Important Information

Latest Software
We recommend that you install the most recent software release to stay up-to-date with the latest functional improvements, stability fixes, security enhancements and protection against new and evolving attacks.

Latest Documentation
The latest version of this document is at: http://supportcontent.checkpoint.com/documentatin_download?ID=13945
For additional technical information, visit the Check Point Support Center (http://supportcenter.checkpoint.com).
For more about this release, see the home page at the Check Point Support Center (http://supportcontent.checkpoint.com/solutions?id=sk67581).

Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 April 2012</td>
<td>First release of this document</td>
</tr>
</tbody>
</table>

Feedback
Check Point is engaged in a continuous effort to improve its documentation.
Please help us by sending your comments (mailto:cp_techpub_feedback@checkpoint.com?subject=Feedback on CPcode for Check Point DLP R75.40 Reference Guide).
Contents

Important Information .................................................................................................................. 3
Developing Custom CPcode Functions ....................................................................................... 8
DLP CPcode Basics .................................................................................................................... 8
The func Statement ................................................................................................................... 8
Restriction on Executed CPcode Statements ............................................................................. 9
CPcode Scope ............................................................................................................................ 9
  Local Variables ....................................................................................................................... 10
  Function Calls ....................................................................................................................... 10
Using Built-in CPcode Functions ............................................................................................... 11
Fastpaths ..................................................................................................................................... 11
Optimizing CPcode .................................................................................................................. 12
  Optimizing Variables .............................................................................................................. 12
  Polymorphic Variables .......................................................................................................... 12
  Network Variables .............................................................................................................. 12
  Deleting Variables .............................................................................................................. 12
  Variables Or Literals ............................................................................................................ 13
Optimizing Arrays .................................................................................................................... 13
  Array Copies ....................................................................................................................... 13
  Index Usage ....................................................................................................................... 13
  Complex Indexes .............................................................................................................. 14
Optimizing Function Calls ....................................................................................................... 14
CPcode Lexicon ....................................................................................................................... 15
  Character Sets .................................................................................................................... 15
  Simple Operators ............................................................................................................... 15
  Compound Operators ....................................................................................................... 16
  Naming Variables .............................................................................................................. 16
  Special Local Variables ..................................................................................................... 17
  NULL .................................................................................................................................. 17
Reserved Words ...................................................................................................................... 18
Constants .................................................................................................................................. 19
Configuring Constants ............................................................................................................ 20
CPcode Data Types .................................................................................................................. 21
  array ................................................................................................................................. 21
  ethmac ............................................................................................................................. 22
  int ..................................................................................................................................... 22
  ipv4host and ipv6host ..................................................................................................... 23
  ipv4net and ipv6net ......................................................................................................... 23
  list ................................................................................................................................. 24
  pattern ........................................................................................................................... 24
  str ................................................................................................................................. 24
tag ..................................................................................................................................... 25
CPcode Operators .................................................................................................................... 26
  Arithmetic Precedence ..................................................................................................... 26
  Arithmetic Operators ..................................................................................................... 27
    + (addition) .................................................................................................................. 27
    - (subtraction) ............................................................................................................. 27
    * (multiplication) ....................................................................................................... 28
    / (division) ................................................................................................................ 28
    % (modulus) ............................................................................................................... 28
    - (negation) ............................................................................................................... 29
  Relational Operators ....................................................................................................... 29
    == (equal to) .............................................................................................................. 30
### Logical Operators
- `!=` (not equal to) .................................................. 31
- `>` (greater than) ................................................ 32
- `>=` (greater than or equal to) .............................. 32
- `<` (less than) .................................................. 33
- `<=` (less than or equal to) ................................... 33

### Bitwise Operators
- `&` (bitwise AND) ............................................ 35
- `|` (bitwise OR) .................................................. 36
- `~` (bitwise NOT) .................................................. 37
- `^` (bitwise XOR) .................................................. 37
- `<<` (left shift) .................................................. 38
- `>>` (right shift) .................................................. 38

### Other Operators
- `=` (assignment) .................................................. 38
- `inside` .................................................................. 39
- `?:` (conditional) .................................................. 40

### CPcode Statements
- `=` (assignment) .................................................. 41
- `{}` (block) ....................................................... 41
- `break` ................................................................ 42
- `foreach` .............................................................. 42
- `func` ................................................................ 43
- `if` .................................................................... 44
- `requires` ............................................................. 44
- `return` ............................................................... 45
- `while` ................................................................. 45

### CPcode Built-in Variables
- **System Variables** .............................................. 46
- **Feature Variables** ............................................. 46
- **Global Variables** .............................................. 47
- Example with Global Variables ............................... 49
- Recommendations with Global Variables ............... 50

### CPcode Built-in Functions
- `align()` ............................................................... 51
- `arrayindices()` .................................................... 53
- `atoi()` ............................................................... 53
- `blobbytes()` ........................................................ 54
- `byte()` ............................................................... 54
- `caseprefix()` ........................................................ 55
- `casesuffix()` ........................................................ 55
- `cat()` ................................................................. 56
- `csv_close()` ....................................................... 57
- `csv_getcols()` .................................................... 57
- `csv_getrows()` .................................................... 58
- `csv_getval()` ..................................................... 58
- `csv_open()` ........................................................ 59
- `debase64()` ........................................................ 59
- `dehex()` .............................................................. 60
- `deutf8()` ............................................................ 60
- `dns_expand()` ..................................................... 61
- `dns_next()` ........................................................ 61
- `elem()` ............................................................... 62
- `errno()` .............................................................. 62
- `ethmac()` ............................................................ 63
- `filename()` .......................................................... 63
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>formattedtime()</td>
<td>64</td>
</tr>
<tr>
<td>hatoi()</td>
<td>65</td>
</tr>
<tr>
<td>host()</td>
<td>65</td>
</tr>
<tr>
<td>host6()</td>
<td>66</td>
</tr>
<tr>
<td>incksum()</td>
<td>66</td>
</tr>
<tr>
<td>index()</td>
<td>67</td>
</tr>
<tr>
<td>io_close()</td>
<td>67</td>
</tr>
<tr>
<td>io_delete()</td>
<td>68</td>
</tr>
<tr>
<td>io_open()</td>
<td>68</td>
</tr>
<tr>
<td>io_read()</td>
<td>68</td>
</tr>
<tr>
<td>io_readlines()</td>
<td>69</td>
</tr>
<tr>
<td>io_write()</td>
<td>69</td>
</tr>
<tr>
<td>kiss_compile()</td>
<td>70</td>
</tr>
<tr>
<td>kiss_pattern()</td>
<td>70</td>
</tr>
<tr>
<td>kiss_search()</td>
<td>70</td>
</tr>
<tr>
<td>le_long()</td>
<td>70</td>
</tr>
<tr>
<td>le_short()</td>
<td>71</td>
</tr>
<tr>
<td>le_ulong()</td>
<td>71</td>
</tr>
<tr>
<td>le_ushort()</td>
<td>72</td>
</tr>
<tr>
<td>listadd()</td>
<td>72</td>
</tr>
<tr>
<td>listcombine()</td>
<td>72</td>
</tr>
<tr>
<td>listglom()</td>
<td>73</td>
</tr>
<tr>
<td>listlen()</td>
<td>73</td>
</tr>
<tr>
<td>long()</td>
<td>73</td>
</tr>
<tr>
<td>lsort()</td>
<td>74</td>
</tr>
<tr>
<td>luniq()</td>
<td>74</td>
</tr>
<tr>
<td>match()</td>
<td>75</td>
</tr>
<tr>
<td>multi_compile()</td>
<td>75</td>
</tr>
<tr>
<td>multi_escape()</td>
<td>75</td>
</tr>
<tr>
<td>multi_pattern()</td>
<td>75</td>
</tr>
<tr>
<td>multi_search()</td>
<td>77</td>
</tr>
<tr>
<td>net()</td>
<td>74</td>
</tr>
<tr>
<td>net6()</td>
<td>79</td>
</tr>
<tr>
<td>pattern()</td>
<td>79</td>
</tr>
<tr>
<td>pred()</td>
<td>79</td>
</tr>
<tr>
<td>prefix()</td>
<td>80</td>
</tr>
<tr>
<td>prunearray()</td>
<td>80</td>
</tr>
<tr>
<td>regcomp()</td>
<td>80</td>
</tr>
<tr>
<td>regexec()</td>
<td>81</td>
</tr>
<tr>
<td>scope()</td>
<td>82</td>
</tr>
<tr>
<td>sed()</td>
<td>83</td>
</tr>
<tr>
<td>short()</td>
<td>83</td>
</tr>
<tr>
<td>split()</td>
<td>84</td>
</tr>
<tr>
<td>str()</td>
<td>84</td>
</tr>
<tr>
<td>strcasecmp()</td>
<td>85</td>
</tr>
<tr>
<td>strerror()</td>
<td>85</td>
</tr>
<tr>
<td>strerror()</td>
<td>86</td>
</tr>
<tr>
<td>strlen()</td>
<td>86</td>
</tr>
<tr>
<td>strstr()</td>
<td>86</td>
</tr>
<tr>
<td>strtype()</td>
<td>87</td>
</tr>
<tr>
<td>sublist()</td>
<td>87</td>
</tr>
<tr>
<td>substr()</td>
<td>88</td>
</tr>
<tr>
<td>succ()</td>
<td>89</td>
</tr>
<tr>
<td>suffix()</td>
<td>90</td>
</tr>
<tr>
<td>tolower()</td>
<td>90</td>
</tr>
<tr>
<td>toupper()</td>
<td>91</td>
</tr>
<tr>
<td>typeof()</td>
<td>91</td>
</tr>
<tr>
<td>ubyte()</td>
<td>92</td>
</tr>
<tr>
<td>ulong()</td>
<td>92</td>
</tr>
<tr>
<td>unpack()</td>
<td>93</td>
</tr>
</tbody>
</table>
Chapter 1
Developing Custom CPcode Functions

In the DLP environment, CPcode can be used for creating new custom DLP Data Types.

**Note** - Throughout this guide, “data types” has two meanings: CPcode data types are the supported data objects that can be used in scripts. For example: a CPcode data type is “integer”.

DLP data types are the representation of data sent by users, to identify messages that should be matched to the DLP policy. For example: a DLP Data Type is “Social Security Numbers”.

In This Chapter

- DLP CPcode Basics
- The **func** Statement
- Restriction on Executed CPcode Statements
- CPcode Scope
- Function Calls
- Using Built-in CPcode Functions
- Fastpaths

DLP CPcode Basics

Each file should contain one filter function: **filter run dlp_dt** which does not receive any variables. This is the main function that will be called each time the Data Type is checked. This function should return 1 if the Data Type is matched, and 0 if not.

The **func** Statement

The **func** statement declares the name and statements that form the body of a function.

In most cases, the function declaration contains multiple statements. Curly braces ([ ]) are used to make the function a block statement. Functions may take an arbitrary number of arguments (accessed using the $1, $2, $3, ... notation). Each argument may be any data type.

<table>
<thead>
<tr>
<th><strong>func</strong></th>
<th><strong>Details</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>func name {[statement]}</td>
</tr>
</tbody>
</table>

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name of the function</td>
</tr>
<tr>
<td>statement</td>
<td>one or more statements to be executed within the function</td>
</tr>
</tbody>
</table>
### Developing Custom CPcode Functions

<table>
<thead>
<tr>
<th>func</th>
<th>Details</th>
</tr>
</thead>
</table>

**Return Value**  
Use the return statement to return values from a function.  
Return values can be any type except function pointers.

**Example**

```cpcode
func counter
{
    if ($count)
        $count = $count + 1;
    else
        $count = 1;
}
```

CPcode functions are implicitly declared upon usage and may be called in a CPcode file before they are defined.

For example:

```cpcode
initialize();
func initialize
{
    ...
}
```

### Restriction on Executed CPcode Statements

The CPcode execution engine keeps a heuristic count of approximate CPcode statements executed. At the beginning of every function call, during the prolog and at the head of all loops, the count of the number of CPcode statements executed is increased and compared against the limit.

Functions that exceed the statement limit are interrupted. The system simulates an immediate return at the point of interruption. Interrupted functions are eligible to run again when their conditions are matched.

The current limits are approximately 5,000 "ticks" from the point of execution (the `run()` function).

### CPcode Scope

Scope refers to the availability of symbols and values within an execution context. Some languages (such as C) support three scopes: function local, file and global scope. CPcode supports multiple scopes. The variables and symbols from one scope are available to another scope by referencing the variable's containing scope. This is similar to providing a fully qualified path name in a file system, and in fact, directly applies to a hierarchical CPcode type/subtype. The scope name is determined when the sensor execution engine loads the CPcode. This name is derived from the CPcode file names.

For example, assume that:

- The protocol configuration file is `dlprules.cfg`
- The `dlprules` CPcode contains a CPcode file named `commands`
- The CPcode for this Protection Group is `commands.cpc`

The directory structure would look something like this:

```
Root dir
    +---- dlprules
        +---- commands.cpc
        +---- othercommands.cpc
```

Therefore, the scope name for variables and functions in this CPcode is `dlprules_commands` (which is the file name of the master CPcode, followed by an underscore, and the file name of the subfunction CPcode). The scope name for protocol-wide is simply the protocol file name. For example, the scope for the protocol-wide `$WORKSPACE/protocols/dlprules.cpc` file is `dlprules`. Multiple Protection Groups can have global
variables and functions with the same name. These functions and variables will not interfere with each other because they are in different scopes. For example, both the commands and othercommands files contain a function named process_request. However, these functions exist within different scopes, dlprules_commands and dlprules_othercommands, respectively. Their fully-scoped names are dlprules_commands:process_request and dlprules_othercommands:process_request.

When you refer to global variables or functions by name, you are implicitly using variables and functions within the same scope as the reference. You can call functions and access variables in different scopes by using the fully qualified function or variable name.

You can declare values and assign them only to variables and functions in the same scope. By default, all CPcode functions and variables you define are global and exist within a specific scope. If you need local variables, you must explicitly create them.

You can use the scope() built-in function (see "scope()" on page 83) to obtain scope name.

**Local Variables**

CPcode local variables are defined by how they are named. To create a local variable, prefix the variable name with a dollar sign ($). The initial value of a local variable is NULL.

A local variable has a local scope. It exists only inside the function where it is declared. For example, the variable $count could hold the number of iterations inside a function named read_data. Another function, named get_dataStream, could also use a variable named $count. The $count variables in different functions hold different values.

Local variables are stack-allocated and are not persistent between calls.

For example:

This example function declares the display_write function, which calls the our_display and our_write functions. The values of the parameters $1 and $2 are available as local variables inside the function, but they cannot be modified.

```cpcode
func display_write
{
    if ($1 == 1)
        $a = our_display($2);
    else
        $a = our_write($2);
    return $a * 2 + 1;
}
```

**Function Calls**

To invoke a declared function, create a statement that includes the function name, followed by the parameters enclosed in parentheses.

For example:

```cpcode
show_service (80, "http");
```

To call a function and ignore the return value: func_name();

Example:

```cpcode
# In the following example, the expression statement calls
# the function count_hosts and ignores the returned value
count_hosts();
# The following statement passes the current line to
# the function for further processing.
another_scope:process_line("data\n");
```
Using Built-in CPcode Functions

Built-in CPcode functions use syntax conventions.

<table>
<thead>
<tr>
<th>Built-in CPcode Type</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Data Types</td>
<td>Functions that can take multiple data types for one parameter.</td>
<td><code>{array, int, list}</code> where any of the data types in the curly braces is valid</td>
</tr>
<tr>
<td>Optional Parameters</td>
<td>Functions that use optional parameters.</td>
<td><code>[int a, int b, string str]</code> where the parameters in the brackets are optional</td>
</tr>
<tr>
<td>Ellipsis</td>
<td>Functions that can take an indeterminate number of parameters.</td>
<td><code>listadd(list li, any item, [...])</code> where the ellipsis (...) can arbitrarily match many arguments</td>
</tr>
<tr>
<td>Default Parameter Values</td>
<td>Functions that use default parameter values. If a specific argument is not passed to one of these functions, the default value is assumed.</td>
<td><code>align(int num, [int hash :=4]);</code> where the default value appears after the := symbol (for example, 4)</td>
</tr>
</tbody>
</table>

Fastpaths

Several built-in CPcode functions have pre-optimized fastpaths they can follow if their most relevant parameter is the same variable as their return variable.

