The NASL2 reference manual

Michel Arboi <mikhail@nessus.org>

$Date: 2005/04/29 08:50:59 $

Abstract

This is the NASL2 reference manual ($Revision: 1.65 $). It describes the language syntax and the internal functions.

If you want tips on how to write a security test in NASL, read *The Nessus Attack Scripting Language Reference Guide* by Renaud Deraison <deraison@nessus.org>.

Contents

1 Introduction

1.1 History

Please read *The Nessus Attack Scripting Language Reference Guide*.
Here is what the man page says:

> NASL comes from a private project called "pkt_forge", which was written in late 1998 by Renaud Deraison and which was an interactive shell to forge and send raw IP packets (this pre-dates Perl's Net::RawIP by a couple of weeks). It was then extended to do a wide range of net work-related operations and integrated into Nessus as "NASL".
>
> The parser was completely hand-written and a pain to work with. In Mid-2002, Michel Arboi wrote a bison parser for NASL, and he and Renaud Deraison re-wrote NASL from scratch. Although the "new" NASL was nearly working as early as August 2002, Michel's lazyness made us wait for early 2003 to have it working completely.

1.2 Differences between NASL1 and NASL2

- NASL2 uses a real Bison parser. It is stricter and can handle complex expressions.
- NASL2 has more built-in functions (although most of them could be back ported to NASL1).
- NASL2 has more built-in operators.
- NASL2 is much quicker (about sixteen times).
- Most NASL2 scripts cannot run under NASL1.
- And a few NASL1 scripts cannot run under NASL2 (but fixing them is easy).
- NASL2 user-defined functions can handle arrays.

1.3 Copyright

1.4 Comments

Please send comments to Michel Arboi <mikhail@nessus.org>.
I checked the spelling of this document with an American dictionary, however the grammar may be incorrect.

2 The NASL2 grammar

2.1 Preliminary remarks

- A comment starts with a **#** and finishes at the end of the current line. It is ignored by the lexical analyzer.
- You may insert "blanks" anywhere between two lexical tokens.
  A blank may be a sequence of white space, horizontal or vertical tabulation, line feed, form feed or carriage return characters; or a comment.
- Token are parsed by a lexical analyzer and returned to the parser.
  - As the lexical analyzer returns the longer token it finds, expressions like `a+++++b` without any white space are erroneous because they will be interpreted as `a++ ++ + b` i.e. `(a++ ++) + b` just like in ANSI C[1]. You have to insert spaces: `a++ + ++b`
  - You cannot insert spaces in the middle of multiple character tokens. e.g. `x = a + +;` will not parse. Write `x = a ++;`

2.2 Syntax

decl_list	instr_decl instr_decl instr_decl_list
instr_decl	instr func_decl;
func_decl	**function** identifier **(** arg_decl **)** block
arg_decl	/*nothing*/ arg_decl_1
arg_decl_1	identifier identifier **,** arg_decl_1
block	**{** instr_list **}** **{ }**
instr_list	instr instr instr_list
instr	s_instr **;** block if_block loop
s_instr	aff post_pre_incr rep func_call ret

[1] They used to work in K&R C.

	inc loc glob **break** **continue** /*nothing*/
ret	**return** expr **return**
if_block	**if (** expr **)** instr **if (** expr **)** instr **else** instr
loop	for_loop while_loop repeat_loop foreach_loop
for_loop	**for (** aff_func **;** expr **;** aff_func **)** instr
while_loop	**while (** expr **)** instr
repeat_loop	**repeat** instr **until** expr **;**
foreach_loop	**foreach** identifier **(** array **)** instr
array	expr
aff_func	aff post_pre_incr func_call /*nothing */
rep	func_call **x** expr
string	STRING1 STRING2
inc	**include (** string **)**
func_call	identifier **(** arg_list **)**
arg_list	arg_list_1 /*nothing*/
arg_list_1	arg arg **,** arg_list_1
arg	expr identifier **:** expr
aff	lvalue = expr lvalue **+=** expr lvalue **-=** expr lvalue ***=** expr lvalue /= expr

lvalue **%=** expr
lvalue > >= expr
lvalue > > >= expr
lvalue < <= expr

lvalue identifier
array_elem

identifier IDENTIFIER
x

array_elem identifier **[** array_index **]**

array_index expr

post_pre_incr ++ lvalue
– lvalue
lvalue **++**
lvalue **–**

expr **(** expr **)**
logic_expr
arith_expr
bit_expr
post_pre_incr
compar
INTEGER
STRING2
STRING1
var
aff
cst_array
ipaddr

logic_expr expr **and** expr
! expr
expr **or** expr

arith_expr expr **+** expr
expr **-** expr
- expr
expr ***** expr
expr / expr
expr **%** expr
expr ****** expr

bit_expr **~** expr
expr **&** expr
expr ^ expr
expr | expr
expr > > expr
expr > > > expr
expr < <expr

compar	expr >< expr expr >**!**< expr expr =~ string expr **!**~ string expr < expr expr > expr expr == expr expr **!**= expr expr >= expr expr <= expr
var	identifier num_arg array_elem func_call
ipaddr	INTEGER **.** INTEGER **.** INTEGER **.** INTEGER
num_arg	$INTEGER $*
cst_array	**[** l_array **]**
l_array	array_data array_data **,** l_array
array_data	atom string => atom
atom	integer string
loc	**local_var** arg_decl
glob	**global_var** arg_decl
INTEGER	is any sequence of decimal digit (preceded by an optional minus sign), or **0** followed by a sequence of octal digits, or **0x** followed by a sequence of hexadecimal digits.
IDENTIFIER	is any sequence of letters (uppercase or lowercase) or digits, starting with a letter. The underscore sign is treated as a letter. Note that "**x**" is not exactly an identifier because it is the "repeat" operator, but can be used for function or variables names.
STRING1	is a string between simple quotes.
STRING2	is a string between double quotes.

2.3 Types

NASL2 handles the following data types:

1. **integers**
   Any sequence of digits with an optional minus sign is an integer. NASL2 uses the C syntax: octal numbers can be enter by starting with **0** and hexadecimal with **0x** (i.e. 0x10 = 020 = 16)

2. **strings**, which can exist in two flavors: “pure” and “impure”[2].

   (a) “Impure” strings are entered between double quotes and are not converted: backslashes remain backslashes. “Impure” strings are transformed into “pure” string by the internal **string** function.

   (b) “Pure” strings are returned by **string** or are entered between simple quotes. In this case, a few escape sequences are transformed[3].

3. **arrays**, which can be indexed with integers[4] or strings[5].

4. And the **NULL** value, which is what you get if you read an initialized variable, or what internal functions returns in case of severe error.
   Read the warning below!

5. **Booleans** are not a standalone type. The comparison operators return **0** for FALSE and **1** for TRUE. Any other value is converted :

   - The undefined or null value is FALSE.
   - Integers are TRUE if not null; **0** is FALSE.
   - Strings are TRUE if not empty and not **"0"**. This is the same behaviour as Perl or NASL1.
     WARNING! Previous versions of this manual were wrong and said that **"0"** was TRUE. We might switch to this more consistent semantics. To be sure of the results, it is better to test “strlen(s) > 0” if non empty string should be TRUE, or “int(s)” if the string should be interpreted as an integer.
   - Although it does not really make sense, arrays are always TRUE, whether they are empty or not.

All built-in or user-defined functions can handle or return all those types (even arrays!).

Warnings about the NULL value

[2]This is an heritage from NASL1, it would have been too complex to break it. The **string** function interprets escape sequences in “impure” strings and returns a “pure” string; it just copy “pure” strings without changing them. Note that **display** calls **string** before printing its argument on the standard output.

[3]Much less than in C, but I don’t think we need the octal representation, wide chars, etc. Note that the parser did not accept **\0** in older NASL2 versions; and **\x00** truncated the string before the nul character. This has been fixed.

So... **\n** is the newline character, **\t** the horizontal tabulation, **\v** the vertical tabulation, **\r** line feed, **\f** form feed, **\’** the single quote, **\ ”** the double quote (just in case), and **\x42** is “B”, because its ASCII code is 0x42 (66 in hex).

[4]Elements are numbered from 0, just like in C. Negative indexes are not supported (yet) and big values are not recommended as they woul eat memory. If you want such indexes, you should convert them into strings, so that they get hashed. I admit that this is neither clean nor efficient.

[5]Like the Perl hashes. Hashes have a big inconvenient: they destroy the order of the data they store.

NULL and the array operator Reading an array element from a NULL value will immediately convert it into an array. An empty array of course, but no more an undefined variable. Changing this means big modifications in the NASL interpreter. For example:

```
v = NULL;
# isnull(v)=TRUE and typeof(v)="undef"
x = v[2];
# isnull(x)=TRUE and typeof(x)="undef"
# But isnull(v)=FALSE and typeof(v)="array"
```

NULL and isnull If you want to check if a variable is undefined, you have to used **isnull(var)**. Testing the equality with the **NULL** constant (**var == NULL**) is not a good idea, as **NULL** will be converted to **0** or the empty string **“”** according to the type of the variable. This is necessary to ensure that variables are “automatically initialized” - changing this would probably break some existing scripts.

2.4 Operators

2.4.1 General operators

- **=** is the assignment operator.
  - `x=42;` puts 42 into the variable **x**. The previous value is forgotten.
  - `x=y;` copies the value of variable **y** into x. If **y** was undefined, **x** becomes undefined too.
- `[]` is the array index operator.
  - A variable cannot be atomic[6] and an array at the same time. If you changed the type, the previous value(s) is (are) lost.
  - However, this operator can be used to extract a character from a string: if **s** = "**abcde**", then **s[2]** = "**c**".
    In NASL1, this could be used to *change* the character too: you could write `s[2] = "C";` and **s** became "**abCde**". This is no longer true; you have to use the **insstr** function and write something like `s = insstr(s, "C", 2, 2);`
    See **insstr** on page 26.
  - `y[1] = 42;` makes an array out of **y** and puts 42 in the second element. If **y** was not an array, it’s first undefined.

2.4.2 Arithmetics operators

Be aware that there is no strict rule on the integer size in NASL2. The interpretor implements them with the native “int” C type, which is 32 bit long on most systems, and maybe 64 bit long on a few one[7]. There is no overflow or underflow protection.

[6] i.e. a “string” or an “integer”, or even “null”.
[7] Yes, no more 16 bit systems! Who wants to port NASL2 to MS/DOS?

- `+` is the addition operator.
- `-` is the subtraction operator.
- `*` is the multiplication operator.
- `/` is the integer division operator. Please note that:
  - NASL2 does not support floating point operations.
  - Division by zero will return 0 instead of crashing the interpretor. How nice of us!
- `%` is the modulo. Once again, if the 2nd operand is null, the interpretor will return 0 instead of crashing on SIGFPE.
- `**` is the exponentiation or power function[8].

2.4.3 Nice C operators

NASL2 imported some nice operators from C:

- `++` is the pre-incrementation (++x) or post-incrementation (x++).
  `++x` adds 1 to **x** and returns the result; `x++` adds 1 to x but returns the previous value.
- `--` is the pre-decrementation (–x) or post-decrementation (x–).
- **`+= -= *= /= %=`** have the same meaning as in C
  e.g. `x += y;` is equivalent to x = x + y;but **x** is evaluated only once. This is important in expressions like `a[i++] *= 2;` where the index "i" is incremented only once.
- **`<<=`** and **`>>=`** also exist; we added **`>>>=`**

2.4.4 String operators

- `+` is the string concatenation. However, you should better use the **string** function.
- `-` is the "string subtraction". It removes the first instance of a string inside another.
  For example **'abcd' - 'bc'** will give **'ad'**.
- `[]` extracts one character from the string, as explained before.
- **`><`** is the "string match" operator. It looks for substrings inside a string.
  `'ab' >< 'xabcdz'` is TRUE; `'ab' >< 'xxx'` is FALSE.
- **`>!<`** is the "string don't match" operator. It looks for substrings inside a string and returns the opposite as the previous operator.
  `'ab' >!< 'xabcdz'` is FALSE; `'ab' >!< 'xxx'` is TRUE.

