a = "this is a test";

b = " is a ";

c = a - b; # c equals to "this test"

a = "test";

a = a+a; # a equals to "testtest"

In addition to this and to the >< operator defined above, NASL has a set of functions dedicated to forge or modify strings :

**4.1  The crap() function**

The function crap() is very convenient to test for buffer overflows. It has two syntaxes :

* crap(<length>) : Will return a string of length <length> containing the character 'X'
* crap(length:<length>, data:<data>) : Will return a string of length <length>, containing the data <data>

Example :

a = crap(5); # a = "XXXXX";

b = crap(4096); # b = "XXXX...XXXX" (4096 X's)

c = crap(length:12, # c = "hellohellohe" (length: 12);

data:"hello");

**4.2  The string() function**

This function is used to make strings of chars or of other strings. It syntax is : string(<string1>, [<string2>, ..., <stringN>])

This function will interpolate the blackslashed characters such as \n or \t.

Example :

name = "Renaud";

a = string("Hello, I am ", name, "\n"); # a equals to "Hello, I am Renaud"

# (with a new line at the end)

b = string(1, " and ", 2, " makes ", 1+2); # b equals to "1 and 2 makes 3"

c = string("MKD ", crap(4096), "\r\n"); # c equals to "MKD XXXXX.....XXXX"

# (4096 X's) followed by a carriage

# return and a new line

**4.3  The raw\_string() function**

Example :

a = raw\_string(80, 81, 82); # a equals to 'PQR'

**4.4  The tolower() function**

This function is used to convert a string to lower case. Its syntax is tolower(<string>). This function will actually return the string <string> in lowered letters.

Example :

a = "Hello";

b = tolower(a); # b equals to "hello"

**5  Writing a Nessus Security test**

**5.1  How to write an efficient Nessus test**

All the security test are launched by nessusd, in a very short period of time, so a well written test must use the results of the other security test. For instance, a test which wants to open a connection to a FTP server should first check that the remote port is open, before opening a connection on port 21. This saves little time and bandwidth against a given host, but this dramatically speeds up the test against a firewalled host which would silently drop TCP packets going to port 21.

**5.1.1  Determining whether a port is open**

The function get\_port\_state(<portnum>) returns TRUE if the port is open, and FALSE if it is not. *This function will return true if the port has not been scanned, that is, if its status is unknown*.

This function uses very little CPU, so you should call it as much as you want.

**5.1.2  The Knowledge Base (KB)**

Each host is associated to an internal knowledge base, which contains all the information gathered by the tests during the scan. The security tests are encouraged to read it and to contribute to it. The status of the ports, for instance, is in fact written somewhere in the knowledge base.

The KB is divided into categories. The ``Services'' category contains the port numbers associated to each known service. For instance, the element Services/smtp is very likely to have the value 25. However, if the remote host has a hidden SMTP server on port 2500, and none on port 25, then this item will have the value 2500.

See Annex B for details about the knowledge base elements.

Basically, there are two functions regarding the knowledge base. The get\_kb\_item(<name>) function will return the value of the knowledge base item <name>. This function is anonymous. The function set\_kb\_item(name:<name>, value:<value>) will mark the new item <name> of value <value> in the knowledge base.

**Note :**You can not read back an knowledge base item you have added. For instance, the following piece of code will not work and never execute what it should :

set\_kb\_item(name:"attack", value:TRUE);

if(get\_kb\_item("attack"))

{

# Perform the attack - will not be executed

# because our local KB has not been updated

}

This is due to the fact that for some security and code stability reason, the Nessus server will in fact start each new security test with a copy of the knowledge base, not the original one, and the function set\_kb\_item() will in fact add an element into the orginal knowledge base, within nessusd, but will not update the current security test knowledge base.

**5.2  NASL script structure**

Each NASL script must register itself to the Nessus server. That is, it must tell nessusd its name, its description, the name of its author, and more. Thus, each NASL script that will be run with nessusd must have the following structure :

#

# Nasl script to be used with nessusd

#

if(description)

{

# register information here...

#

# I will call this section the 'register'

# section

#

exit(0);

}

#

# Script code here. I will call this section the

# 'attack' section.

#

The variable description is a global variable that will be set to TRUE or FALSE depending on whether the script must register or not.