For example:

```plaintext
string = dehex(string);
```

A performance boost is gained by working on the parameter in place without allocating a temporary variable as the return variable. This gain can be significant, especially for the complex data types (such as array, list, and string). Functions that do not return variables of a complex data type would not gain anything by using a fastpath, because their return variables are cheap to allocate. Functions that have pre-optimized fastpaths have that ability noted in their description.
Chapter 2

Optimizing CPcode

The CPcode Compiler optimizes the performance of the CPcode system. You can increase CPcode performance by taking advantage of certain features of the language and avoiding others. On the other hand, CPcode may have an adverse performance impact if you are not careful.

In general: avoid using too many string operations in one Data Type script, and avoid creating long loops.

In This Chapter

Optimizing Variables.............. 12
Optimizing Arrays.................. 13
Optimizing Function Calls......... 14

Optimizing Variables

The CPcode Compiler is optimized to move as much of the run-time type checking into a compile-time type checker. All CPcode variables are cross-referenced with each statement that might assign its value to the variable in question. After all CPcode packages are compiled into an intermediate form, several passes of type inference are performed to determine the types of all variables concretely.

Polymorphic Variables

If the type of a variable cannot be resolved by the type inference process, that variable is called a polymorph. With polymorphs, the engine is forced to use runtime type checking, which is more expensive. To avoid this, assign only one type of data to a variable.

For example, if you initially assign a string to the variable and later assign an integer to the same variable, the variable will be tagged as polymorphic. If one variable is tagged as polymorphic, variables that depend on it will likely also be tagged polymorphic.

Network Variables

Do not mix IP addresses and networks inside a single variable. This will cause the variable to become a polymorph.

The CPcode Compiler tracks IPv4 addresses, IPv4 networks, IPv6 addresses, and IPv6 networks as separate types. However, it condenses them during code generation, because the execution engine considers them to be a unified type.

Deleting Variables

Each variable is either global or local. You may delete a variable by assigning a NULL to it (such as foo = NULL). This reclaims the memory that the variable occupies. You must nullify global variables when they become irrelevant to reclaim their memory.

You do not need to explicitly nullify local variables, unless you intend to reuse or reassign them. The CPcode Compiler will automatically nullify local variables when they fall out of scope, and will handle their destruction more efficiently than could be done with CPcode statements.
 Variables Or Literals

In CPcode, there is no performance penalty in using a variable over a constant literal. In this example, both statements take the same code path, and with the exception of cache and paging, they take the same amount of time:

$foo = 1; $bar = $foo;
$foo = ($bar = 1);

Two exceptions to this rule are constant folding and conditionals.

- **Constant folding** - The CPcode Compiler will attempt to evaluate operations on integer literals at compile time. Therefore, '1 + 1' translates to '2', and in the example:
  
  ```
  FOO = 1;
  BAR = 1;
  BLETCH = FOO + BAR;
  ```

- **Conditionals** - The literal will be evaluated during the compilation process, so the condition will be removed and only one code block will be compiled. For example:
  
  ```
  func absindex {
      if (feature.offsetindex) {
          return offsetindex($1,$2,$3);
      }
      else {
          $i = index(substr($1,$3),$2);
          if ($i == -1 || $i == NULL) {
              return -1;
          }
          else {
              return $i + $3;
          }
      }
  }
  ```

Optimizing Arrays

These guidelines will help you write CPcode scripts that use arrays and are optimized for performance.

Array Copies

Avoid writing CPcode that could result in a duplicate array. Copying an array can consume unnecessary CPU time and memory.

To avoid duplicate arrays:

- Do not use arrays or lists as a key to an array.
- Do not use arrays or lists as a value in an array.
- Do not use a polymorphic variable to contain an array.

Index Usage

Choose a simple data type to use as an array index. The index governs the speed of array operations.

- Use a simple data type, such as an integer, for best performance.
- Using blobs of the same size gives the worst performance. Blob comparison first checks the lengths of the blobs. If they are of different lengths, the comparison will be faster than if the blobs are of identical length, which means that their buffers must be walked.
Complex Indexes

It is common practice to track the number of times a TCP or UDP port-tuple has caused an alert, while sending the alert only once. If IP addresses are not of interest, it is better to combine the two 16-bit ports into a single integer than to use the blobbytes() function to create a blob. For example:

\[
\text{$key = ($src\_port \ll 16) | $dst\_port;} \\
\text{is better than:} \\
\text{$key = \text{blobbytes}(\text{$src\_port, $dst\_port});}
\]

Optimizing Function Calls

If used carefully, CPcode functions will simplify the writing of quality CPcode. If used carelessly, performance will suffer or the CPcode may fail to compile.

To ensure that functions do not adversely affect performance:

- Do not write recursive CPcode functions unless necessary.
- Do not use polymorphic variables in a function call.
Chapter 3
CPcode Lexicon

In This Chapter

<table>
<thead>
<tr>
<th>Character Sets</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Operators</td>
<td>15</td>
</tr>
<tr>
<td>Compound Operators</td>
<td>16</td>
</tr>
<tr>
<td>Naming Variables</td>
<td>16</td>
</tr>
<tr>
<td>Special Local Variables</td>
<td>17</td>
</tr>
<tr>
<td>NULL</td>
<td>17</td>
</tr>
<tr>
<td>Reserved Words</td>
<td>18</td>
</tr>
<tr>
<td>Constants</td>
<td>19</td>
</tr>
<tr>
<td>Configuring Constants</td>
<td>20</td>
</tr>
</tbody>
</table>

Character Sets

<table>
<thead>
<tr>
<th>Character Set</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII Alphanumeric</td>
<td>CPcode supports all ASCII alphanumeric characters.</td>
</tr>
<tr>
<td>Line Continuation</td>
<td>CPcode is not a line-oriented language. It does not have line continuation characters. Where line breaks occur, they must use UNIX-style newline characters, not the Windows-style carriage return and linefeed characters.</td>
</tr>
<tr>
<td>White Space</td>
<td>You can use spaces and horizontal tabs around elements to increase readability and to follow formatting conventions. Comments are also processed as white space.</td>
</tr>
<tr>
<td>Note: You must use spaces on either side of the &quot;inside&quot; operator.</td>
<td></td>
</tr>
<tr>
<td>Comment</td>
<td>A hash mark (#) indicates the beginning of a comment. The comment includes everything between the hash mark and the end of the line.</td>
</tr>
</tbody>
</table>

Simple Operators

Simple operators are one character.

<table>
<thead>
<tr>
<th>Character</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
</tbody>
</table>
### Character Definitions

<table>
<thead>
<tr>
<th>Character</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td>-</td>
<td>negation</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>bitwise NOT</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>!</td>
<td>logical NOT</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
</tbody>
</table>

### Compound Operators

Compound operators are two or more characters. The "inside" operator searches a list for specified values and returns a Boolean.

<table>
<thead>
<tr>
<th>Characters</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>?:</td>
<td>conditional</td>
</tr>
</tbody>
</table>

### Naming Variables

- Variable names can contain only ASCII alphanumeric characters and underscores.
- Variable names must begin with an alphabetic character.
- Variable names are not case sensitive.
• Variable names are limited to 128 characters in length.
• Local variables must be defined using the dollar sign character ($) as the first character of their name.
• You cannot use reserved words as variable names. However, variable names can contain reserved words. For example, if_available is a valid variable name even though if is a reserved word.

Special Local Variables

• $# - the number of parameters passed to a function or filter
• $n - nth parameter ($1 is the first parameter; $2 is the second parameter, and so on)

You cannot assign values to these special local variables, and they do not contain the same value across invocations.

For example:

```cpp
# This example makes use of the $# variable
func foo
{
    sum = bar(1, 2, 3);
    # "There were three parameters to bar 1 2 3"
    # sum == 6
    sum = bar(4, 5);
    # "There were two parameters to bar 4 5"
    # sum == 9
    sum = bar();
    # "Incorrect call to bar()"
    # sum == NULL
}

func bar
{
    if ($# == 3) {
        print("There were three parameters to bar", $1, $2, $3);
        return $1 + $2 + $3;
    } else if ($# == 2) {
        print("There were two parameters to bar", $1, $2);
        return $1 + $2;
    } else {
        print("Incorrect call to bar()");
        return NULL;
    }
}
```

NULL

NULL is a special variable that is a member of every data type.

You can compare expressions against NULL.

For example:

```cpp
# Is $foo an error? (Is it invalid?)
if ($foo == NULL)

# Is $foo not an error? (Is it valid?)
if ($foo != NULL)
```
# Is this division in error
if ( ($foo = $a / $b) == NULL )

NULL is also used to clear variables and free their memory.

For example:

```
func clean_up {
    COUNT = NULL;
}
func count_connections {
    if (COUNT[$1] == NULL) {
        COUNT[$1] = 1;
    } else {
        if (COUNT[$1] > CONNECTION_THRESHOLD) {
            complain($1)
        }
    }
}
```

Most built-in functions will return a NULL when there is a problem with an input parameter. In some instances, the CPcode Compiler will detect the error and reject the statement.

For example:

```
# Most Built-in functions can return a NULL if they don't like the parameters. The compiler will actually reject this statement because it knows it is an error.
if (($length = listlen(42)) == NULL)
```

If you set a variable or any array element to NULL, that variable or array element will be discarded.

For example:

```
# Make the variable $foo invalid
$foo = NULL;
# Make the variable in array at index '1' invalid
$array[1] = NULL;
```

NULLS are also commutative.

For example:

```
$a = NULL;
# $b becomes invalid (and the equivalent of NULL)
$b = $a + 1;
```

### Reserved Words

Do not use CPcode statement names or built-in function names as variable names.

In addition, you cannot use these words as variable names:

- call
- continue
- filter
- func
- global
The CPcode Compiler evaluates operations on integer constants at compile time. Therefore, '1 + 1' translates to '2', and this example:

```plaintext
if (1) {
    foo();
} else {
    bar();
}
```

translates to `foo();`

<table>
<thead>
<tr>
<th>Constant Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>+ - 0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>Prefix 0x and: 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f</td>
</tr>
<tr>
<td>IPv4host</td>
<td>Dotted quad format, 0 to 255. Example: 10.0.0.1</td>
</tr>
<tr>
<td>IPv4net</td>
<td>IPv4net constants in dotted quad (may use netmask notation) or Classless Inter-Domain Routing (CIDR) format. The CIDR element numbers are 0 to 32. Example: 10.0.0.0:255.255.255.0 (dotted quad with netmask) 10.0.0.0/24 (CIDR)</td>
</tr>
<tr>
<td>IPv6host</td>
<td>Hexadecimal, 0 to F. Example: EDAF:AAEE:0000:0053:FEF8:0192:0000:0023 (equals EDAF:AAEE::53:FEF8:192:0::23)</td>
</tr>
<tr>
<td>IPv6net</td>
<td>IPv6/NUM notation, where NUM is 1 to 128. Example: ::/64 ::1/128</td>
</tr>
<tr>
<td>MAC</td>
<td>MAC address, hexadecimal, 00 to ff, each octet must contain two digits. Example: 00:80:00:EA:8C:3A</td>
</tr>
<tr>
<td>String</td>
<td>Sequence of characters enclosed in double or single quotes. Can contain any single-byte character. If the string crosses lines, it will contain the newline character.</td>
</tr>
</tbody>
</table>
In a string constant, you can use special characters with the backslash to create a new meaning.

<table>
<thead>
<tr>
<th>Escape Character</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\n</code></td>
<td>newline</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>carriage return</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>tab</td>
</tr>
<tr>
<td><code>\xnn</code></td>
<td>arbitrary byte value, where nn is any hex digit</td>
</tr>
<tr>
<td><code>\</code></td>
<td>backslash</td>
</tr>
<tr>
<td><code>\0</code></td>
<td>NULL byte</td>
</tr>
<tr>
<td><code>\&quot;</code></td>
<td>double quote required when using a double quote inside a string enclosed with double quotes (for example, &quot;The &quot;Daily Report&quot;)</td>
</tr>
<tr>
<td><code>\'</code></td>
<td>single quote required when using a single quote inside a string enclosed with single quotes (for example 'yesterday's data')</td>
</tr>
</tbody>
</table>

**Configuring Constants**

You can create a *values file* to configure specified values for a constant. The values file name must match the CPcode source file name, except for the file extension.

For example:

- **CPcode file** - time_travel.cpc
- **Values file** - time_travel.values

Content of time_travel.cpc:

```c
filter run dlp_dt( ) {
    echo("value of DELOREAN_SPEED is ", DELOREAN_SPEED);
}
```

Content of time_travel.values:

```c
name DELOREAN_SPEED
text Speed
desc The speed the car needs to get to, desc in order to time travel.
desc mode scalar
88
```

**Note** - When you upload a CPcode file to SmartDashboard you must also upload the values file.
Chapter 4

CPcode Data Types

This chapter details the types of data that CPcode supports - not to be confused with the Data Types that you are creating for DLP policies.

Parameters passed to, or returned from, CPcode functions and filters must be of a valid CPcode data type. Most built-in CPcode functions return NULL when there is a problem with an input parameter. NULL is a special variable that is a member of every data type.

In This Chapter

array
ethmac
int
ipv4host and ipv6host
ipv4net and ipv6net
list
pattern
str
tag

array

<table>
<thead>
<tr>
<th>array</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>ethmac</td>
<td>22</td>
</tr>
<tr>
<td>int</td>
<td>22</td>
</tr>
<tr>
<td>ipv4host and ipv6host</td>
<td>23</td>
</tr>
<tr>
<td>ipv4net and ipv6net</td>
<td>23</td>
</tr>
<tr>
<td>list</td>
<td>24</td>
</tr>
<tr>
<td>pattern</td>
<td>24</td>
</tr>
<tr>
<td>str</td>
<td>24</td>
</tr>
<tr>
<td>tag</td>
<td>25</td>
</tr>
</tbody>
</table>

Description
A one-dimensional array of values

Syntax
array_name[element]

Range
The range for each element depends on the data type of that element.
For example: b = a["hello"];

Comments
- The index into the array can be any valid CPcode data type.
- Multiple data types can be used as an index into the same array.
- Each value in an array can be any valid CPcode data type.
- Data types within an array can be mixed.

To delete an element from an array, set the element to NULL.
To delete an array, set the array to NULL.
For example: $array = NULL;
### array

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| # define constants to use in emulating a structure
# with an array:
USERNAME = 1;
AUTHENTICATED = 2;
STATE = 3;
LINE = 4;

# build up the structure:
...
$cstructure[USERNAME] = $username;
$cstructure[AUTHENTICATED] = 0;
$cstructure[STATE] = STATE_PENDING_AUTH;
...
if ($cstructure[STATE] < STATE_MSG_DONE) {
    # Buffer up the current message for something
    # XXX Should be careful about the length
    $cstructure[MSG] = cat($cstructure[MSG], "new data")
} else {
    # Try each virus signature
    foreach $virus_sig inside (VIRUS_REGEXPES) {
        if (regexec($virus_sig, $cstructure[MSG])) {
            do_alert($cstructure);
            break;
        }
    }
    # Free the current message.
    $cstructure[MSG] = NULL;
} |

### ethmac

<table>
<thead>
<tr>
<th>ethmac</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>An Ethernet MAC address.</td>
</tr>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>Six octets of hexadecimal values separated by colons.</td>
</tr>
<tr>
<td>Range</td>
</tr>
<tr>
<td>00:00:00:00:00:00 to ff:ff:ff:ff:ff</td>
</tr>
<tr>
<td>Example</td>
</tr>
</tbody>
</table>
| # Correct usage
our_mac_address = 00:60:08:31:14:b7;

# Incorrect usage
our_mac_address = 0:60:8:31:14:b7; |

### int

<table>
<thead>
<tr>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>An integer.</td>
</tr>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>A decimal or hexadecimal representation written with a prefix of 0x.</td>
</tr>
<tr>
<td>Range</td>
</tr>
<tr>
<td>(-2^31) to (2^31 - 1) or -2,147,483,648 to +2,147,483,647</td>
</tr>
</tbody>
</table>
### int

**Comments**
- Octets with a value less than 0x10 must start with a leading zero.
- `int` is a signed data type, even when taken as a result of the `ulong` built-in function.

**Example**

| 2001 | 0xAA |

### ipv4host and ipv6host

#### ipv4host

**Description**  
A host IP address in IPv4 format.

**Syntax**  
Four octets of decimal values separated by period (a dotted quad)

**Range**  
0.0.0.0 - 255.255.255.255

**Example**

```plaintext
our_router = 208.244.85.1;
```

#### ipv6host

**Description**  
A host IP address in IPv6 format.

Refer to RFCs 2460 and 2373 for details about ipv6 address formats.