[8] **`**`** is Fortran syntax. Maybe some of you will regret the Basic syntax, but `^` is already used by the exclusive-or (xor) operator (C syntax).

- **=~** is the "regex match" operator. It is similar to a call to the internal function **ereg** but is quicker because the regular expression is compiled only once when the script is parsed
  `s =~ "[ab]*x+"` is equivalent to `ereg(string:s, pattern:"[ab]*x+", icase:1)`
- **!~** is the "regex don't match" operator. It gives the opposite result of the previous one[9].

2.4.5 Compare operators

- **==** is TRUE if both arguments are equals, FALSE otherwise.
- **!=** is TRUE if both arguments are different, TRUE otherwise.
- **>** is the "greater than" operator.
- **>=** is the "greater than or equal" operator.
- **<** is the "lesser than" operator.
- **<=** is the "lesser than or equal" operator.

2.4.6 Logical operators

- ! is the logical "not". TRUE if its argument is FALSE, FALSE otherwise.
- && is the logical "and". Note that if the first argument is FALSE, the second is not evaluated.
- || is the logical "or". If the first argument is TRUE, the second is not evaluated.

2.4.7 Bit fields operators

- ~ is the arithmetic "not", the 1-complement
- & is the arithmetic "and".
- | is the arithmetic "or".
- ^ is the arithmetic "xor" (exclusive or).
- **<<** is the logical bit shift to the left.
- **>>** is the arithmetic / signed shift to the right[10].
- **>>>** is the logical / unsigned shift to the right[11].

In all shift operators, the count is on the right. i.e. `x>>2` is equivalent to `x/4` and `x<<2` is `x*4`

[9] In fact, there is a pathological case where both operator returns **NULL**: when the pattern could not be compiled. You will get an error when the pattern is parsed, then every time you try to execute the line.

[10] The sign bit, if any, is propagated.

[11] The sign bit is pushed to the right and replaced with zero.

2.4.8 Special behavior

- **break** can (but should not) be used to exit from a function or the script.
- In case its arguments have different types, + now tries very hard to do something smart, i.e. a string concatenation, then an integer addition. It prints a warning, though, because such automatic conversion is dangerous.
  - If one of its argument is undefined, + returns the other one.
  - If one of its argument is a "pure string", the other argument is converted to a string, if necessary, and the result is a "pure string". "Impure string" are converted to pure string *without escape sequence interpretation.* i.e. **"AB\n"+'de'** gives **'AB\\nde'**, i.e. "AB", a backslash, then "nde".
  - If one of its argument is an "impure string", the second argument is converted to string if necessary and the result is an "impure string". i.e. **"ABC"+2** gives **"ABC2".**
  - If one of its argument is an integer, the other is converted to integer and the result is an integer.
  - In any other case, NULL is returned.
- The "magical strings" from NASL1 have been removed. In NASL1, adding a string to an integer might give an integer if the string contained only digits.
- The minus operator follows the same type conversion rules as plus.
- Using unitialized variables is **bad**. However, to ensure that old scripts still work, the **NULL** undefined value will be into **0** or **""** according to the context (integer or string). That's why you have to use **isnull** to test if a variable is undefined. See "warnings about the NULL value" in 2.3.

2.5 Precedence

From the higher priority to the lower:

Operators	Associativity
++ - -	None
* *	Right
~ -(unary minus)	Left
!	Left
* / %	Left
+ -	Left
<< >> >>>	Left
&	Left
^	Left
|	Left
< <= > >= == != < > =~ !~ >!< ><	None
&&	Left
||	Left
= += -= *= /= %= <<= >>= >>>=	Right

2.6 Loops and control flow

2.6.1 Operators

- `for (expr1; cond; expr2) block;` is similar to the C operator and is equivalent to
  `expr1; while(cond) block; expr2;`
  A classical construction to count from 1 to 10 is:
  `for(i=1;i<=10;i++) display(i,'{\n');`

- `foreach var (array) block;` iterates all elements in an array. Note that *var* iterates through the *values* stored in the array, not the *indexes*. If you want that, just use: `foreach var (keys(array)) block;`

- `while(cond) block;` executes the block as long as the condition is TRUE. If the condition is FALSE, the block is never executed.

- `repeat block; until (cond);` executes the blocks as long as the condition is TRUE. The block is executed at least once.

- `break` breaks the current loop and jumps at its exit.
  If you are not inside a loop, the behavior is undefined[12].

- `continue`[13] jumps to the next step of the loop.
  If you are not inside a loop, the behavior is undefined.

- **`return`** returns a value from the current function.

2.6.2 Special behavior

2.7 Declarations

2.7.1 Variable declarations

NASL1 had only global variables. NASL2 uses global and local variables. Local variables are created in a function and stop existing as soon as the function returns. When the interpretor looks for a variable, it first searches in the current function context, then in the calling context (if any), etc., until it reaches the top level context that contains the global variables.

Normally, you do not need to declare a variable: either it exists, because you already used it in this context, or because a calling function used it, or it will be created in the current context. However, this may be dangerous in some cases:

1. if you want to write into a *global* variable from within a function and cannot be sure that the variable was created first in the top level context, or created as a local variable in a calling function context.

2. if you want to be sure that you are creating a brand new *local* variable and not overwriting a global variable with the same name.

So you can explicitly declare a variable:

[12] Currently, it exits from the current function or the script. But you should not rely upon this behavior.
[13] WARNING! This operator was introduced in Nessus 2.1.x; Nessus 2.0.x. cannot parse the script.

- `local_var var;`
- `global_var var;`

If the variable already exists in the specified context, you will get an error message, but this will work!

2.7.2 Function declarations

- **`function` name (argname1, argname2) block;**

Note that the argument list may be empty, but if it is not, user-defined function parameters must be named[14]. Unnamed arguments may be used without being declared.

2.7.3 Retrieving function arguments

Inside a NASL function, named arguments are just accessed as any local variable. Unnamed arguments are implemented through the special array **_FCT_ANON_ARGS**[15]. This variable will be **NULL** in interpretors below **NASL_LEVEL** 2190. You may put this at the start of scripts that need this function:

```
if (NASL_LEVEL < 2190) exit(0); # _FCT_ANON_ARGS is not implemented
```

1. Writing to _FCT_ANON_ARGS is undefined. Currently, the memory is wasted but the value cannot be read back.
2. Using _FCT_ANON_ARGS to try to read named arguments is bad too. Currently, there is a protection and a **NULL** value is returned.

2.7.4 Calling functions

Here is an example with named arguments:

```
function fact(x)
{
  local_var i, f;
  f = 1;
  for (i = 1; i <= n; i ++) f *= i;
  return f;
}
display("3 ! = ", fact(x: 3), "\n");
```

And the same with unnamed arguments:

[14] Unnamed arguments were introduced in NASL2.1.

[15] Shell-like special variables **$1**, **$2**... or the **$*** array were introduced in **NASL_LEVEL** 2160, but they broke the compatibility with older interpretors: the scripts could not be parsed. So those “dollar arguments” were removed in **NASL_LEVEL** 2190, because **_FCT_ANON_ARGS** was a more flexible solution. Actually, the special array **__FCT_ANON_ARGS** (with two leading unscores!) was introduced in 2180 level, but it was subtly flawed. It was renamed when the bug was fixed so that nobody uses it.

```
function fact()
{
  local_var i, f;
  f = 1;
  for (i = 1; i <= _FCT_ANON_ARGS[0]; i ++) f *= i;
  return f;
}
display("3 ! = ", fact(3), "\n");
```

And another, mixing the two flavours:

```
function fact(prompt)
{
  local_var i, f;
  f = 1;
  for (i = 1; i <= _FCT_ANON_ARGS[0]; i ++)
  {
    f *= i;
    display(prompt, i, '! = ', f, '\n');
  }
  return f;
}
n = fact(3, prompt: '> ');
```

3 The NASL2 library

3.1 Predefined constants

These constants are actually variables, i.e. you can modify their value in a script. If you really want to shoot you in the foot, that is...

- Booleans constants
  - **FALSE** = 0
  - **TRUE** = 1
- Plugin categories
  - **ACT_INIT**: the plugin just sets a few KB items (kinds of global variables for all plugins).
  - **ACT_SCANNER**: the plugin is a port scanner or something like it (e.g. ping).
  - **ACT_SETTINGS**: just like ACT_INIT, but run after the scanners, once we are sure that the host is alive (for performance).
  - **ACT_GATHER_INFO**: the plugin identifies services, gather data, parses banners, etc.
  - **ACT_ATTACK**: the plugin launches a soft attack, e.g. a web directory traversal.
  - **ACT_MIXED_ATTACK**: the plugin launches an attach that might have dangerous side effects (crashing the service most of the time).
  - **ACT_DESTRUCTIVE_ATTACK**: the plugin tries to destroy data[16] or launch some dangerous attack (e.g. testing a buffer overflow is likely to crash a vulnerable service).
  - **ACT_DENIAL**: the plugin tries to crash a service.
  - **ACT_KILL_HOST**: the plugin tries to crash the target host or disable it (e.g. saturate the CPU, kill some vital service...).
  - **ACT_FLOOD**: the plugin tries to crash the target host or disable it by flooding it with incorrect packets or requests. It may saturate the network or kill some routing, switching or filtering device on the way.
- Network constants
  - Nessus "encapsulation"
    - **ENCAPS_IP** = 1; this is the "transport" value for a pure TCP socket.
    - **ENCAPS_SSLv23** = 2; this is the "transport" value for a SSL connection in compatibility mode. Note that the **find_service** plugin will never declare a port with this "encapsulation", but you may use it in a script.
    - **ENCAPS_SSLv2** = 3. The old SSL version which only supports server side certificates.

[16] By the way, there is only *one* plugin that really tries to destroy data. This is *http_methods.nasl*

    - **ENCAPS_SSLv3** = 4. The new SSL version: it supports server and client side certificates, more ciphers, and fixes a few security holes.
    - **ENCAPS_TLSv1** = 5; TLSv1 is defined RFC 2246. Some people call it “SSL v3.1”.
  - Sockets options
    - **MSG_OOB**, a socket option used to send “out of band data”.
  - Raw sockets
    - **IPPROTO_ICMP** as defined in the system C include files.
    - **IPPROTO_IGMP**
    - **IPPROTO_IP**
    - **IPPROTO_TCP**
    - **IPPROTO_UDP**
    - **pcap_timeout** = 5
    - **TH_ACK** = 0x10. This TCP flag indicates that the packet contains a valid acknowledgment.
    - **TH_FIN** = 0x01. This TCP flag indicates that the packet negociates the end of the session.
    - **TH_PUSH** = 0x08.
    - **TH_RST** = 0x04. This TCP flag indicates that the connection was refused or “reset by peer”.
    - **TH_SYN** = 0x02. This belong to the initial handshake (connection opening).
    - **TH_URG** = 0x20. This TCP flag indicates that the packet contains urgent data.

- Miscellaneous constants
  - **NULL** is the undefined value.