**5.2.1  The register section**

The *register* section **must** call the following functions :

* script\_name(language1:<name>, [...]) which sets the script name as it will appear in the Nessus client window.
* script\_description(language1:<desc>, [...]) which sets the script description as it will appear in the client when the user clicks on the name.
* script\_summary(language1:<summary>, [...]) sets the script summary as it appears in the tooltips. It must be a sum up of the description that fits on one line.
* script\_category(<category>) sets the script category. It must be one of ACT\_ATTACK, ACT\_GATHER\_INFO, ACT\_DENIAL or ACT\_SCANNER.
  + ACT\_GATHER\_INFO : the script will be launched among the first. You know it will not harm the remote computer.
  + ACT\_ATTACK : the script will attempt to gain some priviledges on the remote host. It may harm the remote system (if it tests a buffer overflow for instance)
  + ACT\_DENIAL : the script will attempt to crash the remote host
  + ACT\_SCANNER : the script is a port scanner
* script\_copyright(language1:<copyright>, [...]) sets the copyright of the script. It may be your name, a legal notice or whatever.
* script\_family(language1:<family>, [...]) sets the script family. There are no clearly defined families, so you may choose to register the script in the family ``Joe's PowerTools'', altough I do not recommand it. The currently used families are :
  + Backdoors
  + CGI abuses
  + Denial of Service
  + FTP
  + Finger abuses
  + Firewalls
  + Gain a shell remotely
  + Gain root remotely
  + Misc.
  + NIS
  + RPC
  + Remote file access
  + SMTP problems
  + Useless services

As you may have noticed, most of these functions take a language1 argument. In fact, this is not how they work.

NASL provides Nessus multilingual support. Each script must support the english language, and the exact syntax for all these functions is in fact :

script\_function(english:english\_text, [francais:french\_text,

german:german\_text,

...]);

In addition to these functions, the function script\_dependencie() may be called. It tells nessusd to launch the current script after some other script. This is useful when you want to use the results that another script must store in the KB. The syntax is :

script\_dependencie(filename1 [,filename2, ..., filenameN]);

where filename is the name of the script to be launched after, as it is stored on disk.

**5.2.2  The attack section**

The attack section may contain anything you think is useful for an attack. Once your attack is done, you can report a problem using the security\_warning() and security\_hole() functions which work the same way. security\_warning() must be used when the attack was a success but is not a great security problem. That is, it will not allow instant access to an attacker. These two functions have the following syntaxes :

security\_warning(<port> [, protocol:<proto>]);

security\_hole(<port> [, protocol:<proto>]);

security\_warning(port:<port>, data:<data> [, protocol:<proto>]);

security\_hole(port:<port>, data:<data> [, protocol:<proto>]);

In the first case, the data displayed by the client is the script description, as entered with script\_description(). It is handy, because of the multilingual support.

In the second case, then the client will display the data argument. This is handy if you must display information caught on the fly, such as a version number.

**5.2.3  An example**

See /usr/local/lib/nessus/plugins/ for script examples

**5.3  Your script behavior**

If you plan to share your script then you should obey to these rules :

* *Your script must never interact with the user*. NASL scripts are executed on the server side. Therefore, all the output will not be seen by the user.
* *Your script must test one vulnerability*. If you know how to test multiple vulnerabilities, then write several scripts. So that you stay consistent with all the Nessus scripts
* *Your script should belong to an existing family*. If you plan to share your script, then avoid to create a family like *Joe's Power Tools* but try to stay consistent
* *Send it to the Nessus maintainer*. That is, me :) If you plan to share your script, then make it available to everyone, not only your friends or a newsgroup you hang on. Send it and see it being included in the Nessus distribution

**6  Conclusion**

I hope that you have not been too disappointed with NASL. It is a handy language, but things are still missing. As you could see, it does not currently support user-defined functions, nor does it support structures. (These two items will be done, but some missing features will not). In spite of this, please try to consider it if you want to write Nessus tests.

You will see bugs in the NASL interpretor. That is for sure. I do not know how you program, so it is very likely that you will manage to make it crash. Please, do not keep the bugs for you. Share them, and send them to me.

I hope you enjoyed reading this guide.

-- Renaud Deraison

<deraison@cvs.nessus.org>