### ipv4net and ipv6net

#### ipv4net

**Description**  
A network IP address in IPv4 or CIDR format.

**Syntax**  
A dotted quad defining the network number, followed by a colon, followed by a dotted quad describing the netmask;

Or:
A dotted quad defining the network number, followed by a slash, followed by the number of 1 bits in the netmask (CIDR syntax).

**Range**  
0.0.0.0:0.0.0.0 - 255.255.255.255:255.255.255.255

or

0.0.0.0/0 - 255.255.255.255/32

**Example**

```plaintext
our_network = 208.244.85.0:255.255.255.0;
our_network = 208.244.85.0/24;
```
### ipv4net

| Description | A network IP address in IPv4 format. Refer to RFCs 2460 and 2373 for details about ipv6 address formats. |

### list

| Description | An ordered list of values. |
| Syntax | [element, element] |
| Range | The range for each element depends on the data type of that element. Data types may be mixed within a list. While a list can contain any number of elements, it is best to keep the list short. |
| Comments | Spaces are not allowed. |
| Example | interesting_ports = [21, 23, 80, 443, 2001];
  a = [1, 2, 3];
  $b=1;
  a = [$b, $b+$b, 4-$b]; |

### pattern

| Description | A regular expression returned by the regcomp() or pattern() functions, that can be either passed to the regexec() function or directly compared against a string |

### str

| Description | An arbitrary sequence of bytes. |
| Syntax | • Text, enclosed within single or double quotes.
  • Hex numbers for each byte, separated by a backslash, enclosed within single or double quotes |
## str

<table>
<thead>
<tr>
<th>Range</th>
<th>&quot;&quot; (empty string) to no explicit limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The str data type is not a C-style NULL-terminated string. It may contain binary data including any number of NULL characters.</td>
</tr>
<tr>
<td></td>
<td>The following C-style special characters can be used when creating a str data type:</td>
</tr>
<tr>
<td></td>
<td>• \n - newline</td>
</tr>
<tr>
<td></td>
<td>• \r - carriage return</td>
</tr>
<tr>
<td></td>
<td>• \t - tab</td>
</tr>
<tr>
<td></td>
<td>• \ - backslash</td>
</tr>
<tr>
<td></td>
<td>• \xnn - an arbitrary byte value for this character, where nn is any hex digit</td>
</tr>
<tr>
<td>Comments</td>
<td>Only the CPcode application engine can generate a pattern data type. You cannot define it as a constant literal when developing CPcode or build it directly out of literals.</td>
</tr>
</tbody>
</table>
| Example | ```
foo = "string";
foo = "\x73\x74\x72\x69\x6e\x67";
``` |

## tag

<table>
<thead>
<tr>
<th>Description</th>
<th>Tags are descriptors for triggered filters, which you can use to turn off the filter during later execution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>There is an implicit parameter to filters - $1$ - which is the filter tag for the filter (&quot;Special Local Variables&quot; on page 17).</td>
</tr>
</tbody>
</table>
Chapter 5

CPcode Operators

CPcode operators determine how CPcode operates on one or more expressions within a statement.

In This Chapter

Arithmetic Precedence  26
Arithmetic Operators  27
Relational Operators  29
Logical Operators  34
Bitwise Operators  35
Other Operators  38

Arithmetic Precedence

<table>
<thead>
<tr>
<th>Order</th>
<th>Oper.</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[ ]</td>
<td></td>
</tr>
<tr>
<td>2nd</td>
<td>!</td>
<td>logical NOT</td>
</tr>
<tr>
<td></td>
<td>~</td>
<td>bitwise NOT</td>
</tr>
<tr>
<td>3rd</td>
<td>inside</td>
<td>Boolean, dependent upon whether an item is inside a list</td>
</tr>
<tr>
<td>4th</td>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td>5th</td>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>6th</td>
<td>&lt;&lt;</td>
<td>left bit shift</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td>right bit shift</td>
</tr>
<tr>
<td>7th</td>
<td>&lt;</td>
<td>relational less than</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>relational less than or equal to</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>relational greater than</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>relational greater than or equal to</td>
</tr>
<tr>
<td>8th</td>
<td>==</td>
<td>relational equal to</td>
</tr>
</tbody>
</table>
## CPcode Operators

<table>
<thead>
<tr>
<th>Order</th>
<th>Oper.</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>relational greater than or equal to</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bitwise OR</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>conditional</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>parameter or list delimiter</td>
<td></td>
</tr>
</tbody>
</table>

### Arithmetic Operators

In this section

```
+ (addition)  27
- (subtraction)  27
* (multiplication)  28
/ (division)  28
% (modulus)  28
- (negation)  29
```

#### + (addition)

<table>
<thead>
<tr>
<th>+</th>
<th>addition</th>
</tr>
</thead>
</table>

**Description**
The addition operator adds expression2 to expression1.

**Syntax**

```
int expression1 + int expression2
```

**Operands**

- `int expression1` - an expression
- `int expression2` - an expression

**Return Value**

```
int - the sum of the two expressions
```

**Example**

```
# This example adds 1 to the variable $count
$count = $count + 1;
```

#### - (subtraction)

<table>
<thead>
<tr>
<th>-</th>
<th>subtraction</th>
</tr>
</thead>
</table>

**Description**
The subtraction operator subtracts expression2 from expression1.

**Syntax**

```
int expression1 - int expression2
```
- **subtraction**

<table>
<thead>
<tr>
<th>Operands</th>
<th>int expression1 - an expression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int expression2 - an expression</td>
</tr>
<tr>
<td>Return Value</td>
<td>int - the difference between the two expressions</td>
</tr>
<tr>
<td>Example</td>
<td># Subtract 1 from the $time_left variable $time_left = $time_left - 1;</td>
</tr>
</tbody>
</table>

* (multiplication)

* **multiplication**

<table>
<thead>
<tr>
<th>Description</th>
<th>The multiplication operator multiplies expression1 by expression2.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>int expression1 * int expression2</td>
</tr>
<tr>
<td>Operands</td>
<td>int expression1 - an expression</td>
</tr>
<tr>
<td></td>
<td>int expression2 - an expression</td>
</tr>
<tr>
<td>Return Value</td>
<td>int - the product of the two expressions</td>
</tr>
</tbody>
</table>
| Example     | # Multiply number of minutes by 60 
# to determine number of seconds 
seconds = minutes * 60; |

/ (division)

/ **division**

<table>
<thead>
<tr>
<th>Description</th>
<th>The division operator divides expression1 by expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>int expression1 / int expression2</td>
</tr>
<tr>
<td>Operands</td>
<td>int expression1 - an expression</td>
</tr>
<tr>
<td></td>
<td>int expression2 - an expression</td>
</tr>
<tr>
<td>Return Value</td>
<td>int - the quotient</td>
</tr>
</tbody>
</table>
| Example     | # Divide number of seconds by 60 
# to determine the # number of minutes 
minutes = seconds / 60; |

% (modulus)

% **modulus**

<table>
<thead>
<tr>
<th>Description</th>
<th>The modulus operator divides expression1 by expression2 and returns the remainder.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>int expression1 % int expression2</td>
</tr>
<tr>
<td>Operands</td>
<td>int expression1 - an expression</td>
</tr>
<tr>
<td></td>
<td>int expression2 - an expression</td>
</tr>
<tr>
<td>Return Value</td>
<td>int - the remainder</td>
</tr>
</tbody>
</table>
% modulus

Example # Use division and modulus by 60
    # to convert seconds to minutes and seconds
    seconds = 318;
    minutes = seconds / 60;
    seconds = seconds % 60;

- (negation)

Description The negation operator displays the negative value of an expression.

Syntax - int expression

Operands int expression - an expression

Return Value int - the negation of the expression

Example # Assign the negative of variable $b to variable $a
    $a = - $b;

Relational Operators

In this section

    == (equal to) 30
    != (not equal to) 31
    > (greater than) 32
    >= (greater than or equal to) 32
    < (less than) 33
    <= (less than or equal to) 33
== (equal to)

<table>
<thead>
<tr>
<th>==</th>
<th>equal to</th>
</tr>
</thead>
</table>

Description

The equal to operator compares `expression2` to `expression1` for equality, using different evaluation criteria for each data type.

- **array** - compares each index and the corresponding item
- **ethmac** - compares as a set of six octets. Each octet in `expression1` must be equal to the matching octet in `expression2`.
- **int** - compares by value
- **ipv4host** - compares as a set of four octets. Each octet in `expression1` must be equal to the matching octet in `expression2`.
- **ipv4net** - compares as a set of eight octets or a set of four octets and the netmask. Each octet in `expression1` must be equal to the matching octet in `expression2`.
- **ipv6host** - compares as a set of 16 octets. Each octet in `expression1` must be equal to the matching octet in `expression2`.
- **ipv6net** - compares as a set of 32 octets or a set of four octets and the netmask. Each octet in `expression1` must be equal to the matching octet in `expression2`.
- **list** - compares by each element of each list. Each element in `expression1` must be equal to the matching element in `expression2`.
- **pattern** - compares by pattern
- **str** - compares by value, with regard to case

You can compare expressions that have different data types, for example:

- **ipv4host** and **ipv4net** - if the host is in the network, the expressions are considered equal.
- **str** and **pattern** - if the string is the same as the pattern, the expressions are considered equal.

Syntax

```
expression1 == expression2
```

Operands

- `expression1` - an expression
- `expression2` - an expression

The following data types are valid for these operands:

- **array**
- **ethmac**
- **int**
- **ipv4host**
- **ipv4net**
- **ipv6host**
- **ipv6net**
- **list**
- **pattern**
- **str**

Return Value

- `1` - `expression1` is equal to `expression2`
- `0` - `expression1` is not equal to `expression2`
== | equal to
Example

```bash
# This example compares the subject to NULL, and if it is
# not NULL, then other processing occurs
if ($subject == NULL)
    $subject = $1;
else
    $subject = cat( $subject, $1);
return;
```

!= (not equal to)

| != | not equal to |
Description

The not equal operator compares expression2 to expression1 for inequality, using different evaluation criteria for each data type.

- array - compares each index and the corresponding item
- ethmac - compares as a set of six octets. Each octet in expression1 must be equal to the matching octet in expression2
- int - compares by value
- ipv4host - compares as a set of four octets. Each octet in expression1 must be equal to the matching octet in expression2.
- ipv4net - compares as a set of eight octets or a set of four octets and the netmask. Each octet in expression1 must be equal to the matching octet in expression2.
- ipv6host - compares as a set of 16 octets. Each octet in expression1 must be equal to the matching octet in expression2.
- ipv6net - compares as a set of 32 octets or a set of four octets and the netmask. Each octet in expression1 must be equal to the matching octet in expression2.
- list - compares by each element of each list. Each element in expression1 must be equal to the matching element in expression2
- pattern - compares by pattern
- str - compares by value, with regard to case

You can compare expressions that have different data types, for example:

- ipv4host and ipv4net - if the host is not in the network, the expressions are not equal.
- str and pattern - if the string is not the same as the pattern, the expressions are not equal.

Syntax

```
expression1 != expression2
```
### CPcode Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>not equal to</td>
</tr>
</tbody>
</table>

#### Operands
- expression1 - an expression
- expression2 - an expression

The following data types are valid for these expressions:
- array
- ethmac
- int
- ipv4host
- ipv4net
- ipv6host
- ipv4net
- list
- pattern
- str

#### Return Value
- 1 - expression1 is not equal to expression2
- 0 - expression1 is equal to expression2

#### Example
```
# Compare value in $s_nevermind to 1
if ($s_nevermind != 1)
    webdet_server_process($v);
```

### > (greater than)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
</tbody>
</table>

#### Syntax
```
int expression1 > int expression2
```

#### Operands
- int expression1 - an integer expression
- int expression2 - an integer expression

#### Return Value
- 0 - expression1 is not greater than expression2
- 1 - expression1 is greater than expression2

#### Example
```
# Compare string length of variable $blob to 2048
if (strlen($blob) > 2048)
    $blob = "";
```

### >= (greater than or equal to)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

#### Syntax
```
int expression1 >= int expression2
```

#### Operands
- int expression1 - an integer expression
- int expression2 - an integer expression

#### Return Value
- 0 - expression1 is not greater than or equal to expression2
- 1 - expression1 is greater than or equal to expression2

#### Example
```
# Compare string length of variable $blob to 2048
if (strlen($blob) >= 2048)
    $blob = "";
```
### CPcode Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
</tbody>
</table>

#### >= (Greater Than or Equal To)

<table>
<thead>
<tr>
<th>Operands</th>
<th>Return Value</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int expression1 - an integer expression&lt;br&gt;int expression2 - an integer expression</td>
<td>0 - expression1 is not greater than or equal to expression2&lt;br&gt;1 - expression1 is greater than or equal to expression2</td>
<td># Compare return value of the byte function to 0&lt;br&gt;if (byte ($sblob, bloblimit) &gt;= 0) $state = -1; &quot;&quot;;</td>
</tr>
</tbody>
</table>

#### < (Less Than)

<table>
<thead>
<tr>
<th>Operands</th>
<th>Return Value</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int expression1 - an integer expression&lt;br&gt;int expression2 - an integer expression</td>
<td>0 - expression1 is not less than expression2&lt;br&gt;1 - expression1 is less than expression2</td>
<td># Compare value in $state variable to 0&lt;br&gt;if ($state &lt; 0) return;</td>
</tr>
</tbody>
</table>

#### <= (Less Than or Equal To)

<table>
<thead>
<tr>
<th>Operands</th>
<th>Return Value</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int expression1 - an integer expression&lt;br&gt;int expression2 - an integer expression</td>
<td>0 - expression1 is not less than or equal to expression2&lt;br&gt;1 - expression1 is less than or equal to expression2</td>
<td># Compare the string length of the variable $cmd to 0&lt;br&gt;if (strlen ($cmd) &lt;= 0)</td>
</tr>
</tbody>
</table>
Logical Operators

In this section

&& (logical AND)  34
|| (logical OR)    34
! (logical NOT)   35

&& (logical AND)

<table>
<thead>
<tr>
<th>&amp;&amp;</th>
<th>logical AND</th>
</tr>
</thead>
</table>

**Description**
The logical AND operator evaluates both sides of the expression. If both expressions
evaluate to true (they are non-zero and non-NULL), the result is true. If either, or both,
expression evaluates to false, the result is false.

The CPcode logical AND operator, like the C language && operator, is a short-circuit
operator. If the first expression evaluates to false, the second expression is not
evaluated.

**Syntax**
expression1 && expression2

**Operands**
- expression1 - an expression
- expression2 - an expression

**Return Value**
- 0 - the expression is false
- 1 - the expression is true

**Example**
if (ALERT_ON_ROOT && index($data, "root") >= 0) {
    alert(...);
}

# If ALERT_ON_ROOT is set to non-zero, index() will be called.
# If ALERT_ON_ROOT is set to zero, the value of the conditional
# will be false and index() will not be called.
# Compiler generates code similar to:
# if (ALERT_ON_ROOT) {
#    if (index($data, "root") >= 0) {
#      alert(...);
#    }
# }
#

|| (logical OR)

| || | logical OR |
|----|-------------|

**Description**
The logical OR operator evaluates both sides of the expression. If either or both
expressions evaluates to true (non-zero and non-NULL), the result is true.

This operator is a short circuit operator, like the || operator in the C language. If the first
expression evaluates as true, the second expression is not evaluated.

**Syntax**
expression1 || expression2
### Logical OR

| Operands  | expression1 - an expression  
| Operands  | expression2 - an expression  
| Return Value | 0 - the expression is false  
| Return Value | 1 - the expression is true  
| Example | # because of short circuiting, strlen() will not be called in this  
| Example | # example  
| Example | $string = "foo";  
| Example | if ($string || strlen($string) < 20)  

### Logical NOT

| Logical NOT | !  
| Operands | expression - an expression  
| Return Value | 0 - the expression is not zero  
| Return Value | 1 - the expression is zero  
| Example | $string = "foo";  
| Example | if (! $string || strlen(string) < 20)  

### Bitwise Operators

In this section

- & (bitwise AND)  
- | (bitwise OR)  
- ~ (bitwise NOT)  
- ^ (bitwise XOR)  
- << (left shift)  
- >> (right shift)
### & (bitwise AND)

<table>
<thead>
<tr>
<th>&amp;</th>
<th>bitwise AND</th>
</tr>
</thead>
</table>

**Description**  
The bitwise AND operator evaluates identically positioned bits in both expressions. If both bits are 1, the operator sets the corresponding bit in the result to 1. If either bit is 0, the operator sets the corresponding bit in the result to 0. The following table summarizes this behavior:

<table>
<thead>
<tr>
<th>If bit in expression1 is</th>
<th>and bit in expression2 is</th>
<th>then bit in result will be</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Syntax**  
`int expression1 & int expression2`

**Operands**  
- `int expression1` - an expression  
- `int expression2` - an expression

**Return Value**  
`int` - the bitwise evaluation of both expressions

**Example**  
```c
# Bitwise AND on the value in the variable foo
# and the number 1
if ((foo & 1) == 1)
    echo ("foo is odd\n");
```

### | (bitwise OR)

<table>
<thead>
<tr>
<th></th>
<th>bitwise OR</th>
</tr>
</thead>
</table>

**Description**  
The bitwise OR operator evaluates identically positioned bits in both expressions. If both bits are 0, the operator sets the corresponding bit in the result to 0. If either bit is 1, the operator sets the corresponding bit in the result to 1.

**Syntax**  
`int expression1 | int expression2`

**Operands**  
- `int expression1` - an expression  
- `int expression2` - an expression

**Return Value**  
`int` - the bitwise evaluation of the two expressions

**Example**  
```c
# This example performs a bitwise OR on the values in the
# variables a and b, and displays the result of 7
a = 5;
b = 3;
c = a | b;
 echo ("c is ", c, "\n");
```
### ~ (bitwise NOT)

<table>
<thead>
<tr>
<th>~</th>
<th>bitwise NOT</th>
</tr>
</thead>
</table>

**Description**  
The bitwise NOT operator inverts the bit values in an expression. If the bit is 0, the operator sets the corresponding bit in the result to 1. If the bit is 1, the operator sets the corresponding bit in the result to 0. The following table summarizes this behavior:

<table>
<thead>
<tr>
<th>If bit in expression is</th>
<th>then bit in result will be</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Syntax**