- Nessusd glue
  - **description** is set to **1** when **nessusd** parses the script the first time (to get its name, description, summary, etc.), then to **0** when it is run.
  - **COMMAND_LINE** is set to **0** when the script is run by **nessusd** or to **1** when it is run by the **nasl** standalone interpretor.

3.2 Built-in functions

Internal built-in functions can have unnamed and named arguments. Some use both types.

3.2.1 Knowledge base functions

This KB is used for inter-plugin communication.

- **set_kb_item** creates a new entry in the KB.
  It takes two named string arguments: **name** and **value**.
  Entering an item several times creates a list.

- **get_kb_item** retrieves an entry from the KB.
  It takes one unnamed string argument (the **name** of the KB item).
  If the item is a list, the plugin will fork and each child process will use a different value. Nessus remebers which child got which value: reading the same item a second time will not fork again!
  You should not call this function when some connections are open if you do not want to see several processes fighting to read or write on the same socket.

- **get_kb_list** retrieves multiple entrie from the KB. It takes one unnamed string argument which may either designate a literal KB entry name, or a mask. The returned value is a "hash", i.e. an array with potentially duplicated indexes; because of this, you need to convert it with **make_list**() or use **foreach** to access each element (the **make_array** function allows you to create such hashes).

  ```
  # Retrieves the list of all the web servers
  webservers = get_kb_list("Services/www");
  # Retrieves the list of all the services
  services = get_kb_list("Services/*");
  # Retrieves the whole KB
  services = get_kb_list("*");
  ```

- **replace_kb_item** adds a new entry in the KB or replace the old value.
  It takes two named string arguments: **name** and **value**.
  Entering an item several times does not create a list, it just overwrites the old value.
  As this function is not defined in all Nessus version, it is safer to check that it is defined before calling it or use the **replace_or_set_kb_item** NASL function.

3.2.2 Report functions

Those functions send back information to the Nessus daemon.

- **scanner_status** reports the port scan progress (if the plugin is a port scanner!).
  It takes two named integer arguments:

  - **current**, the number of ports already scanned,
  - **total**, the full number of ports to be scanned.

- **security_note** reports a miscellaneous information.
  It either takes an unnamed integer argument (the port number), or a some of those named arguments:

  - **data** is the text report (the "description" by default).
  - **port** is the TCP or UDP port number of the service (or nothing if the bug concerns the whole machine, e.g. the IP stack configuration).
  - **proto** (or **protocol**) is the protocol (**"tcp"** by default; **"udp"** is the other value).

- **security_hole** reports a severe flaw.
  It either takes an unnamed integer argument (the port number), or a some of those named arguments:

  - **data** is the text report (the “description” by default).
  - **port** is the TCP or UDP port number of the vulnerable service (or nothing if the bug concerns the whole machine, e.g. the IP stack configuration).
  - **proto** (or **protocol**) is the protocol (**"tcp"** by default; **"udp"** is the other value).

- **security_warning** reports a mild flaw.
  It either takes an unnamed integer argument (the port number), or a some of those named arguments:

  - **data** is the text report (the “description” by default).
  - **port** is the TCP or UDP port number of the vulnerable service (or nothing if the bug concerns the whole machine, e.g. the IP stack configuration).
  - **proto** (or **protocol**) is the protocol (**"tcp"** by default; **"udp"** is the other value).

3.2.3 Description functions

All those functions but **script_get_preference** are only used in the “description part” of the plugin, i.e. the block that is run when the **description** variable is **1**. They only make sense in the Nessus environment and have no effect when the plugin is run with the standalone **nasl** interpretor.

- **script_add_preference** adds an option to the plugin.
  It takes tree named arguments:

  - **name** is the option name. As it is displayed “as is” in the GUI, it usually ends with “:”.
  - **type** is the option type. It may be:
    - **checkbox**
    - **entry**
    - **password**
    - **radio**
  - **value** is the default value (“yes” or “no” for checkboxes, a text string for “entries” or “passwords”) except for “radios”, where it is the list of options (separate the items with “;”). e.g.
    script_add_preference(name:"Reverse traversal", type:"radio", value:"none;Basic;Long URL");

- **script_bugtraq_id** sets the SecurityFocus “bid”.
  It takes one or several unnamed integer arguments.

- **script_category** sets the “category” of the plugin.
  Usually, its unnamed integer argument is one of those pre-defined constants[17] explained on page 15:

[17] Using an integer is definitely not a good idea, as new values may be inserted before the one you used. Actually, those values are not constants but initialized variables; changing their values in your script is a good way to shoot you in the foot.

  - **ACT_INIT**
  - **ACT_SCANNER**
  - **ACT_SETTINGS**
  - **ACT_GATHER_INFO**
  - **ACT_ATTACK**
  - **ACT_MIXED_ATTACK**
  - **ACT_DESTRUCTIVE_ATTACK**
  - **ACT_DENIAL**
  - **ACT_KILL_HOST**

- **script_copyright** sets the copyright string of the plugin (usually the author's name).
  It takes an unnamed string argument, or one or several named[18] arguments: **english**, **francais**, **deutsch**, **portuguese**.

- **script_cve_id** sets the CVE IDs of the flaws tested by the script.
  It takes any number of unnamed string arguments. They usually looks like "CVE-2002-042" or "CAN-2003-666".

- **script_dependencie** is the same function as **script_dependencies** (too many typos?).

- **script_dependencies** sets the lists of scripts that should be run before this one (if "optimize mode" is on).
  It takes any number of unnamed string arguments.

- **script_description** sets the "description" of the plugin.
  It takes an unnamed string argument, or one or several named arguments: **english**, **francais**, **deutsch**, **portuguese**. If the argument is unnamed, the default language is **english**.

- **script_exclude_keys** sets the list of "KB items" that must *not* be set to run this script in "optimize mode".
  It takes any number of unnamed string arguments.

- **script_family** sets the "family" of the plugin.
  It takes an unnamed string argument, or one or several named arguments: **english**, **francais**, **deutsch**, **portuguese**. If the argument is unnamed, the default language is **english**.
  There is no standardized family, but you should avoid inventing too many new ones. Here is a list:

[18] If you want to use a full sentence like "this plugin was written by Foo Bar" which would be translated in French, "ce plugin a été écrit par Foo Bar".

english	francais
Backdoors	Backdoors
Brute force attacks	
CGI abuses	Abus de CGI
CGI abuses: XSS	
CISCO	CISCO
Denial of Service	Déni de service
Finger abuses	Abus de finger
Firewalls	Firewalls
FTP	FTP
Gain a shell remotely	Obtenir un shell à distance
Gain root remotely	Passer root à distance
General	General
Misc.	Divers
Netware	
NIS	
Ports scanners	Port scanners
Remote file access	Accès aux fichiers distants
RPC	RPC
Settings	Configuration
Service detection	
SMTP problems	Problèmes SMTP
SNMP	SNMP
Useless services	Services inutiles
Windows	Windows
Windows : User management	
AIX Local Security Checks	
Debian Local Security Checks	
Fedora Local Security ChecksCGI	
FreeBSD Local Security Checks	
Gentoo Local Security Checks	
MacOS X Local Security Checks	
Mandrake Local Security Checks	
Red Hat Local Security Checks	
Solaris Local Security Checks	
SuSE Local Security Checks	

- **script_get_preference** reads an option. It takes an unnamed string argument. Note that it might returns an empty string if you are running the script from the standalone NASL interpretor.

- **script_get_preference_file_content** reads an “file” option. It takes an unnamed string argument. It retuns the content of the file, which is transmited from the Nessus client to the server.
  Note: **script_get_preference_file_content** and **script_get_preference_file_location** are restricted to “trusted” plugins.

- **script_get_preference_file_location** reads an option. It takes an unnamed string argument. It only makes sense if the preference type is “file”; it retuns the path

of the local copy of the file. **script_get_preference** would return the path of the file on the client machine, which is not useful.

- **script_id** sets the script number[19]. It takes an unnamed integer argument.
- **script_name** sets the "name" of the plugin.
  It takes an unnamed string argument, or one or several named arguments: **english**, **francais**, **deutsch**, **portuguese**. If the argument is unnamed, the default language is **english**.
- **script_require_keys** sets the list of "KB items" that must be set to run this script in "optimize mode".
  It takes any number of unnamed string arguments.
- **script_require_ports** sets the list of TCP ports that must be open to run this script in "optimize mode".
  It takes any number of unnamed integer or string arguments. e.g. **23** or **"Services/telnet"**.
- **script_require_udp_ports** sets the list of UDP ports that must be open to run this script in "optimize mode".
  It takes any number of unnamed integer arguments[20].
- **script_summary** sets the "short description" of the plugin.
  It takes an unnamed string argument, or one or several named arguments: **english**, **francais**, **deutsch**, **portuguese**. If the argument is unnamed, the default language is **english**.
  Each of its arguments should be a single line of text.
- **script_timeout** sets the default timeout of the plugin.
  It takes an unnamed integer argument. If it is **0** or **(-1)**, the timeout is infinite.
- **script_version** sets the "version" of the plugin.
  It takes an unnamed string argument[21].

3.2.4 Other "glue" functions

- **get_preference** takes an unnamed string argument and returns the "preference" value. This function is necessary to retrieve some server options. For example:

```
p = get_preference('port_range');    # returns something like 1-65535
```

[19] Which should you use? Well, there is only one rule: two scripts must have two different IDs. Ff your script is integrated into the Nessus distribution, the maintainer will choose an unaffected number.

[20] **find_service.nes** identifies TCP services and has no equivalent for UDP. So do not expect something like "Services/DNS" to returns a value different from 53. Unless you installed **amap** from www.thc.org `http://www.thc.org/` and run the UDP service identification.

[21] Usually, it is set to "$Revision" which is updated by CVS

3.2.5 Network functions

Note: the "socket" data type used by those functions is in fact an integer. However, you should not touch it and it may be turned into an opaque data type some day. In case of error, all those functions returns a value that can be interpreted as FALSE (most of the time NULL).

- **close** closes the socket given in its only unnamed argument.
- **end_denial** takes no argument and returns TRUE if the target host is still alive and FALSE if it is dead. You must have called **start_denial** before your test.
- **ftp_get_pasv_port** sends the "**PASV**" command on the open socket, parses the returned data and returns the chosen "passive" port.
  It takes one named argument: **socket**.
- **get_host_name** takes no argument and returns the target host name.
- **get_host_ip** takes no arguments and returns the target IP address.
- **get_host_open_port** takes no argument and returns an open TCP port on the target host.
  This function is used by tests that need to speak to the TCP/IP stack but not to a specific service.
- **get_port_transport** takes an unnamed integer (socket) argument and returns its "encapsulation" (see page 23).
- **get_port_state** takes an unnamed integer (TCP port number) and returns TRUE if it is open and FALSE otherwise.
  As some TCP ports may be in an unknown state because they were not scanned, the behavior of this function may be modified by the "consider unscanned ports as closed" global option. When this option is reset (the default), **get_port_state** will return TRUE on unknown ports; when it is set, **get_port_state** will return FALSE.
- **get_source_port** takes an unnamed integer (opn TCP socket) and returns the source port (i.e. on the Nessus server side).
- **get_tcp_port_state** is a synonym for **get_port_state**.
- **get_udp_port_state** returns TRUE if the UDP port is open, FALSE otherwise (see **get_port_state** for comments). Note that UDP port scanning may be unreliable.
- **islocalhost** takes no argument and returns TRUE if the target host is the same as the attacking host, FALSE otherwise.
- **islocalnet** takes no argument and returns TRUE if the target host is on the same network as the attacking host, FALSE otherwise.
- **join_multicast_group** takes an string argument (an IP multicast address) and returns TRUE if it could join the multicast group. If the group was already joined, the function joins increments an internal counter.