```
~ int expression
```

**Operands**

- int expression - an expression

**Return Value**

- int - the bitwise evaluation of the expression

**Example**

```
# Do a bitwise NOT on the variable a where a = ~ a;
# this example clears the least significant bit in the variable b
b = b & ~ 1;
```

### ^ (bitwise XOR)

<table>
<thead>
<tr>
<th>^</th>
<th>bitwise XOR</th>
</tr>
</thead>
</table>

**Description**  
The bitwise XOR operator evaluates identically positioned bits in both expressions. If both bits are the same, the operator sets the corresponding bit in the result to 0. If the bits are different, the operator sets the corresponding bit in the result to 1. The following table summarizes this behavior:

<table>
<thead>
<tr>
<th>If bit in expression1 is</th>
<th>and bit in expression2 is</th>
<th>then bit in result will be</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Syntax**

```
int expression1 ^ int expression2
```

**Operands**

- int expression1 - an expression
- int expression2 - an expression

**Return Value**

- int - the bitwise evaluation of the two expressions

**Example**

```
# Do a bitwise XOR on the low 4 bits of the value in
# the variable $a
$a = $a ^ 0xf;
```
<< (left shift)

<table>
<thead>
<tr>
<th>&lt;&lt;</th>
<th>left shift</th>
</tr>
</thead>
</table>

**Description**
The left shift operator shifts all of the bits in expression1 to the left by expression2 bits.

**Syntax**

```
int expression1 << int expression2
```

**Operands**
- `int expression1 << int expression2`

**Return Value**
`int` - the result of the bitwise shift

**Example**

```
# This example determines whether the Nth bit is set
$tmp = 1 << Nth;
if ($1 & $tmp)
    echo ("Nth bit set\n");
```

>> (right shift)

<table>
<thead>
<tr>
<th>&gt;&gt;</th>
<th>right shift</th>
</tr>
</thead>
</table>

**Description**
The right shift operator shifts all of the bits in expression1 to the right by expression2 bits. This is a logical shift, (for example, `0x80000000 >> 1 == 0x40000000`).

**Syntax**

```
int expression1 >> int expression2
```

**Operands**
- `int expression1` - the integer to be shifted
- `int expression2` - an integer representing the number of bits by which expression1 should be shifted

**Return Value**
`int` - the result of the bitwise shift

**Example**

```
# Find the version of the IP packet payload contained in the
# variable $1
$version = ubyte($1,0) >> 4;
```

Other Operators

**In this section**

- `= (assignment)`
- `?: (conditional)`

= (assignment)

<table>
<thead>
<tr>
<th>=</th>
<th>assignment</th>
</tr>
</thead>
</table>

**Description**
The assignment operator sets variable equal to expression. You can use assignments within expressions (often referred to as built-in assignments).

**Syntax**

```
variable = expression
```
### CPcode Operators

#### Assignment

<table>
<thead>
<tr>
<th>=</th>
<th>assignment</th>
</tr>
</thead>
</table>

**Operands**
- variable - a global variable, local variable, or array element
- expression - an expression. All data types are valid.

**Return Value**
variable - the value assigned by the expression

**Note**
Built-in assignments must be enclosed inside parentheses, for example:

```bash
# this is correct:
$foo = ($bar = NULL);
```

**Example**

# Use the assignment operator to set variables to different data types
$services[ "telnet" ] = 1;
router_mac_address = 00:c0:7b:6b:ee:2b;
$count = 318830;
router_IP_address = 208.244.85.1;
our_network = 208.244.85.0/255.255.255.0;
our_network = 208.244.85.0/24;
web_ports = [ 80, 8000, 8080, 2001 ];
company_name = "Check Point";

# The value of the Built-in assignment expression will be the left hand side of the assignment after it is assigned.
$foo = 1;
$bar = ($foo = $foo + 1);

# The value of $bar is 2
# The value of $foo is also obviously 2

---

### Inside

<table>
<thead>
<tr>
<th>inside</th>
<th>special operator</th>
</tr>
</thead>
</table>

**Description**
The inside operator searches for expression1 in the list represented by expression2. The result is true if expression1 is an element of the list represented by expression2.

**Syntax**

```
expression1 inside expression2
```

There must be a space on either side of the inside operator.

Do not confuse the inside operator, whose role is to return a Boolean value, with "inside" used in the syntax of a foreach or declare statement.

**Operands**
- expression1 - an expression. Valid data types are any that can be added to a list.
- expression2 - a list

**Return Value**
- 0 - expression1 is not found inside expression2
- 1 - expression1 is found inside expression2
inside | special operator
---|---
**Example**

# Assume $path contains extracted path element from a data source to test against.
# This example looks at $path and tests it against elements in a list variable, and if a match is found, sets a local variable to 1.

SPOOLER_WINDOWS_ROOT_PATHS=
"c:\winnt","c:\windows";
$bad = 0;

foreach $path_var inside (SPOOLER_WINDOWS_ROOT_PATHS) {
    if (cat($path_var,"\system32") == $path) {
        $bad = 1;
        break;
    }
}

?: (conditional)

<table>
<thead>
<tr>
<th>?:</th>
<th>conditional</th>
</tr>
</thead>
</table>

**Description**
The conditional operator evaluates an expression for a condition. It functions as an IF-THEN-ELSE shortcut.

**Syntax**
variable = expression ? value1 : value2

**Operands**
- expression - the expression to be evaluated
- value1 - the expression that will be assigned to ret_value if expression is true
- value2 - the expression that will be assigned to ret_value if expression is false

**Return Value**
variable - the value returned from the evaluation, which is value1 if expression is true or value2 if expression is false

**Example**

# A counter that will forgive you for starting at NULL
# `'$counter = $counter + 1;'`
# will not work unless $counter initialized to be something other than NULL at some point
$counter = $counter ? $counter + 1 : 1;

# To limit $foo to numbers less than or equal to 100
$foo = $bar < 100 ? $bar : 100;
Chapter 6

CPcode Statements

This section lists valid CPcode statements.

If you are familiar with the C language, you should exercise caution when writing CPcode. While CPcode statements appear to be C-like, there are differences. CPcode may not support some C language statements.

In This Chapter

= (assignment) 41
{} (block) 41
break 42
foreach 42
func 43
if 44
requires 44
return 45
while 45

= (assignment)

| = | assignment |
---|-------------|

**Description**  An assignment statement assigns the value on the right-hand side of the equal sign (=) to the variable on the left-hand side. You can use the assignment statement to set the value of a variable.

```c
variable = expression;
```

**Syntax**  

**Parameters**  

- variable - the variable to which the expression will be assigned
- expression - the expression to be assigned to the variable

**Example**  

```c
# Use the assignment statement to set the value
# of our_router to 192.168.10.1.
our_router = 192.168.10.1;
```

{} (block)

| {} | block |
---|------|

**Description**  Block notation groups together multiple CPcode statements. The block starts with the left curly bracket and ends with the right curly bracket. All statements inside these brackets are part of the block.

```c
{statement_list}
```

**Syntax**

**Parameters**  

statement_list - a list of CPcode statements
### block

**Example**

```cpp
# The block statement groups together several
# statements into a user-defined function
func counter
{
    return ($count = $count + 1);
}
```

### break

**Description**

The break statement ends execution of the innermost foreach or while statement in which it appears. You can use the break statement to exit from a loop.

**Syntax**

```cpp
break;
```

**Parameters**

N/A

**Example**

```cpp
# Go through the buffer line-by-line.
# If check_line finds what it is looking for,
# it will return 1, and we should exit
while (($eol = index($buf, "\n")) > -1) {
    $line = substr($buf, 0, $eol);
    $buf = substr($buf, $eol + 1);
    if (check_line($line)) {
        break;
    }
}
```

### foreach

**Description**

The foreach statement executes the same statements for each element of a list or indices of an array.

When used with a list, the foreach statement sets the variable to the elements of a list and executes the statements once for each element. The sensor execution engine processes the elements in the order in which they appear in the list.

When used with an array, the foreach statement sets a variable to the index value of the array and executes the statements once for each index value. The order in which the index values appear is not defined.

**Syntax**

```cpp
foreach $variable inside [list|array] { statements }
```

**Parameters**

- `$variable` - a local variable
- `list` - a variable, constant, or expression that evaluates to a list
- `array` - a variable, constant, or expression that evaluates to an array
- `statements` - statements to be executed for each element in the list
**foreach**

**Notes**

Do not modify a list or array inside the loop. While the CPcode Compiler normally prevents this, it could cause stability issues with the gateway or appliance. For example, do not code:

```cpcode
foreach $foo inside array {
    if (array[$foo] > 1)
        array[$foo] = NULL;
}
foreach $foo inside list {
    list = listadd(list, $foo * 2);
}
```

**Example**

# Do a set of actions for each recipient listed in a mail message

```cpcode
foreach $tmp inside ($to_list) {
    if (mail_list:present)
        mail_list:rec($start_time, $src_addr, $fromaddr, $dst_addr, $tmp, $subject);
    if (mail_od:present)
        mail_od:rec($start_time, $src_addr, $fromaddr, $dst_addr, $tmp, $subject);
    if (mail_tricks:present)
        mail_tricks:rec($start_time, $src_addr, $fromaddr, $dst_addr, $tmp, $subject);
}
```

# Do a set of actions for each element in an array.

```cpcode
$services["telnet"] = 1;
$services["http"] = 2;
$services["ssl"] = 3;
foreach $tmp inside ($services){
    echo( $tmp, " is element ", $services[$tmp], "\n" );
}
```

This function prints:

telnet is element 1
http is element 2
ssl is element 3

**func**

**Description**

The `func` statement declares the name and statements that form the body of a function. In most cases, the function declaration contains multiple statements.

**Syntax**

```cpcode
func name { [statements] }
```

**Parameters**

- name - the name of the function
- statement - one or more statements to be executed within the function

**Example**

```cpcode
func counter
{
    return $1 + 1;
}
```
### if

<table>
<thead>
<tr>
<th>if</th>
</tr>
</thead>
</table>

**Description**  The if statement evaluates an expression, and if the expression evaluates to true, executes a statement. If the expression evaluates to false, the statement is not executed. If the value of the expression evaluates to false and there is an else clause, the else clause statements are executed.

**Syntax**
- if (expression) statement
- if (expression) statement else statement

**Parameters**
- expression - the expression to be evaluated
- statement - the statement to be executed if the expression is true

Parentheses are required around the expression parameter.
If the expression being evaluated is complex (for example, contains && or ||), the evaluation will terminate at the earliest opportunity.

**Notes**
In the following example, the if statement is used to determine if there is any data in the $blob variable. If the $blob variable does not contain any data, the statement assigns data to it. If the $blob variable does contain data, the statement after the else clause adds to the data already in the variable.

```cpp
if ($blob == NULL)
    $blob = $my_data;
else
    $blob = cat ( $blob, $my_data);
```

**Example**

```cpp
# some_function will never be called because its
# results will not affect the value of the expression.
$false = 0;
$true = 1;
if ($false && some_function() != NULL {                
}    
if ($true || some_function() != NULL {            
}    
```

### requires

<table>
<thead>
<tr>
<th>requires</th>
</tr>
</thead>
</table>

**Description**  The "requires" statement specifies variables that must be set. If any variables are NULL, the rest of the CPcode is not loaded.

You can use the "requires" statement to control whether a protocol or Protection Group is run. If a required variable is not set, the Protection Group or protocol is not loaded.

**Syntax**
requires variable1 variable2 variable3;

**Parameters**
variable_ - one or more variables that must contain non-NULL values before the remaining CPcode is loaded.
### requires

**Example**

```csh
# The requires statement specifies that the value for
# the ourdomainname variable must be set for the protocol to
# load.

requires ourdomainname;
```

### return

**Description**
The return statement exits a function and, optionally, returns a value.

**Syntax**
```csh
return;

return [expression];
```

**Parameters**
- expression - any expression

**Example**

```csh
# The return statement exits the if statement and
# returns the value in the $line variable.

$xxx = index($line, ">");
if ($xxx < 0)
    return $line;
```

### while

**Description**
The while statement evaluates an expression and repeatedly executes the statement as long as the expression remains true, as long as it is non-zero and non-NULL.

**Syntax**
```csh
while (expression) statement;
```

**Parameters**
- expression - expression to be evaluated
- statement - the statement to be executed

**Example**

```csh
The while statement walks along the contents of $cblob. It starts at whatever value is in
$n when the while statement is entered. As long as the $nth byte of $cblob is greater
than zero, it keeps going. When it sees a byte that is less than or equal to zero, it stops.

while (byte($cblob,$n) > 0) {
    $n = $n + 1;
}
```
Chapter 7

CPcode Built-in Variables

This section lists the packet variables of interest to CPcode developers. Built-in variables gather data from the packet just received and provide easy access to information about the connection stream.

In This Chapter

System Variables 46
Feature Variables 46
Global Variables 47

System Variables

System variables extract data from the system running a Check Point appliance or gateway.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>system.maxticks</td>
<td>Returns the maximum number of ticks that can be used.</td>
<td>int</td>
</tr>
<tr>
<td>system.ticks</td>
<td>Returns the current number of ticks that the CPcode has used out of its allotted amount.</td>
<td>int</td>
</tr>
<tr>
<td>system.time</td>
<td>Returns the system time, in seconds, since the beginning of the epoch (Jan 1 1970 UTC).</td>
<td>int</td>
</tr>
</tbody>
</table>

Feature Variables

Feature variables provide access to supported execution engine features and revision information (such as when a built-in CPcode function was changed).

Feature variables are evaluated at compile time. For example:

```cpp
if (feature.filename >= 20030412) {
    //Supports new Microsoft OS's filename scheme
    $file = filename($line);
} else {
    //Do it in CPcode
    file = filename_ms($line);
}
```

translates at compile to either:

```cpp
$file = filename($line);
```

or
```
$file = filename_ms($line);
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature.&lt;function_name&gt;</td>
<td>Information about when the built-in CPcode function specified by <code>&lt;function_name&gt;</code> was last changed. <code>&lt;function_name&gt;</code> must be the name of a valid, built-in function.</td>
<td>int</td>
</tr>
</tbody>
</table>

## Global Variables

DLP functions in CPcode can use any of the CPcode functions and variables that are not traffic related (notice that UDP and TCP variables are not supported).

In addition, there are DLP specific variables.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy.internal_domains</td>
<td>List of domains (strings), that come from &quot;My Organization&quot; object in the policy.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wind</td>
</tr>
<tr>
<td>message.protocol</td>
<td>IP protocol.</td>
<td>&quot;SMTP&quot; &quot;FTP&quot; &quot;HTTP&quot;</td>
</tr>
<tr>
<td>message.source</td>
<td>The source of the message.</td>
<td><a href="mailto:dan@domain.com">dan@domain.com</a> &quot;190.1.1.1&quot;</td>
</tr>
<tr>
<td>message.source_username</td>
<td>The username of the message sender or an empty string if the sender's email is not mapped to a username.</td>
<td>&quot;dan&quot;</td>
</tr>
<tr>
<td>message.dests</td>
<td>List of destinations.</td>
<td>In SMTP: [&quot;<a href="mailto:sam@domain.com">sam@domain.com</a>&quot;,&quot;<a href="mailto:joe@domain.com">joe@domain.com</a>&quot;] In FTP: [&quot;154.1.0.0&quot;,&quot;154.1.1.0&quot;]</td>
</tr>
<tr>
<td>message.dests_username</td>
<td>The username of the message's destination or an empty string if the recipient's email is not mapped to a username.</td>
<td>[&quot;sam&quot;, &quot;joe&quot;]</td>
</tr>
<tr>
<td>message.mail_subject</td>
<td>For SMTP - contains the mail subject (in the recipient's part). For HTTP/FTP - empty.</td>
<td>&quot;RE: New List&quot;</td>
</tr>
<tr>
<td>message.header_names</td>
<td>Contains protocol header fields.</td>
<td>MIME headers</td>
</tr>
<tr>
<td>message.header_values</td>
<td>Contains protocol header values.</td>
<td>MIME headers</td>
</tr>
<tr>
<td>message.num_of_files</td>
<td>For SMTP - Contains the number of attachments For HTTP/FTP - Contains the number of files</td>
<td>3 - 3 attachments 3 - 3 files</td>
</tr>
<tr>
<td>message.uid</td>
<td>ID of the message</td>
<td>&quot;[5483F78E-D5AF-8E60-CB28-283968ED426A]&quot;</td>
</tr>
<tr>
<td>message.mail_body</td>
<td>Contains the text of the mail body. Relevant only for SMTP messages.</td>
<td>&quot;Hello my name is Alice … Regards, Alice&quot;</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>part.content</td>
<td>A buffer containing the text of a part. It is either the text of a mail's body, or text of a file (attached to mail, uploaded in http or ftp).</td>
<td>&quot;This is the body.... Regards &quot;</td>
</tr>
<tr>
<td>part.id</td>
<td>Running index stating the part number in a file (starting from 0)</td>
<td>0 1 2</td>
</tr>
<tr>
<td>part.type</td>
<td>Describes the type of the part.</td>
<td>&quot;Recipients&quot; &quot;Content&quot;</td>
</tr>
<tr>
<td>part.file_type_id</td>
<td>An identifier for file type. See “Defining New Files” in the DLP Administration Guide for the full list of different types.</td>
<td>11 for “SmartWare” 311 for “Quattro Pro Win 12.0” 169 for “Java Script”</td>
</tr>
<tr>
<td>part.file_type_name</td>
<td>File type description. See “Defining New Files” in the DLP Administration Guide for the full list of different types.</td>
<td>“Word for DOS 5.x” “OfficeWrite”</td>
</tr>
<tr>
<td>part.file_family_type</td>
<td>The main family/category type of the file.</td>
<td>“Presentation” “Word Processor”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The complete list is in Note 3 below.</td>
</tr>
<tr>
<td>part.name</td>
<td>The name of the attachment, or the name of the file uploaded.</td>
<td>&quot;body&quot; &quot;file.doc&quot;</td>
</tr>
<tr>
<td>part.is_binary</td>
<td>This variable equals 1 when the part.content is the binary representation of the file. If the variable is 0, part.content contains the textual representation of the file.</td>
<td>1 0</td>
</tr>
<tr>
<td>part.is_file</td>
<td>This variable equals 1 when the scanned part is an attachment, otherwise it equals 0.</td>
<td>Relevant only for SMTP.</td>
</tr>
<tr>
<td>part.is_corrupted</td>
<td>This variable equals 1 when the scanned part is considered corrupted. Files are considered corrupted files when DLP cannot extract their content. This variable is relevant to CPcode that runs when the Data Type is matched (CPcode defined in Advanced Matching).</td>
<td>1 - File is corrupted 0 - File is not corrupted</td>
</tr>
<tr>
<td>part.sub_filenames</td>
<td>This field is a list of the names of the sub-files.</td>
<td>Images in Word Document: [&quot;object001&quot;,&quot;object002&quot;,&quot;object3&quot;] Files in an archive: [&quot;notepad.exe&quot;,&quot;mydoc.doc&quot;,&quot;demo.tar&quot;]</td>
</tr>
<tr>
<td>part.sub_family_types</td>
<td>This field is a list of the main family names of the sub-files.</td>
<td>[&quot;Images&quot;,&quot;Images&quot;,&quot;Images&quot;]</td>
</tr>
<tr>
<td>part.sub_file_size</td>
<td>This field is a list of the file sizes in bytes of the sub-files.</td>
<td>[156, 41234, 36552 ]</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>verify.str</td>
<td>A string that was matched to a pattern (The pattern was defined in a data type).</td>
<td>&quot;0404787875&quot;</td>
</tr>
<tr>
<td>verify.index</td>
<td>An index in CONTENT variable to the end of the matched string.</td>
<td>50</td>
</tr>
</tbody>
</table>

**Notes:**

1. Variables marked with a 1 require the existence of one of these:
   - A Cooperate Template Data Type in the DLP policy Rule Base with the **consider template images** feature enabled.
   - A **File Marking** rule in the DLP policy Rule Base.
   - **always_extract_embedded_images** field in dlp.conf is set to 1.
2. For all **part.sub** variables:
   - A file can contain several sub-files. All sub variables refer to these sub-files:
     - Embedded images in office document.
     - Files listed in archives.
   - There might be cases where names are automatically generated.
   - Elements in the lists **part.sub_filenames**, **part.sub_family_types** and **part.sub_file_size** correspond to each other.
   - The fields are relevant only when **part.type** equals "Content".
   - Sub-files of Archives are relevant only for CPcode that runs when the Data Type is matched (CPcode defined in Advanced Matching).
3. For **part.file_family_type**, the family can be one of these:
   - "None"
   - "Word Processor"
   - "Spreadsheet"
   - "Presentation"
   - "Viewer"
   - "Text"
   - "Markup"
   - "Multimedia"
   - "Email"
   - "Images"
   - "Executable"
   - "Archive"
   - "Database"
   - "V-Calendar"
   - "Drawings"
   - "Graphic Design"
   - "JPEG"
   - "GIF"
   - "TIF"
   - "ZIP"
   - "RAR"
   - "TAR"
   - "PDF"
   - "Visio"
   - "Professional Images"
   - "Microsoft Office files"
   - "Unrecognized"
   - "Unsupported Archive"
   - "Other"

**Example with Global Variables**

This script shows a CPcode Data Type that is turned on when there are more than 2 external domains in an Email message.
filter run dlp_dt() {
    
    # This data type is active for SMTP only, for the recipient part
    if (message.protocol !="SMTP" || part.type !="Recipients")
        return 0;
    $num_of_external_domains = 0;
    $external_domain_list = [];

    foreach $recipient inside message.dests {
        
        # get the domain of the mail address
        $domain = global:get_domain_from_address($recipient);

        # check if the domain is external
        if (!global:is_internal_domain($domain)) {
            if (!global:is_member($external_domain_list,$domain)) {
                # Add it to the external domains list
                $external_domain_list = listadd($external_domain_list,$domain);
                # Check if we reached the max of external domains
                $num_of_external_domains = $num_of_external_domains + 1;
                if ($num_of_external_domains >= MAX_EXTERNAL_DOMAINS)
                    return 1;
            }
        }
    }
    return 0;
}

Recommendations with Global Variables

CPcode Data Types, just like any other Data Type, run on every part of a message. If an email is scanned, the CPcode function will be called several times: once for the Recipients part, then for each attachment, and the body of the email.

Include the following code at the beginning of your filter. If you want the CPcode Data Type to run only once for each email, allow it to run only in the "recipients part":

if (message.protocol !="SMTP" || part.type !="Recipients")
    return 0;
Chapter 8

CPcode Built-in Functions

In This Chapter

align() 53
arrayindices() 53
atoi() 54
blobbytes() 54
byte() 55
caseprefix() 55
casesuffix() 56
cat() 56
csv_close() 57
csv_getcols() 57
csv_getrows() 58
csv_getval() 58
csv_open() 59
dehex64() 59
dehex() 60
deufl8() 60
dns_expand() 61
dns_next() 61
elem() 62
erro() 62
ethmac() 63
filename() 63
formattedtime() 64
hatoi() 65
host() 65
host6() 66
incksum() 66
index() 67
io_close() 67
io_delete() 68
io_open() 68
io_read() 68
io_readlines() 69
io_write() 69
kiss_compile() 70
kiss_pattern() 70
kiss_search() 70
le_long() 70
le_short() 71
le_ulong() 71
le_ushort() 72
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>listadd()</td>
<td>72</td>
</tr>
<tr>
<td>listcombine()</td>
<td>72</td>
</tr>
<tr>
<td>listglom()</td>
<td>73</td>
</tr>
<tr>
<td>listlen()</td>
<td>73</td>
</tr>
<tr>
<td>long()</td>
<td>73</td>
</tr>
<tr>
<td>lsort()</td>
<td>74</td>
</tr>
<tr>
<td>luniq()</td>
<td>74</td>
</tr>
<tr>
<td>match()</td>
<td>75</td>
</tr>
<tr>
<td>multi_compile()</td>
<td>75</td>
</tr>
<tr>
<td>multi_escape()</td>
<td>75</td>
</tr>
<tr>
<td>multi_pattern()</td>
<td>75</td>
</tr>
<tr>
<td>multi_search()</td>
<td>77</td>
</tr>
<tr>
<td>net()</td>
<td>78</td>
</tr>
<tr>
<td>net6()</td>
<td>79</td>
</tr>
<tr>
<td>pattern()</td>
<td>79</td>
</tr>
<tr>
<td>pred()</td>
<td>79</td>
</tr>
<tr>
<td>prefix()</td>
<td>80</td>
</tr>
<tr>
<td>prunearray()</td>
<td>80</td>
</tr>
<tr>
<td>regcomp()</td>
<td>81</td>
</tr>
<tr>
<td>regexec()</td>
<td>82</td>
</tr>
<tr>
<td>scope()</td>
<td>83</td>
</tr>
<tr>
<td>sed()</td>
<td>83</td>
</tr>
<tr>
<td>short()</td>
<td>84</td>
</tr>
<tr>
<td>split()</td>
<td>84</td>
</tr>
<tr>
<td>str()</td>
<td>85</td>
</tr>
<tr>
<td>strcasecmp()</td>
<td>85</td>
</tr>
<tr>
<td>strerror()</td>
<td>86</td>
</tr>
<tr>
<td>strlen()</td>
<td>86</td>
</tr>
<tr>
<td>strstr()</td>
<td>86</td>
</tr>
<tr>
<td>strtype()</td>
<td>87</td>
</tr>
<tr>
<td>sublist()</td>
<td>88</td>
</tr>
<tr>
<td>substr()</td>
<td>89</td>
</tr>
<tr>
<td>succ()</td>
<td>89</td>
</tr>
<tr>
<td>suffix()</td>
<td>90</td>
</tr>
<tr>
<td>tolower()</td>
<td>90</td>
</tr>
<tr>
<td>toupper()</td>
<td>91</td>
</tr>
<tr>
<td>typeof()</td>
<td>91</td>
</tr>
<tr>
<td>ubyte()</td>
<td>92</td>
</tr>
<tr>
<td>ulong()</td>
<td>92</td>
</tr>
<tr>
<td>unpack()</td>
<td>93</td>
</tr>
<tr>
<td>url_flush()</td>
<td>94</td>
</tr>
<tr>
<td>url_needs_refresh()</td>
<td>94</td>
</tr>
<tr>
<td>url_open()</td>
<td>95</td>
</tr>
<tr>
<td>url_readall()</td>
<td>95</td>
</tr>
<tr>
<td>url_readlines()</td>
<td>95</td>
</tr>
<tr>
<td>ushort()</td>
<td>96</td>
</tr>
</tbody>
</table>
align()

The `align()` function bit aligns data items that are larger than one byte (for example, a record where a four-byte integer only occurs at an address that is a multiple of four bytes).

| align() | align(int num, [int hash :=4]); |
| Syntax | |
| Parameters | Parameter | Description |
| | int num | The integer to align. |
| | int hash | The single bit bitmask to align on. |
| Return Value | int - The aligned integer. |

Example

```
align(0) == 0
align(1) == 4
align(2) == 4
align(3) == 4
align(4) == 4
align(5) == 8
align(0,2) == 0
align(1,2) == 2
align(2,2) == 2
align(3,2) == 4
align(4,2) == 4
```

arrayindices()

The `arrayindices()` function creates a sorted list of an array's associative indices.

| arrayindices() | arrayindices(array arr); |
| Syntax | |
| Parameters | Parameter | Description |
| | array arr | The array from which to create the list. |
| Return Value | list - a list of all the array's indices |

It is illegal to modify an array being iterated through using a foreach statement. Therefore, it is necessary to first create a list of all the array's indices, and then use foreach to iterate through that list.
arrayindices()

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| array[0] = "zero";
array[1] = "one";
foreach $iterator inside (arrayindices(array)) {
    echo("array[", $iterator, "] = ", array[$iterator], "\n");
    # Output:
    # array[0] = zero
    # array[1] = one
    # The next statement would be illegal if iterating
    # over the array instead of a copy of its indices.
    array[$iterator] = NULL;
}

atoi()

The atoi() function converts a string to an integer representation. If you specify an index, the conversion begins at that location inside the string.

The atoi() function ignores any white space leading up to the number.

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>atoi(string str, [int index := 0]);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string str</td>
<td>The string that is to be converted to an integer.</td>
</tr>
<tr>
<td>int index</td>
<td>The point in the string at which to begin the conversion.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>• int - the integer value of the string, str</td>
</tr>
<tr>
<td>• 0 - the string is empty, does not contain digits, or contains non-digits before the first digit in the string</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>atoi(&quot;192&quot;) == 192</td>
</tr>
<tr>
<td>atoi(&quot; 192 &quot;) == 192</td>
</tr>
<tr>
<td>atoi(&quot;1 9 2 &quot;, 4) == 2</td>
</tr>
<tr>
<td>atoi(&quot;192ABC&quot;) == 192</td>
</tr>
<tr>
<td>atoi(&quot;ABC&quot;) == 0</td>
</tr>
<tr>
<td>atoi(&quot;ABC192&quot;, 3) == 192</td>
</tr>
<tr>
<td>atoi(&quot;&quot;&quot;) == 0</td>
</tr>
<tr>
<td>atoi(42) == error</td>
</tr>
</tbody>
</table>

blobbytes()

The blobbytes() function creates a blob from the byte values of an expression. Integers are an exception - their lowest significant bytes are used. For example, if the integer is 0x23, only one byte is used; if the integer is 0x1234, two bytes are used; etc.

The cat() function should, preferably, be used when concatenating strings.
The `blobbytes()` function has been optimized to generate a blob from non-strings, and is usually used to generate a hash index for an array.

### blobbytes()

**Syntax**

```
blobbytes(...);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>The values to be added to the blob/string.</td>
</tr>
</tbody>
</table>

**Return Value**

- **string** - a blob/string
  
  The overall return string for the `blobbytes()` function cannot be larger than 8KB.

**Example**

```plaintext
# Get first 4 bytes of a data packet
# but swap them around
$a = udp.blob;
$b = blobbytes(substr(a, 3, 1), substr($a, 2, 1),
               substr($a, 1, 1), substr($a, 0, 1));
```

---

### byte()

The `byte()` function returns a byte from a specified offset in a string. The returned value is signed. You can use the `ubyte()` function to return an unsigned value.

The `byte()` function is useful for determining if all of the desired fields have been gathered (for example, to determine if the byte at offset 40 in a particular string contains the expected value).

**Syntax**

```
byte(string str, int offset);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string str</td>
<td>The string to be examined. The string begins at byte 0.</td>
</tr>
<tr>
<td>int offset</td>
<td>The offset location in the string for the byte to be returned.</td>
</tr>
</tbody>
</table>

**Return Value**

- **int** - the offset in the string as a signed value
- **error** - the offset was out of bounds

**Example**

```plaintext
# the eighth byte is the signed type code
$code = byte(udp.blob, 7);
```

---

### caseprefix()

The `caseprefix()` function compares two strings and returns true if the second string is a prefix of the first string, without regard to case.