- **leave_multicast_group** takes an string argument (an IP multicast address).
  Note that if **join_multicast_group** was called several times, each call to **leave_multicast_cast** only decrements a counter; the group is left when it reaches 0.

- **open_priv_sock_tcp** opens a "privileged" TCP socket to the target host.
  It takes two named integer arguments:

  - **dport** is the destination port,
  - **sport** is the source port, which may be inferior to 1024.

- **open_priv_sock_udp** opens a "privileged" UDP socket to the target host.
  It takes two named integer arguments:

  - **dport** is the destination port,
  - **sport** is the source port, which may be inferior to 1024.

- **open_sock_tcp** opens a TCP socket to the target host[22].
  It takes an unnamed integer argument (the port number) and two optional named integer arguments:

  - **bufsz**, if you want to bufferize IO (this is disabled by default).
    This parameter has been added after Nessus 2.0.10.
  - **timeout**, if you want to change it from the default,
  - **transport**, to force Nessus a specific "transport". Its main use is to disable Nessus "auto SSL discovery" feature on dynamic ports (e.g. FTP data connections).
    The possible values for **transport** were explained in § 3.1 on page 15. They are:

    * **ENCAPS_IP**
    * **ENCAPS_SSLv23**
    * **ENCAPS_SSLv2**
    * **ENCAPS_SSLv3**
    * **ENCAPS_TLSv1**

- **open_sock_udp** opens a UDP socket to the target host.
  It takes an unnamed integer argument, the port number.

- **recv** receives data from a TCP or UDP socket.
  For a UDP socket, if it cannot read data, NASL will suppose that the last sent datagram was lost and will sent it again a couple of time.
  It takes at least two named arguments:

  - **socket** which was returned by **open_sock_tcp**, for example,
  - and **length**, the number of bytes that you want to read at most.
    recv may return before **length** bytes have been read: as soon as at least one byte has been received, the timeout is lowered to 1 second. If no data is

[22] In NASL, there is no way you can open connections to some specific host. This way, a NASL script cannot be trojaned.

received during that time, the function returns the already read data; otherwise, if the full initial timeout has not been reached, a 1 second timeout is re-armed and the script tries to receive more data from the socket. This special feature was implemented to get a good compromise between reliability and speed when Nessus talks to unknown or complex protocols. Two other optional named integer arguments can twist this behavior:

  - **min** is the minimum number of data that must be read in case the "magic read function" is activated and the timeout is lowered. By default this is **0**.
  - **timeout** can be changed from the default.

- **recv_line** receives data from **socket** and stops as soon as a *line feed* character has been read, **length** bytes have been read or the default timeout has been triggered.

- **send** sends data on a socket.
  Its named arguments are:

  - **socket**,
  - **data**, the data block. A string is expected here (pure or impure, this does not matter).
  - **length** is optional and will be the full **data** length if not set,
  - **option** is the flags for the send() system call. You should not use a raw numeric value here; the only interesting constant is **MSG_OOB**. See § 3.1 on page 16.

- **scanner_add_port** declares an open port to nessusd.
  It takes two named arguments and returns no value:

  - **port** is the port number,
  - **proto** is "**tcp**" or "**udp**".

- **scanner_get_port** walks through the list of open ports. It takes one unnamed integer argument, an index, and returns a port number or **0** when the end of the list if reached. A good way to use it is:

```
i = 0;
while (port = scanner_get_port(i++))
{
  do_something_with_port;
}
```

- **tcp_ping** launches a "TCP ping" against the target host, i.e. tries to open a TCP connection and sees if anything comes back (SYNACK or RST). The named integer argument **port** is not compulsory: if it is not set, **tcp_ping** will use an internal list of common ports[23].

- **telnet_init** performs a telnet negotiation on an open socket [RFC 854 / STD 8]. This function takes one unnamed argument (the open socket) and returns the data read (more or less the telnet dialog plus the banner).

[23] 22 (SSH), 25 (SMTP), 53 (DNS), 110 (POP3), 113 (IDENT), 443 (HTTPS), 993 (IMAPS), 8080 (alt HTTP), 65534.

- **this_host** takes no argument and returns the IP address of the current (attacking) machine.
- **this_host_name** takes no argument and returns the host name of the current (attacking) machine.
- **ftp_log_in** performs a FTP identification / authentication on an open socket. It returns TRUE if it could login successfully, FALSE otherwise (e.g. wrong password, or any network problem). It takes three named arguments:
  - **user** is the user name (it has *no* default value like "anonymous" or "ftp"),
  - **pass** is the password (again, no default value like the user e-mail address),
  - and **socket**.
- **start_denial** initializes some internal data structure for **end_denial**. It takes no argument and returns no value.

3.2.6 String manipulation functions

- **chomp** takes an unnamed string argument and removes any spaces at the end of it. "Space" means white space, vertical or horizontal tabulation, carriage return or line feed.
- **crap** returns a buffer of required length. This function is mainly used in buffer overflow tests. Its arguments are:
  - **length**, the size of the wanted buffer,
  - **data**, the pattern that will be repeated to fill the buffer. By default **'X'**.
- **display** takes an unlimited number of arguments, calls **string** on them, then displays them.
  It returns the number of output characters.
  Unprintable characters are replaced with "**.**".
- **egrep** looks for a pattern in a string, line by line and returns the concatenation of all lines that match. Its arguments are:
  - **icase**,
  - **pattern**,
  - **string**.
- **ereg** matches a string against a regular expression. It returns the first found pattern. Its arguments are:
  - **string**,
  - **multiline**, which is FALSE by default (string is truncated at the first "end of line"), and can be set to TRUE for multiline search.
  - **pattern** (standard extended POSIX regex, no PCRE for the moment!),
  - and **icase**, which is FALSE by default, and can be set to TRUE for case insensitive search.

- **ereg_replace** searches and replaces all the occurrences of a pattern inside a string. It returns the modified string, or the original string if the pattern did not match. Its arguments are:
  - **string**, the original string,
  - **pattern**, the pattern that should be matched,
  - **replace**, the replacement, which may contain escape sequences like **1** to reference found sub-patterns. The index is the number of the opening parenthesis, as usual[24],
  - **icase**, the case insensitive flag.
- **eregmatch** searches for a pattern into a string and returns NULL if it did not match or an array of all found sub-patterns. There is at least one returned pattern, which is the part of the string that matched the whole pattern. For those used to Perl, the elements of the returned array are equivalent to **$0**, **$1**, **$2**...[25]. Its argument are
  - **icase**,
  - **pattern**,
  - **string**.

Note that all the regex functions work the same way. If you want to match from the beginning / end of your string (or your line, in the case of **egrep**), you'll have to use ^ or **$**. If you want to eliminate what's before or after a pattern with **ereg_replace**, you'll have to play with something like **^.*** or **.*$** and **\1**.
You should read your (POSIX) system manual for details on regular expressions.

- **hex** converts its unnamed integer argument into the hexadecimal representation. It returns a string.
- **hexstr** takes one unnamed string argument and returns a string made of the hexadecimal representation of the ASCII codes of each input character. For example, **hexstr('aA\n')** returns **'61410a'**.
- **insstr** takes three or four unnamed arguments: a first string, a second string, a start index and an optional end index . Indexes starts at 0.
  The function replaces the declared slice in the first string by the second string, and returns the result. For example,
  `insstr('abcdefgh', 'xyz', 3, 5)`
  returns **'abcxyzgh'**.
- **int** converts its unnamed argument into an integer. If the argument is not a string, it returns **0**.

[24] For example,
`ereg_replace(string:'ZABCABD',pattern:'A([ABC]+)D',replace:'\\1')`
will return **'ZBCAB'**.
[25] For example,
`v = eregmatch(string:'XYZ IADAOZOOH',pattern:'([AEIOU]+).*(Z.*H)');`
will set **v[0]='I'ADA OZOOH' v[1]='IA'**
and **v[2]='ZOOH'**.

- **match** matches a string against a simple shell-like pattern and returns TRUE or FALSE. This function is less powerful than **ereg** but it is quicker and its interface is simple. Its arguments are:
  - **icase** if the match should be case insensitive.
  - **string** is the input string.
  - **pattern** is the searched pattern. The only wildcards are * (for any string, even empty) and **?** (for any character).
- **ord** takes one unnamed string argument and returns the (integer) ASCII code of the first character of the string.
- **raw_string** takes any number of unnamed arguments and returns a "pure" string resulting from these operations:
  - "Impure" strings are parsed and escaped sequences are interpreted[26].
  - Each integer is converted to the corresponding ASCII character[27].
  - Undefined variables are skipped[28].
  - Arrays are converted to some ASCII representation[29].
  - "Pure" strings are left as they were
  - And last but not least, the processing stops as soon as RAW_STR_LEN = 32768 have been entered. **string** does not have such a limitation.
- **str_replace** replaces any occurence of a substring inside a bigger string and returns the modified string. Its arguments are:
  - **string** is the original string.
  - **find** is the sub-string that is looked for.
  - **replace** is the replacement sub-string.
  - **count** is optional; if set, **str_replace** stops after this number of occurences have been replaced and leave the rest of the string as it is.
- **string** takes any number of unnamed arguments and returns a "pure" string[30] resulting from these operations:
  - "Impure" strings are parsed and escaped sequences are interpreted.
  - Integer are converted to their ASCII representation (in decimal base). That's where it is different from **raw_string**.
  - Undefined variables are skipped[31].

[26] In NASL1, only the first character of the string was kept.

[27] That's the only way to enter a null character into a string in older version of NASL2. Remember this if you want to be portable on old Nessus versions.

[28] Old versions of Nessus 1.3 were badly designed and **string** stopped processing its arguments at the first undefined value. Other functions may suffer from this bug; do not hesitate to tell.

[29] Which is not necessarily a good idea. Maybe we should expand them; the problem is hash elements are not ordered.

[30] Note that its size is unlimited

[31] Old versions of Nessus 1.3 were badly designed and **string** stopped processing its arguments at the first undefined value. Other functions may suffer from this bug; do not hesitate to tell.

  - Arrays are converted to some ASCII representation.
  - "Pure" strings are left as they were.

- **strcat** takes any number of unnamed arguments and returns a "pure" string resulting from these operations:
  - Integer are converted to their ASCII representation (in decimal base).
  - Undefined variables are skipped.
  - Arrays are converted to some ASCII representation[32].
  - "Pure" and "impure" strings are left as they were.

- **stridx** takes two or three unnamed arguments, looks for a substring inside a string (starting from the optional position) and returns its index (or -1 if not found or in case of error).
  - The first argument is the string (the haystack).
  - The second is the substring that is looked for (the needle)
  - The optional third argument is the starting position (by default **0**)
  - Note that the return value is not **NULL** if the substring was not found but **-1**.

- **strstr** takes two unnamed string arguments and searches the first occurrence of arg2 into arg1. It returns NULL if nothing was found, or the piece of arg2 from the first matching character till the end. For example **strstr('zabadz', 'ad')** returns **'adz'**.

- **split** splits a string into an array of "lines" or "sub strings". It takes an unnamed parameter (the input string), an optional **sep** string argument and an optional **keep** integer argument; it returns the array.
  If **sep** is not set, **split** cuts the input strings into lines. A line is supposed to end with the single character **LF** or the sequence **CR LF**.
  By default[33], the separator (whatever it is) will be included in the sub-strings or lines, unless **keep** is set top **0**

- **strlen** returns the length of the unnamed string argument. If the argument is not a string, you get an undefined result[34].