Unlike the `index()` and `prefix()` functions, the `caseprefix()` function performs a case-insensitive comparison of the second argument against the beginning of the first argument.

You can use the `prefix()` function for a case-sensitive comparison.
### caseprefix()

The `caseprefix()` function compares two strings and returns true if the second string is a suffix of the first string, without regard to case.

You can use the `suffix()` function for a case-sensitive comparison.

<table>
<thead>
<tr>
<th>caseprefix()</th>
<th>caseprefix(string big, string little);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>caseprefix(string big, string little);</td>
</tr>
<tr>
<td>Parameters</td>
<td>Parameter</td>
</tr>
<tr>
<td></td>
<td>string big</td>
</tr>
<tr>
<td></td>
<td>string little</td>
</tr>
<tr>
<td>Return Value</td>
<td>1 - little is the same as the end of big, regardless of case</td>
</tr>
<tr>
<td></td>
<td>0 - little is not the same as the end of big</td>
</tr>
<tr>
<td>Example</td>
<td>caseprefix(&quot;ABCDEF&quot;,&quot;EFG&quot;) == 1</td>
</tr>
<tr>
<td></td>
<td>caseprefix(&quot;ABCDEF&quot;,&quot;efg&quot;) == 1</td>
</tr>
<tr>
<td></td>
<td>caseprefix(&quot;ABCDEF&quot;,&quot;ABC&quot;) == 0</td>
</tr>
</tbody>
</table>

### casesuffix()

The `casesuffix()` function compares two strings and returns true if the second string is a suffix of the first string, without regard to case.

Unlike the `suffix()` function, the `casesuffix()` function performs a case-insensitive comparison of the second argument against the end of the first argument.

<table>
<thead>
<tr>
<th>casesuffix()</th>
<th>casesuffix(string big, string little);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>casesuffix(string big, string little);</td>
</tr>
<tr>
<td>Parameters</td>
<td>Parameter</td>
</tr>
<tr>
<td></td>
<td>string big</td>
</tr>
<tr>
<td></td>
<td>string little</td>
</tr>
<tr>
<td>Return Value</td>
<td>1 - little is the same as the end of big, regardless of case</td>
</tr>
<tr>
<td></td>
<td>0 - little is not the same as the end of big</td>
</tr>
<tr>
<td>Example</td>
<td>casesuffix(&quot;ABCDEF&quot;,&quot;EFG&quot;) == 1</td>
</tr>
<tr>
<td></td>
<td>casesuffix(&quot;ABCDEF&quot;,&quot;efg&quot;) == 1</td>
</tr>
<tr>
<td></td>
<td>casesuffix(&quot;ABCDEF&quot;,&quot;ABC&quot;) == 0</td>
</tr>
</tbody>
</table>

### cat()

The `cat()` function concatenates arguments by appending them into a single string. Non-string arguments are converted to their textual representation before concatenation. For list arguments, each element in the list is appended to the new string.

<table>
<thead>
<tr>
<th>cat()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>cat();</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>Return Value</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>cat(&quot;Hello&quot;, &quot;world&quot;), &quot;!&quot;);</td>
</tr>
<tr>
<td></td>
<td>cat(&quot;Hello&quot;, [&quot;world&quot;, &quot;!&quot;])</td>
</tr>
</tbody>
</table>
**cat()**

<table>
<thead>
<tr>
<th>Syntax</th>
<th><code>cat(...)</code>;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>… - the arguments to concatenate</td>
</tr>
<tr>
<td>Return Value</td>
<td>string - a blob/string</td>
</tr>
</tbody>
</table>

The `cat()` function will ignore NULL parameters. Therefore, a statement such as the following would be safe when `$blob` has not yet been initialized:

```plaintext
$blob = cat($blob, $data);
```

When using `cat()` to concatenate an array or a list with an array member, the textual representation of the array must not exceed 8KB.

**Example**

```plaintext
# Keep data around in a single blob
if ($blob == null)
    $blob = $newdata;
else
    $blob = cat($blob, $newdata);
```

**csv_close()**

The `csv_close()` function closes a CSV file previously opened with `csv_open()`.

<table>
<thead>
<tr>
<th>csv_close()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
<tr>
<td>Example</td>
</tr>
</tbody>
</table>

```plaintext
$fd = io_open("./myfile.csv");
$csvp = csv_open($fd);
csv_close($csvp);
```

**csv_getcols()**

The `csv_getcols()` function returns the number of columns in a particular row in a CSV file previously opened with `csv_open()`.

<table>
<thead>
<tr>
<th>csv_getcols()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
</tbody>
</table>
| Parameters          | • Int `csvp` – the file descriptor returned from `csv_open`.  
                      • Int `row` – which row to get columnar count |
| Return Value        | Int – number of columns in the specified row |
### csv_getcols()

**Example**

```php
$fd = io_open("./myfile.csv");
$csvp = csv_open($fd);
...
$numcols = csv_getcols($csvp,1);
echo("numcols ":$numcols");
csv_close($fd);
io_close($fd);
```

### csv_getrows()

The `csv_getrows()` function returns the number of rows in a particular row in a CSV file previously opened with `csv_open()`.

**Syntax**

```php
int csv_getrows(int cvsp);
```

**Parameters**

- `Int cvsp` – the file descriptor returned from `csv_open`.

**Return Value**

- `Int` – number of rows in the specified CSV file

**Example**

```php
$fd = io_open("./myfile.csv");
$csvp = csv_open($fd);
...
$numrows = csv_getrows($csvp);
$current_row = 0;
while( $current_row < $numrows) {
    echo("numcols ":csv_getcols($csvp,$current_row);
    $current_row = $current_row + 1;
}
csv_close($csvp);
io_close($fd);
```

### csv_getval()

The `csv_getval()` function returns the value of a cell in a specified row and column in a CSV file previously opened with `csv_open()`.

**Syntax**

```php
string csv_getval(int cvsp, int row, int col);
```

**Parameters**

- `Int cvsp` – the file descriptor returned from `csv_open`.
- `Int row` – row from which to retrieve value
- `Int col` – column from which to retrieve value

**Return Value**

- `String` – string-encoded return value
### csv_getrows()

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| $fd =   
  url_fetch("http://www.mysite.com/myfile.csv");  
$csvp = csv_open($fd);  
...$numrows = csv_getrows($csvp,1);  
$current_row = 0;  
while( $current_row < $numrows) {  
  echo("first value in row ",  
  $current_row, ", "
  csv_getval($csvp,$current_row,1));  
  $current_row = $current_row + 1;
}  
csv_close($csvp);  
io_close($fd); |

### csv_open()

The csv_open() function opens a CSV file.

<table>
<thead>
<tr>
<th>csv_open()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
</tbody>
</table>
| $fd =   
  url_fetch("http://www.mysite.com/myfile.csv");  
$csvp = csv_open($fd);  
...$numrows = csv_getrows($csvp,1);  
$current_row = 0;  
while( $current_row < $numrows) {  
  echo("first value in row ",  
  $current_row, ", "
  csv_getval($csvp,$current_row,1));  
  $current_row = $current_row + 1;
}  
csv_close($csvp);  
io_close($fd) |

### debase64()

The debase64() function decodes a base64-encoded string.

The debase64() function has a pre-optimized fastpath, where str is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.
The `debase64()` function may be impacted by the change in the behavior of string variables. See `str` for additional information about this behavior.

### debase64()

**Syntax**

```
debase64(string str);
```

**Parameters**

- `string str` - the string to decode

**Return Value**

- `string` - the decoded input string

The overall return string for the `debase64()` function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

**Example**

```bash
# Decode email attachment
$email_attachment = debase64($email_attachment);

# IMAP4 authentication challenges and responses
# are also base64 encoded
$challenge = debase64($challenge);
```

### dehex()

The `dehex()` function undoes the ASCII translation performed for HTTP requests. When sending special characters in a URL, the HTTP specification says to encode them as their hexadecimal values preceded by a percent sign (%). The `dehex()` function looks for such metacharacters and returns them to their original values.

The `dehex()` function has a pre-optimized fastpath, where `str` is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The `dehex()` function may be impacted by the change in the behavior of string variables ("str" on page 24).

**Syntax**

```
dehex(string str, [string delimiter]);
```

**Parameters**

- `string str` - a string that contains hex characters
- `string delimiter` - a single-character string used as the delimiter, with a default value of %.

**Return Value**

- `string` - a string that has had its characters changed back to the original values

The overall return string for the `dehex()` function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

**Example**

```
dehex("abcd%41%42%43%44") == "abcdABCD"
dehex("abcd%4a%4b%4c%4d") == "abcdJKLM"
dehex("abcd%zz") = "abcd%zz"
```

### deutf8()

The `deutf8()` function decodes UTF-8 encoding, first passing the input string through the `dehex()` function.

The `deutf8()` function has a pre-optimized fastpath, where `str` is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The `deutf8()` function may be impacted by the change in the behavior of string variables ("str" on page 24).
### deutf8()

**Syntax**

deutf8(string str);

**Parameters**

- string str - the string to decode

**Return Value**

- string - the decoded input string

The overall return string for the debase64() function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

**Example**

```plaintext
# Decode email attachment
$email_attachment = debase64($email_attachment);

# IMAP4 authentication challenges and responses
# are also base64 encoded
$challenge = debase64($challenge);
```

### dns_expand()

The `dns_expand()` function decodes a DNS-encoded string. It operates safely on recursive names and operates correctly on heavily nested names.

DNS names may be compressed and some needed information may reside at the beginning or end of the string. Therefore, it is important to pass the entire string into `dns_expand()`, not just the relevant substring.

**Syntax**

```plaintext
string dns_expand(string str, int offset);
```

**Parameters**

- string str - the string containing the encoded name
- int offset - the offset at which to start decoding the encoded name

**Return Value**

- string - the lowercase decoded name, or an error string (for example, "Recursive pointer" and "Bad pointer").

**Example**

```plaintext
# Decode a SOA record with 2 contiguous strings
# (name and admin)
$name = dns_expand($dns_blob, $offset);
if (!($next = dns_next($dns_blob, $offset))) return -1;

$admin = dns_expand($dns_blob, $next);
if (!($next = dns_next($dns_blob, $next))) return -1;

$offset = $next;
# $offset will point to the next resource record
```

### dns_next()

The `dns_next()` function finds the next DNS resource record in a string, after a decoded name. It returns the offset into the string of the next DNS RR or a NULL error indication if it could not be fully or correctly decoded.
### dns_next()

**Syntax**

```cpp
int dns_next(string str, int offset);
```

**Parameters**

- `string str` - the string containing multiple DNS resource records
- `int offset` - the offset at which to start decoding the current encoded name

**Return Value**

- `int` - the offset into the string of the next resource record
- `NULL` - the encoded name could not be fully or correctly decoded

**Example**

```cpp
# Decode SOA record which contains two contiguous strings (name and admin)
$name = dns_expand($dns_blob, $offset);
if (!($next = dns_next($dns_blob, $offset)))
  return -1;
$admin = dns_expand($dns_blob, $next);
if (!($next = dns_next($dns_blob, $next))) return -1;
$offset = $next;
# And $offset points to next resource record
```

### elem()

The **elem()** function extracts a single element from a list. If you do not specify an index, the function returns the first element in the list.

**Syntax**

```cpp
elem(list li, [int index := 0]);
```

**Parameters**

- `list li` - the list from which to extract the element
- `int index` - the index to the element you want to extract. The index value starts at zero (that is, an index value of 0 extracts the first element).

**Return Value**

- `any` - an element from the list.

Full run-time type checking is performed on the return value and any variables that depend on it.

**Example**

```cpp
# Extract the third element in the list
portlist = [21, 23, 53, 80];
third = elem(portlist, 2);
third == 53;
```

### errno()

The **errno()** function returns the Boolean value associated with an error.

**Syntax**

```cpp
errno(any argument);
```
### errno()

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any argument</td>
<td>the expression you want evaluated for error status</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int - a Boolean value corresponding to the</td>
<td>error status of the argument, where:</td>
</tr>
<tr>
<td>argument</td>
<td>1 - the argument evaluated to an error</td>
</tr>
<tr>
<td></td>
<td>0 - the argument did not evaluate to an error</td>
</tr>
</tbody>
</table>

The `errno()` function has changed significantly beginning with CPcode v3. It now returns a Boolean integer instead of a numeric value. Unlike past CPcode versions, the CPcode execution engine will stop if a catastrophic error is encountered.

**Example**

```plaintext
Print("Error status ", errno($foo + $bar));
```

### ethmac()

The `ethmac()` function constructs an Ethernet MAC address from the supplied arguments.

<table>
<thead>
<tr>
<th>ethmac()</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td><code>ethmac({string, int} a, [int b, int c, int d, int e, int f]);</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{string, int} a</td>
<td>either the first octet or a string with the textual representation of the MAC address</td>
</tr>
<tr>
<td>int b</td>
<td>the second octet</td>
</tr>
<tr>
<td>int c</td>
<td>the third octet</td>
</tr>
<tr>
<td>int d</td>
<td>the fourth octet</td>
</tr>
<tr>
<td>int e</td>
<td>the fifth octet</td>
</tr>
<tr>
<td>int f</td>
<td>the sixth octet</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ethmac</code> - the MAC address</td>
<td></td>
</tr>
</tbody>
</table>

**Example**

```plaintext
ethmac("00:00:ba:de:ca:fe") == 00:00:BA:DE:CA:FE  
ethmac(0, 0, 0xba, 0xde, 0xca, 0xfe) == 00:00:BA:DE:CA:FE  
```

### filename()

The `filename()` function will deobfuscate ("cleanse") a filename. It contracts ../ and . directory hiding and removes superfluous //.

The `filename()` function has a pre-optimized fastpath, where str is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The `filename()` function may obfuscate the actual occurrences when symbolic links are present. For example:

```plaintext
mkdir /tmp/link  
cd /tmp/link  
ln -s /usr foo  
filename("/tmp/link/foo/..") == "/tmp/link"  
```

but
The filename() function may be impacted by the change in the behavior of string variables. Refer to the description of the string data type on page B-5 for additional information about this behavior.

Depending on the protocol being decoded, it may be necessary to run the input filename through the deutf8() or dehex() functions before passing it through filename().

The overall return string for the filename() function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

<table>
<thead>
<tr>
<th><strong>filename()</strong></th>
<th>filename(string str, [int BackSlashFlag := 0]);</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>filename(string str, [int BackSlashFlag := 0]);</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>• string str - the original filename</td>
</tr>
<tr>
<td></td>
<td>• int BackSlashFlag - Boolean used to treat the filename as a Microsoft Windows filename</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>string - a cleansed filename</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>filename(&quot;/usr/../.etc/passwd&quot;) == &quot;/etc/passwd&quot;</td>
</tr>
<tr>
<td></td>
<td>filename(&quot;../etc/../passwd&quot;) == &quot;../etc/passwd&quot;</td>
</tr>
<tr>
<td></td>
<td>filename(&quot;/etc/foo..bar&quot;) == &quot;/etc/foo..bar&quot;</td>
</tr>
<tr>
<td></td>
<td>filename(&quot;/etc/././foo&quot;) == &quot;/etc/foo&quot;</td>
</tr>
<tr>
<td></td>
<td>filename(&quot;/etc/.../foo&quot;) == &quot;/etc/.../foo&quot;</td>
</tr>
</tbody>
</table>

The formattedtime() function returns the time of day as a formatted string.

The formattedtime() function may be impacted by the change in the behavior of string variables. Refer to the description of the string data type on page B-5 for additional information about this behavior.

The formattedtime() function is extremely slow and should be used judiciously.

The overall return string for the formattedtime() function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

<table>
<thead>
<tr>
<th><strong>formattedtime()</strong></th>
<th>formattedtime(int time,[string format]);</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>formattedtime(int time,[string format]);</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>• int time - the time to be formatted</td>
</tr>
<tr>
<td></td>
<td>• string format - the format to use for displaying the time. Refer to the strftime in UNIX man pages for formatting options. If a format is not specified, the following default format is used:</td>
</tr>
<tr>
<td></td>
<td>• [%m%d%y %H:%M:%S]</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>string - a string containing the time in the format specified</td>
</tr>
</tbody>
</table>
### formattedtime()

**Example**

```plaintext
time = formattedtime(system.time);
timeonly = formattedtime(system.time, "%T %Z");
dateonly = formattedtime(system.time, "%A, %d %B %Y");

echo("The current system time is now ", time, "\n");
echo("The time is now ", timeonly, "\n");
echo("The date is now ", dateonly, "\n");
# Current system time = [08/06/98 16:49:24]  
# Time is now 16:49:24 EDT
# Date is now Thursday, 06 August 1998
```

### hatoi()

The hatoi() function converts a two's complement hex string to an integer. If you specify an index, the conversion begins at that location inside the string.