- **substr** takes two or three unnamed arguments: a string, a start index (counting from 0) and an optional end index (by default, the end). It returns the desired substring.
  For example, **substr('abcde', 2)** returns **'cde'** and **substr('abcde', 1, 3)** returns **'bcd'**.

- **tolower** converts its unnamed string argument to lower case.

- **toupper** converts its unnamed string argument to upper case.

[32] Which is not necessarily a good idea. Maybe we should expand them; the problem is hash elements are not ordered.

[33] The keep argument appeared in Nessus 2.0.2; older versions of the NASL library do not recognize it.

[34] Most of the time, the "internal size" of the data, which might be 0 even if it is not true!

3.2.7 HTTP functions

- **cgibin** takes no argument and returns the cgi-bin path elements. In fact the NASL interpretor forks and each process gets one value. This function should be considered as *deprecated* and **cgi_dirs()** should be used instead.

- **http_delete** formats an HTTP DELETE request for the server on the port. It will automatically handle the HTTP version and the basic or cookie based authentication. The arguments are **port** and **item** (the URL). **data** is not compulsory and probably useless in this function. It returns a string (the formatted request).

- **http_get** formats an HTTP GET request for the server on the port. It will automatically handle the HTTP version and the basic or cookie based authentication. The arguments are **port** and **item** (the URL). **data** is not compulsory and probably useless in this function. It returns a string (the formatted request).

- **http_close_socket** closes a socket. Currently, it is identical to **close** but this may change in the future.

- **http_head** formats an HTTP HEAD request for the server on the port. It will automatically handle the HTTP version and the basic or cookie based authentication. The arguments are **port** and **item** (the URL). **data** is not compulsory and probably useless in this function. It returns a string (the formatted request).

- **http_open_socket** opens a socket to the given port. Until Nessus 2.0.10, this functions is identical to **open_sock_tcp**; afterwards, it sets a 64K buffer for IO.

- **http_recv_headers** reads all HTTP headers on the given socket (unnamed integer argument). It stops at the first blank line and returns a string made of all headers, starting with the HTTP answer code.

- **http_post** formats an HTTP POST request for the server on the port. It will automatically handle the HTTP version and the basic or cookie based authentication. The arguments are **port**, **item** (the URL) and **data**. It returns a string (the formatted request).

- **http_put** formats an HTTP PUT request for the server on the port. It will automatically handle the HTTP version and the basic or cookie based authentication. The arguments are **port**, **item** (the URL) and **data**. It returns a string (the formatted request).

- **is_cgi_installed** tests if a CGI is found. If the path is relative (does not start with a slash), the CGI is search into the cgi-bin path. This functions returns the port of the web server where it was found (it will fork if there are several web servers); this magical behavior allows you to write very short plugins. For example:
  `if (port = cgi_installed("vuln.cgi")) security_warning(port);`
  The arguments are:

  - **item**, for the CGI path,
  - and **port**; by default, the function will look on all found web servers (i.e. read the KB entry **Services/www**).

3.2.8 Raw IP functions

All those functions work on blocks of data which are implemented as “pure strings”. This means that you could change them with the string manipulation functions, but this is probably not very easy.

- **dump_ip_packet** dumps IP datagrams. It takes any number of unnamed (string) arguments and does not return anything.
- **dump_tcp_packet** dumps the TCP parts of datagrams. It takes any number of unnamed arguments.
- **dump_udp_packet** dumps the UDP parts of datagrams. It takes any number of unnamed arguments.
- **forge_icmp_packet** fills an IP datagram with ICMP data. Note that the **ip_p** field is not updated. It returns the modified IP datagram. Its arguments are:
  - **data** is the payload.
  - **icmp_cksum** is the checksum, computed by default.
  - **icmp_code** is the ICMP code.
  - **icmp_id** is the ICMP ID.
  - **icmp_seq** is the ICMP sequence number.
  - **icmp_type** is the ICMP type.
  - **ip** is the IP datagram that is updated.
  - **update_ip_len** is a flag (TRUE by default). If set, NASL will recompute the size field of the IP datagram.
- **forge_igmp_packet** fills an IP datagram with IGMP data. Note that the **ip_p** field is not updated. It returns the modified IP datagram. Its arguments are:
  - **code**
  - **data**
  - **group**
  - **ip** is the IP datagram that is updated. Note that the IGMP checksum is automatically computed.
  - **type**
  - **update_ip_len** is a flag (TRUE by default). If set, NASL will recompute the size field of the IP datagram.
- **forge_ip_packet** returns an IP datagram inside the block of data. The named argument are:
  - **data** is the payload.
  - **ip_hl** is the IP header length in 32 bits words. **5** by default.
  - **ip_id** is the datagram ID; by default, it is random.
  - **ip_len** is the length of the datagram. By default, it is **20** plus the length of the **data** field.

  - **ip_off** is the fragment offset in 64 bits words. By default, **0**.
  - **ip_p** is the IP protocol. **0** by default.
  - **ip_src** is the source address in ASCII. NASL will convert it into an integer in network order.
    Note that the function accepts an **ip_dst** argument but ignore it!
  - **ip_sum** is the packet header checksum. It will be computed by default.
  - **ip_tos** is the "type of service" field. **0** by default
  - **ip_ttl** is the "Time To Live". **64** by default.
  - **ip_v** is the IP version. **4** by default.

- **forge_tcp_packet** fills an IP datagram with TCP data. Note that the **ip_p** field is not updated. It returns the modified IP datagram. Its arguments are:

  - **data** is the TCP data payload.
  - **ip** is the IP datagram to be filled.
  - **th_ack** is the acknowledge number. NASL will convert it into network order if necessary.
  - **th_dport** is the destination port. NASL will convert it into network order if necessary.
  - **th_flags** are the TCP flags.
  - **th_off** is the size of the TCP header in 32 bits words. By default, **5.**
  - **th_seq** is the TCP sequence number. NASL will convert it into network order if necessary.
  - **th_sport** is the source port. NASL will convert it into network order if necessary.
  - **th_sum** is the TCP checksum. By default, the right value is computed.
  - **th_urp** is the urgent pointer. **0** by default.
  - **th_win** is the TCP window size. NASL will convert it into network order if necessary. **0** by default.
  - **th_x2** is a reserved field and should probably be left unchanged.
  - **update_ip_len** is a flag (TRUE by default). If set, NASL will recompute the size field of the IP datagram.

- **forge_udp_packet** fills an IP datagram with UDP data. Note that the **ip_p** field is not updated. It returns the modified IP datagram. Its arguments are:

  - **data** is the payload.
  - **ip** is the old datagram.
  - **uh_dport** is the destination port. NASL will convert it into network order if necessary.
  - **uh_sport** is the source port. NASL will convert it into network order if necessary.
  - **uh_sum** is the UDP checksum. Although it is not compulsary, the right value is computed by default.

  - **uh_ulen** is the data length. By default it is set to the length the **data** argument plus the size of the UDP header.
  - **update_ip_len** is a flag (TRUE by default). If set, NASL will recompute the size field of the IP datagram.

- **get_icmp_element** returns an ICMP element from a IP datagram. It returns a data block or an integer, according to the type of the element. Its arguments are:
  - **element** is the name of the TCP field (see **forge_tcp_packet**).
  - **icmp** is the IP datagram (*not* the ICMP part only).

- **get_ip_element** extracts a field from a datagram. It returns an integer or a string, depending on the type of the element. It takes two named string arguments:
  - **element** is the name of the field, e.g. **"ip_len"** ou **"ip_src"**.
    Note that **"ip_dst"** works here!
  - **ip** is the datagram or fragment.

- **get_tcp_element** returns a TCP element from a IP datagram. It returns a data block or an integer, according to the type of the element. Its arguments are:
  - **element** is the name of the TCP field (see **forge_tcp_packet**).
  - **tcp** is the IP datagram (*not* the TCP part only).

- **get_udp_element** returns an UDP element from a IP datagram. It returns a data block or an integer, according to the type of the element. Its arguments are:
  - **element** is the name of the UDP field (see **forge_udp_packet**).
  - **udp** is the IP datagram (*not* the UDP part only).

- **insert_ip_options** adds an IP option to the datagram and returns the modified datagram. Its arguments are:
  - **code** is the number of the option.
  - **length** is the length of the option data.
  - **ip** is the old datagram.
  - **value** is the option data.

- **pcap_next** listens to one packet and returns it. Its arguments are:
  - **interface** is the network interface name. By default, NASL will try to find the best one.
  - **pcap_filter** is the BPF filter. By default, it listens to everything.
  - **timeout** is **5** seconds by default.

- **set_ip_elements** modifies the fields of a datagram. The named argument **ip** is the datagram; the other arguments are the same as **forge_ip_packet**. Once again, **ip_dst** is ignored. It returns the new datagram.

- **set_tcp_elements** modifies the TCP fields of a datagram. The named argument **tcp** is the IP datagram; the other arguments are the same as **forge_tcp_packet**. It returns the new IP datagram.
- **set_udp_elements** modifies the UDP fields of a datagram. The named argument **udp** is the IP datagram; the other arguments are the same as **forge_udp_packet**. It returns the new IP datagram.
- **send_packet** sends a list of packets (passed as unnamed arguments) and listens to the answers. It returns a block made of all the sniffed "answers".
  - **length** is the length of each packet by default.
  - **pcap_active** is TRUE by default. Otherwise, NASL does not listen for the answers.
  - **pcap_filter** is the BPF filter. By default it is **"ip and (src host** target **and dst host** nessus_host**)"**.
  - **pcap_timeout** is **5** by default.

3.2.9 Cryptographic functions

They are only implemented if Nessus is linked with OpenSSL.

- **HMAC_DSS** takes two named string arguments (**data** and **key**) and returns the HMAC as a string.
- **HMAC_MD2** takes two named string arguments (**data** and **key**) and returns the HMAC as a string.
- **HMAC_MD4** takes two named string arguments (**data** and **key**) and returns the HMAC as a string.
- **HMAC_MD5** takes two named string arguments (**data** and **key**) and returns the HMAC as a string.
- **HMAC_RIPEMD160** takes two named string arguments (**data** and **key**) and returns the HMAC as a string.
- **HMAC_SHA** takes two named string arguments (**data** and **key**) and returns the HMAC as a string.
- **HMAC_SHA1** takes two named string arguments (**data** and **key**) and returns the HMAC as a string.
- **MD2** takes an unnamed string argument and returns the hash as a string.
- **MD4** takes an unnamed string argument and returns the hash as a string.
- **MD5** takes an unnamed string argument and returns the hash as a string.
- **RIPEMD160** takes an unnamed string argument and returns the hash as a string.
- **SHA** takes an unnamed string argument and returns the hash as a string.
- **SHA1** takes an unnamed string argument and returns the hash as a string.