If the leading bit is set to (1<<31), then the return value will be a negative number.

The hatoi() function ignores any white space leading up to the number. However, unlike the atoi() function, hatoi() will return a NULL if there are any characters after the hex number or if the hex value overflows the size of the CPcode integer.

<table>
<thead>
<tr>
<th><strong>hatoi()</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td>hatoi(string str, [int index := 0]);</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
</tbody>
</table>
| • string str - the hex string that is to be converted to an integer  
• int index - the point in the string at which to begin the conversion |
| **Return Value** |
| • int - the integer value of the hex string, str  
NULL - the string is empty, contains characters that are not valid hex digits, or is too large to fit inside an CPcode integer. |

<table>
<thead>
<tr>
<th><strong>Example</strong></th>
</tr>
</thead>
</table>
| { hatoi("10") == 16  
hatoi("ff") == 255  
hatoi("ffXX") == NULL  
hatoi("80000000") == -2147483648  
hatoi("ffffffffffffff") == NULL  
hatoi("monkeys: 41", 8) == 65  
hatoi(""") == NULL  
hatoi(42) == NULL |

### host()

The host() function constructs an IPv4host address from other parameters (for example, information in a data stream).
For example, the PORT command in FTP sends the octets of an IP address as a comma-separated, printable list. The host() function could be used to create an IP address from the list. The long() function could be used to gather information from a packet, using the single-integer syntax of the host() function to create an IP address. This can be easier than extracting each octet individually.

### host()

**Syntax**

```
host({ip, string, list, int} a, [int b, int c, int d]);
```

**Parameters**

- `{ip, string, list, int}` - a 32-bit, host-endian integer containing:
  - a list of four integers representing each octet in the address
  - the full IPv4 address
  - a string representation of an IP address
  - the first of four integers representing each IP address octet
  - int b - in the case of four octets, the second octet
  - int c - in the case of four octets, the third octet
  - int d - in the case of four octets, the fourth octet

**Return Value**

`ip` - an IPv4 address in network byte order

**Example**

```
$foo = host(208, 217, 179, 1);
$foo = host(0xD0D9b301);
$foo = host([208, 217, 179, 1]);
$foo = host("208.217.179.1");
$foo = host(208.217.179.1);
```

### host6()

The host6() function constructs an IPv6 host address from the supplied parameter.

**Syntax**

```
host6(string address, [int binary]);
```

**Parameters**

- `string address` - an textual or binary IPv6 address
- `int binary` - tells host6() whether to parse the address parameter as a binary 16-byte address, where:
  - `0` - indicates that the address parameter is textual
  - `1` - indicates that the address parameter is binary

**Return Value**

`ip` - an IPv6 address in network byte order

**Example**

```
$foo = host6("::1");
$foo = host6("abcd:0123::5432:1000", 0);;
```

### incksum()

The incksum() function computes a 16-bit checksum, summing two characters at a time, which corresponds to the IP header checksum.

incksum() can be used to either verify or compute checksums.
incksum()

<table>
<thead>
<tr>
<th>incksum()</th>
<th>incksum({string, list} in);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>incksum({string, list} in);</td>
</tr>
<tr>
<td>Parameters</td>
<td>• {string, list} in - the string or list of strings for which the checksum should be computed</td>
</tr>
<tr>
<td>Return Value</td>
<td>int - an integer checksum</td>
</tr>
<tr>
<td>Example</td>
<td>incksum(&quot;foo bar&quot;) == 61865</td>
</tr>
<tr>
<td></td>
<td>incksum([&quot;foo&quot;, &quot;bar&quot;])) == 53673</td>
</tr>
<tr>
<td></td>
<td>incksum([&quot;fo&quot;, &quot;o &quot;, &quot;ba&quot;, r&quot;])) == 61865</td>
</tr>
</tbody>
</table>

index()

The index() function searches for the first occurrence of the little string in the big string. The index count starts with zero.

Using a list of little strings may not always be an ideal solution. If the list is static, it may be significantly faster to use a regular expression before calling the index() function.

The prefix() function is a faster case of the index() function.

<table>
<thead>
<tr>
<th>index()</th>
<th>index(string big, {list, string} little);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>index(string big, {list, string} little);</td>
</tr>
<tr>
<td>Parameters</td>
<td>• string big - the string to be searched</td>
</tr>
<tr>
<td></td>
<td>• {list, string} little - the string, or list of strings, to be located</td>
</tr>
<tr>
<td>Return Value</td>
<td>• int - the index of the little string into the big string</td>
</tr>
<tr>
<td></td>
<td>• -1 - an occurrence of little string could not be found in big string</td>
</tr>
<tr>
<td>Example</td>
<td>index(&quot;Network Security&quot;, &quot;work&quot;) == 3</td>
</tr>
<tr>
<td></td>
<td>index(&quot;123&quot;, &quot;1&quot;) == 0</td>
</tr>
</tbody>
</table>

io_close()

The io_close() function closes a previously opened file descriptor.

io_* functions do not integrate with the caching implementation provided with the url_* functions. If caching is desired or network access is desired, use the url_* functions instead.

<table>
<thead>
<tr>
<th>io_close()</th>
<th>io_close (int fd);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>io_close (int fd);</td>
</tr>
<tr>
<td>Parameters</td>
<td>Int fd – file descriptor to be closed</td>
</tr>
<tr>
<td>Return Value</td>
<td>None</td>
</tr>
<tr>
<td>Example</td>
<td>io_close($fd);</td>
</tr>
</tbody>
</table>
io_delete()  
The io_delete() function deletes a file from the file system.  
Io_* functions do not integrate with the caching implementation provided with the url_* functions. If caching is desired or network access is desired, use the url_* functions instead.

<table>
<thead>
<tr>
<th>io_delete()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
</tbody>
</table>
| Return Value         | `0 on success  
                       Error code otherwise` |
| Example              | `$filename = "./test.txt";  
                       io_delete($filename);` |

io_open()  
The io_open() function returns an I/O descriptor for reading or writing. The file will be automatically closed when the calling CPCode filter exits.  
Io_* functions do not integrate with the caching implementation provided with the url_* functions. If caching is desired or network access is desired, use the url_* functions instead.

<table>
<thead>
<tr>
<th>io_open()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
<tr>
<td>Example</td>
</tr>
</tbody>
</table>

io_read()  
The io_read() function reads data from an I/O descriptor.  
Io_* functions do not integrate with the caching implementation provided with the url_* functions. If caching is desired or network access is desired, use the url_* functions instead.

<table>
<thead>
<tr>
<th>io_read()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
</tbody>
</table>
| Return Value       | `list – list of strings representing the contents of the file.  
                       NULL if file descriptor is invalid.` |
### io_read()

The `io_read()` function reads data from an I/O descriptor.

**Example**

```bash
$fd = io_open("./test.txt");
...
$contents = io_read($fd);
io_close($fd);
```

### io_readlines()

The `io_readlines()` function reads data from an I/O descriptor.

**Io_* functions do not integrate with the caching implementation provided with the url_* functions. If caching is desired or network access is desired, use the url_* functions instead.**

#### Syntax

```
list io_readlines (int fd, <int chomp>);
```

#### Parameters

- `int fd`: file descriptor to be read
- `int chomp`: optional flag indicating whether or not to remove newline characters from the lines returned. Will remove "\n" as well as "\n\n."

#### Return Value

- `list`: list of strings representing the contents of the file.
- `NULL` if file descriptor is invalid.

**Example**

```bash
$fd = io_open("./test.txt");
...
$contents = io_readlines($fd);
io_close($fd);
```

### io_write()

The `io_write()` function writes data to an I/O descriptor. Writes do not append lines to a file, rather a single io() call will replace the file in question.

**Io_* functions do not integrate with the caching implementation provided with the url_* functions. If caching is desired or network access is desired, use the url_* functions instead.**

#### Syntax

```
list io_write(int fd, list data);
```

#### Parameters

- `int fd`: file descriptor to be written to.
- `list data`: A list of strings to write into a file.

#### Return Value

`None`

**Example**

```bash
$fd = io_open(cat(scope(),"-known-ports"));
...
io_write($fd, ["80","8080","9090"]);
io_close($fd);
```
kiss_compile()

A different name for the kiss_pattern() function. See kiss_pattern() (on page 70).

kiss_pattern()

The kiss_pattern() function creates a list of patterns into a single multi-pattern search. It behaves and is used in a manner similar to the multi_pattern() function.

<table>
<thead>
<tr>
<th>kiss_pattern()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td>String kiss_pattern(list pattern_list);</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>list pattern_list – this is a list of lists. Each list pair contains a pattern and a return token. Function pointers are not allowed for return tokens but any other data types are.</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td>string – the compiled pattern suitable for using with kiss_search();</td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>MY_PATTERN = kiss_pattern([ [&quot;x0&quot;,1],[&quot;x0x0&quot;,2], [&quot;y0y0&quot;,3], [&quot;y0y0dyne&quot;,4] ]);</td>
</tr>
</tbody>
</table>

kiss_search()

The kiss_search() function compares a kiss_pattern expression to a given string. It behaves and is used in a manner similar to multi_search();

<table>
<thead>
<tr>
<th>kiss_search()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td>list kiss_search(string kiss_pattern, string str);</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>string kiss_pattern – compiled regular expression returned from kiss_pattern()</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td>list – a list of tokens that are returned</td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>MY_PATTERN = kiss_compile([ [&quot;x0&quot;,1],[&quot;x0x0&quot;,2], [&quot;y0y0&quot;,3], [&quot;y0y0dyne&quot;,4]]</td>
</tr>
<tr>
<td>func my_search {</td>
</tr>
<tr>
<td>if ($matches = kiss_search(MY_PATTERN,$1)) {</td>
</tr>
<tr>
<td>foreach $elm inside ($matches) {</td>
</tr>
<tr>
<td>if ($elm == 1) echo (&quot;x0&quot;);</td>
</tr>
<tr>
<td>else if ($elm == 2) echo (&quot;x0x0&quot;);</td>
</tr>
<tr>
<td>else if ($elm == 3) echo (&quot;y0y0&quot;);</td>
</tr>
<tr>
<td>else if ($elm == 4) echo (&quot;Is your name John Smith?&quot;);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
| }

le_long()

The le_long() function returns a four-byte signed integer from the specified offset within a string. It extracts the long integer in host order without converting it.
### le_long()

**Syntax**

```cpp
le_long(string str, int offset);
```

**Parameters**

- `string str` - the string to be examined, beginning at byte 0
- `int offset` - the offset in the string for the first byte to be returned

**Return Value**

- `int` - the offset integer, as a signed value
- `error` - the offset was out of bounds

**Example**

```cpp
le_long("^A^@^@^@", 0) == 0x00000001;
le_long("^@^A^@^@", 0) == 0x00000100;
le_long("^@^@^A^@", 0) == 0x00010000;
```

### le_short()

The `le_short()` function returns a two-byte signed integer from the specified offset within a string. It extracts the short integer in host order without converting it.

**Syntax**

```cpp
le_short(string str, int offset);
```

**Parameters**

- `string str` - the string to be examined, beginning at byte 0
- `int offset` - the offset in the string for the first byte to be returned

**Return Value**

- `int` - the offset integer, as a signed value
- `error` - the offset was out of bounds

**Example**

```cpp
le_short("^A^@", 0) == 0x00000001;
le_short("^@^A", 0) == 0x00000100;
```

### le_ulong()

The `le_ulong()` function returns a four-byte unsigned integer from the specified. It extracts the long integer in host order without converting it.

The `le_ulong()` function returns the same result as the `le_long()` function. It is provided for backwards compatibility. The internal representation of all CPcode integers is signed. Therefore, the `le_ulong()` function yields a signed integer.

**Syntax**

```cpp
le_ulong(string str, int offset);
```

**Parameters**

- `string str` - the string to be examined, beginning at byte 0
- `int offset` - the offset in the string for the first byte to be returned

**Return Value**

- `int` - the offset integer, as an unsigned value
- `error` - the offset was out of bounds

**Example**

```cpp
le_ulong("^A^@^@^@", 0) == 0x00000001;
le_ulong("^@^A^@^@", 0) == 0x00000100;
le_ulong("^@^@^A^@", 0) == 0x00010000;
le_ulong("\x00xff\xff\xff", 0) == -256
```
le_ushort()

The le_short() function returns a two-byte unsigned integer from a specified offset within a string. It extracts the short integer in host order without converting it.

<table>
<thead>
<tr>
<th>le_short()</th>
<th>le_ushort(string str, int offset);</th>
</tr>
</thead>
</table>

**Syntax**

- string str - the string to be examined, beginning at byte 0
- int offset - the offset in the string for the first byte to be returned

**Parameters**

- string str
- int offset

**Return Value**

- int - the offset integer, as an unsigned value
- error - the offset was out of bounds

**Example**

le_ushort("^A^@", 0) == 0x00000001;
le_ushort("^@^A", 0) == 0x00000100;

listadd()

The listadd() function constructs a list at runtime, or adds items to an existing list.

NULL items are added to the list.

The listadd() function has a pre-optimized fastpath, where li, ... is its most relevant parameter. It operates more efficiently if its input list is the same CPcode variable as its output list.

However, duplicating the list may be an expensive operation, as in the example:

$listB = listadd($listA, 1, 2, 2);

<table>
<thead>
<tr>
<th>listadd()</th>
<th>listadd(list li, any item, [...]);</th>
</tr>
</thead>
</table>

**Syntax**

- list li - the existing list. If this value is NULL, the function creates an empty list and adds to that list.
- any item - the first item to be added to the tail of the list
- ... - a series of items to be added to the tail of the list

**Parameters**

- list li
- any item
- ...

**Return Value**

- The original list, or the newly constructed list, with the items added to it.

**Example**

badIPs = [ 208.217.179.1, 208.217.179.3 ];
badIPs = listadd(badIPs, ip.src, ip.dst);
A = [1, 2, 3];
B = [4, 5, 6];
C = listadd(A, B); # C == [1, 2, 3, [4, 5, 6]]

listcombine()

The listcombine() function combines two lists. It appends each element from the second list to the first list.

The listcombine() function differs from the listadd() function, which adds the entire second list to the first list instead of adding each individual element of the second list.

The listcombine() function has a pre-optimized fastpath, where list is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

<table>
<thead>
<tr>
<th>listcombine()</th>
<th></th>
</tr>
</thead>
</table>
listcombine()

Syntax
list listcombine(list a, list b);

Parameters
- list a - the first list to process
- list b - the second list to process

Return Value
- list - a list containing all the elements of both lists
- The original list, or the newly constructed list, with the items added to it.

Example
a = [1, 2, 3];
b = [4, 5, 6];
c = listcombine(a, b);
c == [1, 2, 3, 4, 5, 6]

listglom()

See listcombine() (on page 72)

listlen()

The listlen() function returns a count of the number of elements in a list.

Syntax
listlen(list li);

Parameters
- list li - the list whose elements are to be counted

Return Value
- int - the number of elements in the list
- error - the parameter is not a list

The listlen() function does not count the elements of sublists as separate elements.

The listlen() function is faster than the description implies. Lists keep an internal count of their members, which this function returns.

Example
portlist = [21, 23, 80];
listlen(portlist) == 3;
superlist = [1, 2, 3, [4, 5, 6]];
listlen(superlist) == 4;

long()

The long() function returns a four-byte signed integer from a specified offset within a string. It converts the value from network byte order in the string to the host's byte order. Network byte order is big endian, therefore the most significant byte is stored at the lowest address in memory.

The long() function is useful for extracting values from packets.

Syntax
long(string str, int offset);
**long()**

| Parameters | string str - the string to be examined, beginning at byte 0  
|           | int offset - the offset in the string for the first byte to be returned |
| Return Value | int - the offset integer, as a signed value  
|            | error - the offset was out of bounds |
| Example   | # the structure starts with a 4-byte  
|           | # integer in network byte order.  
|           | # It will be converted to host byte  
|           | # order before placing it in $data  
|           | $data = long($struct_data, 0); |

**lsort()**

The lsort() function sorts the members of a list according to type, grouping elements of the same type together. Within each data type, the lsort() function sorts the elements in ascending order.

The order of the elements within the group is not guaranteed.