3.2.10 Miscellaneous functions

- **cvsdate2unixtime** takes one named string argument (date) and returns the number of seconds since 1970. The argument is supposed to be a date field automatically generated by CVS; the purpose of this function is to detect out of date plugins.
- **defined_func** takes one unnamed string argument and returns TRUE if a function with this named is defined. Whether it is a user or a built-in function does not matter.
- **dump_ctxt** is a debugging function which is not very useful for end users. It does not take any argument.
- **func_has_arg** takes a first string arguments (the function name) and a second string or integer argument (the argument name or number). It returns TRUE if the function accepts this argument, FALSE otherwise.
- **func_named_args** takes one unnamed string argument (the function name) and returns an array of all named arguments.
- **func_unnamed_args** takes one unnamed string argument (the function name) and returns the number of unnamed arguments.
- **gettimeofday** takes no argument and returns the number of seconds and microseconds since January 1st 1970. The return value is a character string formated like a floating point number: the seconds are on the left of the decimal point and the microseconds on the right, on six digits. For example: **"1067352015.030757"** means **1067352015** seconds and **30757** microseconds.
  The string manipulation functions can be used to extract the two numbers**.** e.g. **v = split(value, sep:'.');** would convert it into an array of two elements.
- **isnull** takes one unnamed argument and returns TRUE if it is not initialized, and FALSE otherwise.
  Remember that most of the time, (**x == NULL**) will not give the same result as **isnull(x)**
- **localtime** takes one integer unnamed argument (a "Unix time" = number of seconds since 1970-01-01) and one boolean named argument **utc**. Both can be ommited: by default, the time is the current time and **utc** is **FALSE**. The function returns an array that contains those keys[35]:

sec	The number of seconds after the minute, normally in the range 0 to 59, but can be up to 61 to allow for leap seconds.
min	The number of minutes after the hour, in the range 0 to 59.
hour	The number of hours past midnight, in the range 0 to 23.
mday	The day of the month, in the range 1 to 31.
mon	The number of the month, in the range 1 to 12.

[35]The values are slightly different from the structure returned by "localtime" or "gmtime". Some counts start at 1 instead of 0. I find this more intuitive.

year The year (4 digits).

wday The number of days since Sunday, in the range 0 to 6.

yday The current day in the year, in the range 1 to 366.

isdst A flag that indicates whether daylight saving time is in effect at the time described. The value is positive if daylight saving time is in effect, zero if it is not, and negative if the information is not available.

- **make_array** takes any *even* number of unnamed arguments and returns an array made from them. Contrary to **make_list**, only "atomic" values are accepted. The first argument in each pair is the key (either an integer or a character string), the second is the value. For example, **v=make_array(1,'one', 'Two', 2);** is equivalent to **v[1]='one'; v['Two']=2;**
  make_array can return arrays with duplicated keys, that have to be converted with **make_list** or walked through with **foreach**

- **make_list** takes any number of unnamed arguments of any types and returns an array made from them. If an argument is an array, it is split into its elements (i.e. make_list does not create a multi-dimensional array); the "integer indexed" elements will be re-indexed but the order will be kept.
  e.g., this:
  `v = make_list(0,-1,'two'); w = make_list('A', v);`
  is equivalent to:
  `v[0] = 0; v[1] = -1; v[2] = 'two';`
  `w[0] = 'A'; w[1] = 0; w[2] = -1; w[3] = 'two';`

- **max_index** takes one unnamed array argument and returns the bigger integer index used plus 1.
  e.g., to add an element at the end of any array, you may write `w[max_index(w)] = value;`

- **mktime**(sec, min, hour, mday, mon, year, isdst) takes seven integer named arguments and returns the "Unix time" (= number of seconds since 1970-01-01) as an integer, or **NULL** if some values are invalid. The arguments have the same meaning as the keys used in **localtime** (see above), but there are no **wday** or **yday** arguments.
  Default values are zero for all arguments, which is invalid for **year**, but not for **mon** or **mday**: C mktime normalizes the date[36].
  year can be on 4 digits or 2 digits; in this case, **1900** is added to the value before processing. **104** means **2004**.

- **replace_or_set_kb_item** calls **replace_kb_item** if this function exists, **set_kb_item** otherwise. It takes too named arguments (name & value).

- **safe_checks** takes no argument and returns the boolean value of the "safe checks" option.
  Dangerous plugins which may crash the remote service are expected to change their behavior when "safe checks" is on. Usually, they just identify the service version (e.g. from the banner) and check if it is known as vulnerable.

[36] See "man 3 mktime". 40 October => 9 November.

In “safe checks” mode, plugins from the most dangerous “categories” (ACT_DESTRUCTIVE_ATTACK, ACK_DENIAL and ACT_KILL_HOST) are not launched. So you do not need to test the value of **safe_checks** in those scripts.
You shouldn’t either write code like `if (safe_checks()) exit(0);`. If you do not want to run your test in this mode (e.g. because you do not know how to parse the banner), you should move your plugin to one of those “dangerous” categories, probably **ACT_DESTRUCTIVE_ATTACK**.

- **sleep** takes one unnamed integer argument and waits for this number of seconds.
- **type_of** returns the type of the argument. The return value is a string:
  - **"undef"** if the variable / argument is not initialized.
  - **"int"** if it is an integer.
  - **"string"** it if is an “impure string”.
  - **"data"** if it is a “pure string”.
  - **"unknown"** if the type is unknown, which means that you have found a bug in the interpretor!
- **usleep** takes one unnamed integer argument and waits for this number of microseconds.
- **unixtime** returns the current Unix time, i.e. the number of seconds since January 1st 1970.

3.2.11 “unsafe” functions

The following functions are only allowed in “trusted” signed scripts[37]. If they could run anywhere, a user could upload a script and run arbitrary root code or perform a denial of service aginst the Nessus server.

- **find_in_path** searches a command in **$PATH** and returns **TRUE** if found, or **FALSE** if not. It takes one string argument (the command name).
- **pread** launches a process, reads its whole output and returns it as a string. The arguments are:
  - **cmd** is the name of the program that will be run. If it is not an absolute path, the program will be searched in **$PATH**.
  - **argv** is an array of strings. Each string is an argument. Note that **argv[0]** is the name of the program (which may be different from **cmd**, but will be equal in most cases).
  - **cd** is a boolean, **FALSE** by default. If **TRUE**, Nessus changes its current directory to the directory where the command was found.
  - **nice**[38] is an integer which changes the son process priority. You want to set it to a positive value if you launch CPU hog commands.

[37] The command line interpretor trusts the script if the option -X is given. And the Nessus server trusts any script if **nasl_no_signature_check** is set to **yes** in **nessusd.conf**

[38] This argument appeared in version 2.1.2.

- **file_close** takes a file descriptor (unnamed integer argument), closes it and returns **0** or **NULL** if there was a problem.
- **file_open** takes two named string arguments and returns a file descriptor (integer):
  - **mode** is a string: “r” or “w”.
  - **name** is the file name.
- **file_read** takes two named integer arguments and returns the data:
  - **fp** is the file descriptor.
  - **length** is the desired data length.
- **file_seek** takes two named integer arguments and seeks into the file. It returns NULL if there was an error or 0 if it worked.
  - **fp** is the file descriptor.
  - **offset** is the desired absolute offset (= position from the begining of the file).
- **file_stat** takes a file name (unnamed string argument) and returns the file size or **NULL** if there was a problem (unexisting file, for example).
- **file_write** takes two arguments and returns the number of bytes that were written.
  - **fp** is the file descriptor (integer).
  - **data** is the buffer (string).
- **fread**[39] reads a file on the Nessus server. It takes one unnamed string argument (the file name) and returns the whole file content in a string variable or **NULL** if an error occured.
- **fwrite** writes a file on the Nessus server. It takes two named string argument (the file name) and returns the number of written byte or **NULL** if an error occured.
  - **data** is the data that will be written to the file.
  - **file** is the file name.
- **get_tmp_dir** returns a temporary directory name including the trailing slash.
- **unlink**[40] removes a file on the Nessus server. It takes one unnamed string argument (the file name) and does not return any value.

3.3 NASL library

It is implemented through “include files”. Some of the functions are not very interesting because they were not designed to be called directly: they are used by other functions in the “.inc” file.

[39] This function appeared in Nessus 2.1.2. Previous versions can emulte it with something like: `x = pread(cmd: "/bin/cat", argv: make_list("cat", file_name));`

[40] This function appeared in Nessus 2.1.2. Previous versions can emulte it with something like: `x = pread(cmd: "/bin/rm", argv: make_list("rm", file_name));`

3.3.1 dump.inc

- **dump**(ddata, dtitle)
  prints the optional title and dumps the data block to the standard output. This function is useful for debugging only.

- **hexdump**(ddata)
  dumps a data block into hexadecimal and returns the results (as a string).

3.3.2 ftp_func.inc

- **ftpclose**(socket)
  cleanly closes a FTP connection: sends "QUIT", waits for the answer and then closes the socket. This functions does not return any value.

- **get_ftp_banner**(port)
  returns the FTP banner that was stored in the KB under **"ftp/banner/**port_number". If the KB item is not present, the function connects to the FTP server, reads the banner, stores it into the KB and returns it.

- **ftp_recv_line**(socket)
  reads a line on the socket until the 4^{th}character is different from "-". Useful to skip a long login banner.

3.3.3 http_func.inc

- **check_win_dir_trav**(port, url, quickcheck)
  connects to port and sends a HTTP GET request to the given **url**. You are supposed to try to access AUTOEXEC.BAT, BOOT.INI or WIN.INI
  If **quickcheck** is TRUE, the function returns TRUE if it gets a 200 (OK) answer.
  If **quickcheck** is FALSE, it looks for pattern in the answer; it will returns TRUE if it can find "ECHO", "SET", "export", "mode", "MODE", "doskey", "DOSKEY", "[boot loader]", "[fonts]", "[extensions]", "[mci extensions]", "[files]", "[Mail]", or "[operating systems]".
  You are supposed to set **quickcheck** if the server answers with clean 404 codes to requests to unknown pages, i.e. if "**www/no404/**port" is not set in the KB.

- **get_cgi_path**(port)
  returns the list of directories where the CGI might be installed. The list is a string where the items are separated with ":".
  WARNING: this function is not a good idea and may disappear in the future.

- **get_http_banner**(port)
  returns the HTTP banner that was stored in the KB under **"www/banner/**port_number". If the KB item is void, the function connects to the HTTP server, sends a GET request, and stores the result into the KB.

- **get_http_port**(default)
  reads the KB item **"Services/www**", verifies that the port is open, that there is an HTTP server behind it, and returns it. Note that the function will fork if there are several web servers on the target machine.
  If the KB item is void, the **default** port is tested.
  If no HTTP port is found, *the script exits*.

- **http_40**x(port,code)
  returns **TRUE** if the HTTP answer "code" is between 400 and 409 or something identified by no404.nasl; **FALSE** otherwise.

- **http_is_dead**(port, retry)
  tries very hard to test if the web server is still alive even if there is a transparent or reverse proxy on the way. It sends a HTTP GET request for a random page (**/NessusTest**<rand>**.html**) and waits for the answer. The optional argument **retry** is the number of times it should wait (one second) and retry to open the socket to the remote service if this failed in the first time (by default, there is no retry).
  It returns TRUE if

  - the connection was refused, or
  - no valid HTTP answer was received, or
  - a 502 (bad gateway) or 503 (service unavailable) was received.

- **http_recv_body**(socket, headers, length)
  reads N bytes from the **socket**. N is defined like this:

  - If the **header** field is not defined, the function first calls **http_recv_headers**; the "Content-Length" field is extracted from the headers.
    Note that the headers will not be returned, only the HTTP "body".
  - Then, if **length** is set
    - if content_length could be extracted from the headers, N = max(length, content_length)
    - otherwise, N = length
  - else if content_length could be extracted from the headers, N = content_length,
  - else N defaults to 8192 bytes.

- **http_recv**(socket, code)
  reads the HTTP headers and data from the socket and returns all this.
  This function is efficient because it just reads the right number of bytes without waiting for a network timeout. The code argument is optional. If you read the HTTP code (with **recv_line**), you have to put it into this argument[41].

- **http_recv_length**(socket, bodylength)
  reads the HTTP headers, then calls **http_recv_body** with length=bodylength, and returns the concatenated headers and body.

- **locate_cgi**(port, item)
  looks for a given CGI on a web server. It returns its path if it could be found, or NULL otherwise.
  WARNING: the implementation is wrong, so this function may disappear in the future.

[41] In fact, **http_recv** only needs to know that the code was read, because **http_recv_header** may not work in this case. **http_recv** uses its own loop to read the remaining headers before the body.

- **php_ver_match**(banner, pattern)
  the function returns TRUE if the regex pattern matches a “Server:” ou “X-Powered-by:” line in the banner. A way to use this function is, for example:

```
if (php_ver_match(banner:banner,
     pattern:".*PHP/((3.*)|(4\.0.*)|(4\.1\.[01].*))"))
 security_hole(port);
```

- **cgi_dirs()**
  returns an array containing all the directories that may have CGIs in it (by default /cgi-bin and /scripts). Several scripts try to augment this list (in particular *webmirror.nasl*).

3.3.4 http_keepalive.inc

Nessus 2.0.1 and newer support HTTP keep-alive connections, which avoid to re-open a socket for each request. This saves bandwidth and CPU cycles, especially through SSL/TLS. At this time, only the requests made from within the same plugin can be kept alive, however sharing one socket among multiple plugins could be done in the future. To work properly, this file must be included after **http_func.inc**.

- **http_keepalive_send_recv**(port, req)
  sends the request **req** to the remote web server listening on port **port** and returns the result of the request, or NULL if the connection could not be established. Internally, this function will automatically determine if the remote host supports Keep-Alive connections and will restore the connection if it was cut. **req** is a full HTTP request, as returned by **http_get()**.
  It is not recommanded to send potentially destructive attacks on top of a kept-alive connection.

- **is_cgi_installed_ka(**port, item)
  acts the same way as **is_cgi_installed()** but on top of a kept-alive connection.

- **check_win_dir_traversal_ka**(port, url, quickcheck)
  acts the same way as **check_win_dir_traversal()** but on top ofa kept-alive connection.

3.3.5 misc_func.inc

- **register_service**(port, proto,ipproto)
  “registers” a service. Used values for the proto arguments are: **aos**, **bugbear**, **DCE/**guid, **dns**, **lpd**, **uucp**, **irc**, **daytime**, **ftp**, **smtp**, **nntp**, **ssh**, **auth**, **finger**, **www**, **mldonkey-telnet**, **nessus**, **QMTP**, **radmin**, **RPC/**name, **portmapper**, **rsh**, **x11**, **xtel**, **xtelw**.
  By default, **ipproto** is **tcp**; **udp** was introduced in Nessus 2.1.2 and is used by experimental scripts only, at this tome.
  In practice, this function defines two items in the KB:

  – **Known/tcp/**port = proto
    or **Known/udp/**port = proto

  - **Services**/proto = port
    or **Services/udp/**proto = port
    This may create a list if several servers are known on differents ports.

- **known_service**(port,iproto)
  returns the service name[42] if the service is known on the port, **NULL** otherwise. Note that if the service was "registered" several times, **known_service** may fork. So the best way to use this function is to exit if it returns a defined value. For example:

```
port = get_kb_item("Services/unknown");
# This was set by find_service.nes but another plugin
# may have identified the service. So:
if (known_service(port: port)) exit(0);
```

- **get_unknown_banner**(port, dontfetch, ipproto)
  reads **unknown/banner/**port from the KB. If a value is found, it is returned. If no value is found and **dontfetch** is set, the function returns NULL. Otherwise the function connects to the port, tries to read a banner, stores it in the KB and returns it.
  By default, **ipproto** = **tcp**

- **set_unknown_banner**(port, banner, ipproto)
  sets **unknown/banner/**port to **banner** in the KB.
  By default, **ipproto** = **tcp**

- **get_service_banner_line**(service, port,ipproto)
  reads **Services/**service from the KB. If no value is found, uses the **port** parameter. It then reads service**/banner/**port from the KB; if it exists, it is returned. If not, the function connects to the port, reads one line and returns it, *but does not store it in the KB.*
  Note that this function may fork.
  By default, **ipproto** = **tcp**

- **get_rpc_port**(program, protocol)
  calls the portmapper and gets the port where the service specified by the parameters is located. **program** is a RPC number and **protocol** may be IPPROTO_TCP or IPPROTO_UDP. If the portmapper could not be reached or the service is down, the function returns **0**.

- **service_is_unknown**(port,ipproto)
  returns **TRUE** if the service was "registered" (see above) or **FALSE** otherwise. This function does not fork!
  ipproto is **tcp** by default.

3.3.6 nfs_func.inc

NFS read and write functions are not defined yet. You can only mount a NFS share and inspect its contents.

[42] **proto** parameter for **register_service**

- **mount**(soc, share)
  attempts to mount **share** (defined in **NFS/exportlist** in the KB). **soc** is a UDP socket opened to the remote mount daemon (mountd, rpc program#100005). This function returns NULL in case of failure, or a file handle (fid) in case of success.

- **umount**(soc, share)
  unmounts **share** - basically, this tells the remote mount daemon that we will stop using its services. **soc** is a UDP socket opened to the remote mount daemon.

- **readdir**(soc, fid)
  returns the content of the directory pointed by **fid**. **soc** is a UDP socket opened to the remote NFS daemon (nfsd, rpc program #100003). This function returns an array.

- **cwd**(soc, fid, dir)
  changes directories. **soc** is a UDP socket opened to the remote NFS daemon, **fid** is the current working directory and **dir** is the name of the directory we would like to change it. This function returns NULL on failure, or a handle (fid) to the directory we changed to.

3.3.7 smb_nt.inc

The SMB library provides a way to interact with Windows hosts using SMB, either on top of port 139 or on top of port 445. Since Microsoft protocol is barely documented, most if not all of these functions have been coded by packet analysis. Therefore, the name of the functions may vary compared to what you would find in Microsoft-Land.

The functions described here are both low-level and high-level. This a description of the SMB protocol (and DCE/RPC over SMB) is beyond the scope of this manual, we suggest you refer to the books listed in the bibliography if needed. The functions are defined in this guide in the order they are usually used :

Setting up an SMB session

- **smb_session_request**(soc, remote)
  pre-establishes a SMB session with the remote host. **soc** is a socket opened to port 139 or 445 if the remote host supports it. You must open the connect to the port pointed by the KB item **SMB/transport**, which is defined in the plugin *cifs445.nasl*. **remote** is the netbios name of the remote host (as stored in the KB item **SMB/name**, created in the plugin *netbios_name_get.nasl*). If the name is not defined you can try to use ***SMBSERVER** which is recognized by most SMB hosts. If the connection takes place on top of port 445, this function immediately returns as it is unnecessary in this case.

- **smb_neg_prot**(soc)
  negociates the protocol we will use to log into the remote host. This function asks for NTLMv1 authentication if possible, and returns a buffer suitable to be used with **smb_session_setup()**, which contains the authentication protocols the remote host supports. **soc** must be the socket opened to the remote SMB server, and a call to **smb_session_request()** has to be made before this function is called.

- **smb_session_setup**(soc, login, password, domain, prot)
  setups the SMB session to the remote host. It logs as **login** with the password **password**, in the domain **domain** (which can be NULL, in which case the function will log locally). This function returns a buffer suitable to use with the function **session_extract_uid()**, or NULL if the authentication failed. Internally, the function will use either clear-text or NTLMv1 authentication, depending on what the remote host supports and the options set by the user. **prot** is the buffer returned by the function **smb_neg_prot()**. **soc** must be the socket opened to the remote SMB server, and a call to **smb_neg_prot()** must have been made prior to calling this function. **smb_session_setup()** returns a buffer suitable to be used with **session_extract_uid()**.

- **session_extract_uid**(reply)
  extracts the user id from **reply**. It is used each time a new SMB call is made. It returns 0 if **smb_session_setup()** failed.

Connecting and reading from the remote shares Each SMB host exports shares - virtual directories accessible from accross the network, usually containing files. The list of shares exported by a given host is written in **SMB/shares**, which is written to by *smb_enum_shares.nasl***.**

- **smb_tconx**(soc, name, uid, share)
  connects to **share** (ie: "IPC$") on top of the socket **soc** connected to the smb host whose name is **name.** The option **uid** comes from the call to **session_extract_uid()**. This function returns a buffer suitable to be used with **tconx_extract_tid()**.

- **tconx_extract_tid**(reply)
  extracts the tree id from **reply**, which is a buffer returned by a call to **smb_tconx()**. It returns 0 if the call to **smb_tconx()** failed.

- **OpenAndX**(socket, uid, tid, file)
  opens **file** on the share pointed by **tid**, and returns a file id (fid) or NULL if the call failed (ie: file does not exist or can not be read).

- **ReadAndX**(socket, uid, tid, count, off)
  reads **count** bytes starting at offset **off** in the file **fid** and returns the content (or NULL if the call failed)

- **smb_get_file_size**(socket, uid, tid, fid)
  returns the size of the file pointed by **fid.**

Accessing the remote registry

- **smbntcreatex**(soc, uid, tid)
  this function creates a connection to the remote \winreg named pipe. It should be rewritten to support a fourth argument (pipename) but it is not the case at this time. **soc** is a socket connected to the remote SMB host, **uid** is our user id (obtained via **smb_session_setup()** and **session_extract_uid()**) and **tid** is pointing to the special share IPC$. This function returns a buffer suitable to be used with **smbntcreatex_extract_pipe()** or NULL if the called failed (ie: there is no \winreg named pipe).

- **smbntcreatex_extract_pipe**(reply)
  extracts the pipe id from the buffer returned by **smbntcreatex()**. It returns 0 if the call failed.

- **pipe_accessible_registry**(soc, uid, tid, pipe)
  what this function does is quite unclear. It should be called before continuing to explore the registry, just after **smbntcreatex()**. **pipe** is the integer returned by **smbntcreatex_extract_pipe()**.

- **registry_access_step_1**(soc, uid, tid, pipe)
  this function should be renamed registry_open_hklm (and will probably be). It opens HIVE_KEY_LOCAL_MACHINE and returns a buffer suitable to use with **registry_get_key()** and **registry_get_key_security()**.

- **registry_get_key**(soc, uid, tid, pipe, key, reply)
  opens the registry key "**key**" (as in "SOFTWARE\Microsoft\Windows NT") and returns a buffer suitable to use with **registry_get_item_dword()**, **registry_get_item_sz()**, or **registry_get_key_security()**. **reply** is the buffer returned by **registry_access_step_1()**. This function returns NULL if the key does not exist or is not accessible.

- **registry_get_item_sz**(soc, uid, tid, pipe, item, reply)
  returns the content of **item** in the currently opened key (designated by **reply**, which is a buffer returned by **registry_get_key()**). It returns a buffer which needs to be decoded with **registry_decode_sz()**. **item** must be a string key value. If **reply** is the reply to a call to **registry_get_key**(key:"SOFTWARE\Microsoft\Windows NT"), **item** could be equal to "CurrentVersion".

- **registry_decode_sz**(data)
  decodes the value returned by **registry_get_item_sz()** and returns a string containing the value, or NULL if the call to **registry_get_item_sz()** failed.

- **registry_get_item_dword**(soc, uid, tid, pipe, item, reply)
  returns the content of **item** in the currently opened key (designated by **reply**, which is a buffer returned by **registry_get_key()**). It returns a buffer which needs to be decoded with **registry_decode_dword()**. **item** must be an integer key value.

- **registry_decode_dword**(data)
  decodes the value returned by **registry_get_item_dword()** and returns an integer containing the value, or NULL if the call to **registry_get_item_dword()** failed.

- **registry_get_key_security**(soc, uid, tid, pipe, reply)
  obtains the ACLs associated to the key opened with **registry_get_key()**. **reply** is the buffer returned by **registry_get_key()**. It returns a security descriptor which contains the ACLs and which has to be parsed manually. The function **registry_key_writeable_by_non_admin**() is a great example of usage for this.

- **registry_key_writeable_by_non_admin**(security_descriptor)
  decodes the buffer returned by **registry_get_key_security()** and returns TRUE if a user other than the owner of the key or a member of the administrator group can write to the key.

SAM access

- OpenPipeToSamr(soc, uid, tid)
- SamrConnect2(soc, tid, uid, pipe, name)
- _SamrEnumDomains(soc, uid, tid, pipe, samrhdl)
- SamrDom2Sid(soc, tid, uid, pipe, samrhdl, dom)
- SamrOpenDomain(soc, tid, uid, pipe, samrhdl, sid)
- SamrOpenBuiltin(soc, tid, uid, pipe, samrhdl)
- SamrLookupNames(soc, uid, tid, pipe, name, domhdl)
- SamrOpenUser(soc, uid, tid, pipe, samrhdl, rid)
- SamrQueryUserGroups(soc, uid, tid, pipe, usrhdl)
- SamrQueryUserInfo(soc, uid, tid, pipe, usrhdl)
- SamrQueryUserAliases(soc, uid, tid, pipe, usrhdl, sid, rid)

3.3.8 smtp_func.inc

- **smtp_send_socket**(socket, from, to, body)
  sends a SMTP message on an open socket and returns TRUE if the message for accepted for delivery, or FALSE if some problem occured.
- **smtp_send_port**(port, from, to, body)
  opens a socket to **port**, sends a SMTP message, and closes the socket. It returns TRUE if the message for accepted for delivery, or FALSE if some problem occured.
- **smtp_from_header**()
  returns the default "From" address. If the KB item **SMTP/headers/from** is not set, the default address is "nessus@example.com".
- **smtp_to_header**()
  returns the default "To" adress. If the KB item **SMTP/headers/to** is not set, the default address is "postmaster@[1.2.3.4]" (where 1.2.3.4 is the target host IP).
- **get_smtp_banner**(port)
  reads the KB item **smtp/banner/**port and returns it, or if it is not set, connects to the port, reads the SMTP banner, stores it into the KB and returns it.
- **smtp_recv_banner**(socket)
  reads lines from the socket and returns the first line that does not started with "220-".

3.3.9 telnet.inc

- **get_telnet_banner**(port)
  reads **telnet/banner/**port from the KB and returns it. If no value is found, connects to the port, grabs the telnet banner, stores it into the KB and returns it.

- **set_telnet_banner**(port, banner)
  writes **banner** into the KB item **telnet/banner/**port

3.3.10 uddi.inc

- **create_uddi_xml**(ktype,path,key,name)
  formats a UDDI XML query, whatever this means.
  Can anybody write something about this?

4 Hacking your way inside the interpretor

4.1 How it works

4.1.1 The parser

The lexical analyzer It is written directly in C because flex cannot generate C reentrant code[43]. That's why it is rather crude. Anyway, I was surprised to see that according to cachegrind, we do not lose much time in it.
The lexer entry point is the "mylex" function in **nasl_grammar.y**. The parser calls it; you are not supposed to do it. I mention it because that's where you can add "tokens".

The syntactic analyzer It is written in Bison and you *cannot* compile it with Yacc, because we use the **%pure_parser** instruction. This generates a reentrant parser, allowing us to handle "includes" very simply[44]. While reading the source, the parser builds a "syntax tree".

The syntax tree You can find a description of the "cell type" in **nasl_tree.h**. The only used data type is the **tree_cell** structure. Each cell maybe linked to children cells: from 0 (if it is a leaf) to 4 (if I remember correctly, only the "for" instruction uses this). For example, this code:

```
x = y * 2;
f(arg1: x);
```

will become this tree:

```
NODE_INSTR_L
1: NODE_AFF
   1: NODE_VAR Val="x"
   2: EXPR_MULT
      1: NODE_VAR Val="y"
      2: CONST_INT Val=2
2: NODE_INSTR_L
   1: NODE_FUN_CALL Val="f"
      1: NODE_ARG Val="arg1"
         1: NODE_VAR Val="x"
```

4.1.2 The interpretor

> To iterate is human, to recurse is divine.

The entry point is **nasl_exec**. This function takes two arguments (a "lexical context" and a "tree cell") and returns the result another "tree cell", the result of the evaluation of the a "tree cell" in the "context". To perform its job, **nasl_exec** calls itself again and again[45].

[43] It is able to generate reeentrant C++ code but we do not want to link Nessus with C++.

[44] OK, a good preprocessor could do it. But the fact that `include("file.inc");` is a simple instruction allows some interesting things, e.g.
`if (!defined_func("gizmo") include("gizmo_compat.inc");`.

[45] Although there are much quicker ways to interprete a language, walking along the syntax tree is simple. We know that we could run 10 times faster or even more by implementing a code generator and a Virtual Machine, but we do not need it yet. Maybe there will be a NASL3.

4.1.3 Memory management

Memory copy is expensive[46], memory allocation too. So I tried to avoid unnecessary duplications of "cells". That's why I implemented a poor man's garbage collector: each "cell" has a reference count. **ref_cell** increments it, and **deref_cell** decrements it. Once it reaches 0, the cell is freed[47].
To use, do not try to be smart, just follow a couple of simple rules:

- **nasl_exec** never tries to free its input argument.
- **nasl_exec** returns a value that is "referenced" (i.e. ref_count > 0). Once you have finished playing with it, you have to "dereference" it.
- Internal functions should return "referenced" cells.

4.1.4 Internal functions interfaces

Every internal function uses the same interface: it reads a "lexical context" on input and returns a "cell". The interface is described in details in the next paragraph.
The function name and NASL arguments are declared in **nasl_init.c**

4.2 Adding new internal functions

4.2.1 Interface

Every internal function has the same interface:

- it takes one input argument, a "lexical context". The NASL arguments are variables in the context, either "named" or "numbered". The context is chained to the calling context.
- and it returns a "tree cell". The returned cell should be "referenced" once; you shouldn't have to do anything as all the cell allocation functions set "ref_count" to 1.
  - If you do not want to return a value, returns **FAKE_CELL**.
  - If you want to return a serious error, returns **NULL**.

A simple example:

```
tree_cell*
my_test_function(lex_ctxt* lexic)
{
  fprintf(stderr, "My test function was called\n");
  /* let's look at the context */
  dump_ctxt(lexic);
  /* And return nothing (in NASL) */
  return FAKE_CELL;
}
```

[46] If you do not believe me, run a slow plugin like webmirror.nasl with cachegrind and look at the result.
[47] And if it becomes negative, the interpretor aborts because this is a serious bug! In fact, the reference count becomes negative when the cell is "referenced" too many times (integer roll over).

4.2.2 Reading arguments

The arguments are stored as “named” or “numbered” variables in the context. This NASL code:

```
f(1, "TWO", a: 33, z: "three");
```

will create four variables in the context, two “numbered” and two “named”:

- 0 -> 1
- 1 -> "TWO"
- a -> 33
- z -> "three"

To read those arguments, you can use one of those functions:

- char* **get_str_var_by_num** (lex_ctxt* lexic, int num)
  reads the variable and converts it to a string if necessary. Do not free the result and do not call the function twice in a row on a non-string variable[48] without copying the result somewhere, as the function returns a pointer to a static buffer in this case.
  If the variable is not initialized or cannot be converted to character, NULL is returned.
- int **get_int_var_by_num**(lex_ctxt* lexic, int num, int default_value)
  reads the variable and converts it to an integer if necessary.
  If the variable is not initialized or cannot be converted, the default value is returned.
- char* **get_str_local_var_by_num**(lex_ctxt* lexic, const char* name)
- int **get_int_local_var_by_num**(lex_ctxt* lexic, int num, int default_value)
- int get_local_var_size_by_name
- int get_var_size_by_num

4.2.3 Returning a value

Returning void is easy: just returns **FAKE_CELL** (which is currently defined as “**(void*)1**”, but this might change). To return a value, you have to allocate a cell, reference it once (this is automatically done by all the alloc_*cell functions) and put data into it. Examples:

```
tree_cell    *retc;
char         *p;
/* return 42 */
retc = alloc_typed_cell(CONST_INT);
retc->x.i_val = 42;
return retc;
```

[48] i.e. integer or array

```
/* return "abcd" */
retc = alloc_typed_cell(CONST_DATA);
retc->x.size = 4;
retc->x.str_val = p = emalloc(5);
strcpy(p, "abcd");
return retc;
```

4.2.4 Adding your function in nasl_init.c

Your function is not yet known to the NASL interpretor. You have to add it into **nasl_init.c**

4.2.5 Cave at

You should be careful not to open security holes with your new C functions. Here are examples of potentially dangerous system calls:

open — as it allows to read protected files if the argument is not properly checked (the Nessus daemon runs as root).

unlink — as it allows to delete protected files.

fork — as a malicious user may implement a fork bomb. More, it breaks the current model, where Nessus controls the son processes.

kill — as you might kill system processes if the arguments is not properly checked.

4.3 Adding new features to the grammar

4.3.1 caveat

First, if you do not know what "yacc" or "bison" do, how they do it and why, if you ignore what a lexical analyzer is, a regular expression or a LALR context-free grammar, a finite state machine or a stack automata, just *don't* touch the grammar.

This is important: the current grammar is clean. The precedence of every operator is clearly defined; the grammar has only one shift/reduce conflict, the classical "dangling else" ambiguity[49]. That's why there is an "**%expect 1**" directive. If you modify the grammar and add ambiguities, you are *not* supposed to solve them by increasing the expected number of conflicts. Do whatever is necessary (and clean) to remove them.

One last time: if you have never studied language theory and theoretical computer science, stop reading here!

4.3.2 Adding a new operator in the grammar

You will have to modify the lexical analyzer to recognize the token.

4.3.3 Adding a new type to the grammar

4.4 Checking the result

[49] In the construction "`if (T1) if (T2) I1; else I2;`" the "`else`" can be attached to the first or the second "`if`". All modern parsers attach it the second (= nearest) "`if`".

References

[RFC 821] SMTP protocol...

[RFC 854 / STD 8] Telnet protocol...

[RFC 1945] Hypertext Transfer Protocol – HTTP/1.0. T. Berners-Lee, R. Fielding, H. Frystyk. May 1996.

[RFC2246] The TLS Protocol Version 1.0. T. Dierks, C. Allen. January 1999.

[RFC2616] Hypertext Transfer Protocol – HTTP/1.1. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. June 1999.

[SSL v3] SSL 3.0 SPECIFICATION `http://wp.netscape.com/eng/ssl3/`

[SSL v3 (03/96)] `http://wp.netscape.com/eng/ssl3/` The SSL Protocol Version 3.0 - Internet Draft - March 1996 (Expires 9/96) - Alan O. Freier, Netscape Communications; Philip Karlton, Netscape Communications, Paul C. Kocher, Independent Consultant.

[SSL v3 (11/96)] `http://wp.netscape.com/eng/ssl3/draft302.txt` The SSL Protocol Version 3.0 - November 18, 1996 - Alan O. Freier, Netscape Communications; Philip Karlton, Netscape Communications, Paul C. Kocher, Independent Consultant.

[DCE/RPC] DCE/RPC over SMB - Luke Kenneth Casson Leighton - Macmillan Technical Publishing - ISBN 1-57870-150-3