The lsort() function has a pre-optimized fastpath, where li is its most relevant parameter. It operates more efficiently if its input list is the same CPcode variable as its output list.

| Syntax      | lsort(list li); |
| Parameters  | list li - the list to be sorted |
| Return Value | list - the sorted list |
| Example     | foo = [ 8, 4, 2, 9 10, 6, 4 ];  
|             | echo(lsort(foo));   # Slow  
|             | # [2, 4, 4, 6, 8, 9, 10]  
|             | foo = [ 8, 4, 2, 9 10, 6, 4 ];  
|             | foo = lsort(foo);  # Fast  
|             | echo(foo);  # [2, 4, 4, 6, 8, 9, 10] |

**luniq()**

The luniq() function creates a list of the unique elements from another list (that is, removes duplicates). It compares adjacent elements of a list. If they are equal, the luniq() function adds only one of the elements to the new list. If they are not equal, the luniq() function adds both elements to the list.

The luniq() function determines if elements are equal using the criteria for the == operator.

To guarantee that the new list contains unique elements, ensure that the list passed to the luniq() function is sorted first. It is much faster to assign the sorted result to the list being sorted than it is to use a temporary list, either explicitly or implicitly.

| Syntax      | luniq(list li); |
### luniq()

| Parameters | • list li - the list from which duplicate elements will be removed |
| Return Value | • list - a new list with duplicate adjacent elements removed  
• error - the parameter is not a list |
| Example | foo = [ 8, 4, 2, 8 10, 6, 4 ];  
echo(luniq(foo));   # Slow  
# [8, 4, 2, 8, 10, 6, 4]  
foo = [ 8, 8, 9, 9 10, 10, 4 ];  
foo = luniq(foo);  
echo(foo);  
# [8, 9, 10, 4] |

### match()

See regexec() (on page 82).

### multi_compile()

See multi_pattern() (on page 75).

### multi_escape()

The multi_escape() function escapes any reserved special characters in a string.
If used in conjunction with the multi_pattern() function, the multi_pattern() function can then accept the string literally.

<table>
<thead>
<tr>
<th>multi_escape()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>multi_escape(string str);</td>
</tr>
<tr>
<td>Parameters</td>
<td>• string str - the string to escape</td>
</tr>
<tr>
<td>Return Value</td>
<td>• string - the escaped string</td>
</tr>
</tbody>
</table>
| Example | { $foo = multi_escape("foo.exe");  
  # $foo == "foo\.exe"  
  $mp = multi_pattern([$foo, 1]);  
} |

### multi_pattern()

The multi_pattern() function creates a complex regular expression from a list of multiple patterns. Unlike normal regular expressions, a multi-pattern search can perform the match function in a single iteration through the search string. A typical multi-pattern search is sub 0(n).

The allowable regular expression characters are:

• +
• *
Multi-pattern search compilation is resource intensive and should be done only once and then assigned to a global variable.

The `multi_pattern()` function should never be called from a filter.

<table>
<thead>
<tr>
<th><strong>multi_pattern()</strong></th>
<th>multi_pattern(list pattern_list, [int case_insensitive]);</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>list pattern_list - the list of patterns and their return tokens</td>
</tr>
<tr>
<td></td>
<td>int case_insensitive - tells multi_pattern() whether the pattern is case insensitive, where:</td>
</tr>
<tr>
<td></td>
<td>0 - indicates that the pattern is case sensitive</td>
</tr>
<tr>
<td></td>
<td>a non-zero integer - indicates that the pattern is case insensitive</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>string - a complex regular expression</td>
</tr>
</tbody>
</table>
multi_pattern()

Example

code = multi_pattern( [ ["one", 1], ["two", 2], ["three", 3] ] );
multi_search(code, "one three") == [1,3]
multi_search(code, "two") == [2]
multi_search(code, "three") == [3]
multi_search(code, "THREE") == NULL
multi_search(code, "foo") == NULL

code = multi_pattern( [ ["[oO][nN][eE]", 1] ] );
multi_search(code, "one") == [1]
multi_search(code, "ONE") == [1]
multi_search(code, "oNe") == [1]
multi_search(code, "xxx oNe xxx") == [1]
multi_search(code, "foo") == NULL

code = multi_pattern( [ ["^[oO][nN][eE]", 1] ] );
multi_search(code, "one") == [1]
multi_search(code, "ONExxx") == [1]
multi_search(code, "oNExxx") == [1]
multi_search(code, "xxx oNe xxx") == NULL
multi_search(code, "foo") == NULL

code = multi_pattern( [ ["x90", "NOP string"], ["root", "root login"] ] );
multi_search(code, "xxx x90\x90\x90 xxx") == "NOP string"
multi_search(code, "xxx root xxx") == "root login"

code = multi_pattern( [ ["foo", "bar", "zah"] ] );
multi_search(code, "xxx foo xxx") == "foo"
multi_search(code, "xxx zah xxx") == "zah"
multi_search(code, "xxx dog xxx") == NULL

multi_search()

The multi_search() function compares a multi-pattern expression to a string and returns the tokens associated with all matches.

multi_search()

Syntax
multi_search(string multi_pattern, string str);
### multi_search()

| Parameters | string multi_pattern - the multi-pattern expression to be compared  
|           | string str - the string to be compared  
| Return Value | list - the matched patterns' tokens  
|             | NULL - no string match was found within the multi-pattern expression  
| Example   | See the multi_pattern() function Examples (see "multi_pattern()" on page 75).  

### net()

The net() function uses the supplied arguments to construct an IPv4 address from several different data types.

| Syntax | net({string, list, ip, int} a, [{ip, int} b, int c, int d, int e, int f, int g, int h]);  
| Parameters | {string, list, ip, int} a - a 32-bit host-endian integer containing:  
|           | a list of eight integers representing each octet in the address/mask  
|           | a full IPv4 address  
|           | a string representation of an IP address and netmask  
|           | the first of four integers representing each IP address octet  
|           | {ip, int} b - one of the following:  
|           | in the case of eight octets, the second octet  
|           | in the case of two 32-bit integers, the netmask  
|           | in the case of two network variables, the netmask  
|           | int c - the third octet of the network address  
|           | int d - the fourth octet of the network address  
|           | int e - the first octet of a netmask, or the netmask in CIDR notation  
|           | int f - the second octet of a netmask  
|           | int g - the third octet of a netmask  
|           | int h - the fourth octet of a netmask  
| Return Value | ip - an IPv4 address/network in network byte order  


### net()

The `net()` function constructs a network from the supplied parameter.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>net6(string network);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>string network - a textual IPv6 network</td>
</tr>
<tr>
<td>Return Value</td>
<td>ip - an IPv6 network in network byte order</td>
</tr>
<tr>
<td>Example</td>
<td>$foo = net6(&quot;::1/128&quot;);</td>
</tr>
<tr>
<td></td>
<td>$foo = net6(&quot;abcd:0123::/32&quot;);</td>
</tr>
</tbody>
</table>

### net6()

The `net6()` function constructs an IPv6 network from the supplied parameter.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>net6(string network);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>string network - a textual IPv6 network</td>
</tr>
<tr>
<td>Return Value</td>
<td>ip - an IPv6 network in network byte order</td>
</tr>
<tr>
<td>Example</td>
<td>$foo = net6(&quot;::1/128&quot;);</td>
</tr>
<tr>
<td></td>
<td>$foo = net6(&quot;abcd:0123::/32&quot;);</td>
</tr>
</tbody>
</table>

### pattern()

See `regcomp()` (on page 81)

### pred()

The `pred()` function finds and returns the logical predecessor of the indexed item in an array. The `pred()` function allows reverse foreach-like behavior, but at a larger performance cost.

| Syntax          | pred(array arr, any index); |
### pred()

**Parameters**
- array arr - the array from which to extract the predecessor
- any index - the index to the item whose predecessor will be extracted

**Return Value**
any - the predecessor of the indexed item in the array (that is, the item that will be extracted)

**Example**
```c
array[0] = "zero";
array[1] = "one";
array[2] = "two";
pred(array, -1) == NULL
pred(array, 0) == NULL
pred(array, 1) == 0
pred(array, 2) == 1
pred(array, 500) == NULL
```

### prefix()

The prefix() function compares two strings and returns true if the second string is a prefix of the first string.

The prefix() function is a faster case of the index() function.

Use the caseprefix() function for a case-insensitive comparison.

**Syntax**
```c
prefix(string big, string little);
```

**Parameters**
- string big - the string against which little will be compared
- string little - the smaller string, which will be compared to the beginning of big

**Return Value**
int, where:
- 1 - little is the same as the beginning of big
- 0 - little is not the same as the beginning of big

**Example**
```c
prefix("ABCDEFG", "ABC") == 1
prefix("ABCDEFG", "aBc") == 0
prefix("ABCDEFG", "DEF") == 0
```

### prunearray()

The prunearray() function randomly removes a specified percentage of elements in an array, and is intended to be used in conjunction with the overflow memory exception and the system.pressure packet variable.

If CPcode uses a global state table in an array, the system stability can be put at risk if the array grows arbitrarily large and uses all of the system's memory. The prunearray() function should be used to randomly (and fairly) free part of those global arrays.

**Syntax**
```c
prunearray(array ar, int percent);
```
prunearray()

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array ar</td>
<td>the array to prune</td>
</tr>
<tr>
<td>int percent</td>
<td>the percentage of the array to prune</td>
</tr>
</tbody>
</table>

| Return Value               | array - the pruned array                          |

| Example                    | STATE_TABLE = prunearray(STATE_TABLE, system.pressure) |

regcomp()

The regcomp() function creates and compiles a regular expression from a string. The compiled regular expression can be compared against other strings using the regex() function.

The regcomp() function is not a fast operation. Therefore, it should be used only on global variables, and only once for each variable.

To mitigate the performance impact of regular expression compilation from a filter, there is a 256-entry, fully associative regcomp cache. However, it is still better to execute a single regcomp() on a global variable.

The regcomp() function recognizes those regular expressions that are recognized by grep or the regcomp() function on UNIX systems.

The regcomp() function does not work on strings that contain Nulls.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>regcomp(string pattern);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>string pattern - the string to be converted into a regular expression</td>
</tr>
<tr>
<td>Return Value</td>
<td>string - a regular expression pattern</td>
</tr>
</tbody>
</table>
### regcomp()

**Example**

```plaintext
example = regcomp("abc+");
"abc" == example     # TRUE
"abcccc" == example  # TRUE
"defccc" == example  # FALSE
"abcdef" == example  # TRUE
"defabc" == example  # TRUE
"defabcdef" == example # TRUE
example = regcomp("^abc");
"abc" == example     # TRUE
"abcccc" == example  # TRUE
"defccc" == example  # FALSE
"abcdef" == example  # TRUE
"defabc" == example  # FALSE
"defabcdef" == example # FALSE
example = regcomp("^abc|^def");
"abc" == example     # TRUE
"abcccc" == example  # TRUE
"defccc" == example  # TRUE
"abcdef" == example  # TRUE
"defabc" == example  # TRUE
"defabcdef" == example # TRUE
"Xdefabcdef" == example # FALSE
```

### regexec()

The `regexec()` function compares a regular expression to a string, and returns an integer result indicating whether the patterns matched. This function utilizes regular expressions compiled by the `regcomp()` function.

#### Syntax

```plaintext
regexec(string pattern, string str);
```

#### Parameters

- **string pattern** - the regular expression to be compared
- **string str** - the string to be compared

#### Return Value

`int`, where:

- **0** - patterns did not match
- **1** - patterns matched

#### Example

```plaintext
example = regcomp("abc+");
regexec(example, "defabc") == 1
example = "^abc";
regexec(example, "abcdef") == 1
regexec(example, "defabc") == 0
```
**scope()**

The `scope()` function returns the name of the scope in which the CPcode is currently executing. Calls to `scope()` are evaluated at compile time and are the functional equivalent of using a string literal with a scope name. For example:

```cp
$my_scope = scope();
```

translates to:

```cp
$my_scope = "ftp_commands";
```

| **scope()** |  
| --- | --- |
| **Syntax** | `scope();` |
| **Parameters** | None |
| **Return Value** |  
| - string - a string showing the name of the scope in which the CPcode is currently executing. The string displays as:  
  - the protocol, an underscore, and the Protection Group name if this CPcode is for a Protection Group  
  - the protocol if this CPcode is protocol-wide CPcode |
| **Example** | func foo  
{  
echo("debug:: foo called from", $1);  
...  
}  
...  
foo(scope()); |

**sed()**

The `sed()` function rewrites strings. It corresponds to the UNIX `sed` command (Stream EDitor), which simplifies later processing by replacing or removing items from a string.

The `sed()` function must compile the pattern into a regular expression, which is extremely slow. To alleviate the compilation bottleneck, the `sed()` function uses a 64-entry, fully associative regex cache. Using the same pattern is normally much faster than using many patterns. To this end, the cache is not LRU (least recently used); its replacement scheme is to remove the closest match to the new entry on the assumption that they were both dynamically generated from the same Protection Group. Dynamic patterns are discouraged and are penalized by low cache precedence.

| **sed()** |  
| --- | --- |
| **Syntax** | `sed(string pattern, string replace, string input, [int matchOnce :=0]);` |
| **Parameters** |  
| - string pattern - the string pattern to be matched and replaced  
- string replace - the replacement string  
- string input - the input string  
- int matchOnce - an optional Boolean where:  
  - 0 - replace all occurrences of the pattern  
  - 1 - replace the first occurrence of the pattern |
| **Return Value** | string - the rewritten string |
**sed()**

The sed() function returns a two-byte signed integer from a specified offset within a string. It converts the value from network byte order in the string to the host's byte order. Network byte order is big endian; therefore, the most significant byte is stored at the lowest address in memory.

The sed() function is useful for extracting values from packets.

**Syntax**

```
short(string str, int offset);
```

**Parameters**

- string str - the string to be examined, beginning at byte 0
- int offset - the offset in the string for the first byte to be returned

**Return Value**

- int - the offset integer, as a signed value
- error - the offset was out of bounds

**Example**

```
# Structure starts with 2-byte int
# in network byte order.
# It will be converted to host byte order
# before placing it in $data
$data = short($struct_data,0);
```

**split()**

The split() function breaks apart and returns a list of strings derived from another string. Characters before the specified delimiter become the first element in the list, and characters after the delimiter become the second element in the list, and so forth.

It is permissible to split on the NULL character or other binary characters by entering hex characters (for example, `\x00\xff`) in the string.
The split() function will ignore any leading delimiters in the string. If no delimiter is specified, white space characters (a space, \t, \r, and \n) will become the default.

<table>
<thead>
<tr>
<th>split()</th>
<th>split(string str,[string delimiter]);</th>
</tr>
</thead>
</table>

**Syntax**

**Parameters**
- string str - the string to be split
- string delimiter - the parameter on which the string should be split

**Return Value**
list - a list of strings

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>split(&quot;www.company.com&quot;, &quot;.&quot;) == [&quot;www&quot;, &quot;company&quot;, &quot;com&quot;]</td>
</tr>
<tr>
<td>split(“Foo and his pet rock”) == [&quot;Foo&quot;, &quot;and&quot;, &quot;his&quot;, &quot;pet&quot;, &quot;rock&quot;]</td>
</tr>
<tr>
<td>split(“One,Two Three;four”, &quot;,;”) == [”One”, &quot;Two&quot;, &quot;Three&quot;, &quot;four&quot;]</td>
</tr>
<tr>
<td>split(&quot;    ONE   TWO   &quot;) == [&quot;ONE&quot;, &quot;TWO&quot;]</td>
</tr>
</tbody>
</table>

str()

The str() function converts each parameter to its textual representation and returns a list of those strings. The str() function may be impacted by the change in the behavior of string variables. Refer to the description of the string data type on page B-5 for additional information about this behavior.

<table>
<thead>
<tr>
<th>str()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>str(…);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>… - parameters to convert to textual representation and put in a list</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>list - a list of strings</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>str(1, 2, &quot;three&quot;, 4.5.6.7) == [&quot;1&quot;, &quot;2&quot;, &quot;three&quot;, &quot;4.5.6.7&quot;]</td>
</tr>
</tbody>
</table>

strcasecmp()

The strcasecmp() function compares two strings, character by character, ignoring case. Two characters are considered equal if they have the same byte value, or if they are alphabetically the same character without regard to case (for example, D is equal to d).

The strcasecmp() function is useful for comparing case-insensitive strings. Many protocols specify that strings must be case insensitive.

The strcasecmp() function has a legacy optimization to first compare string lengths. If one string is shorter than the other is, the character comparison is not done and <0 is returned.

The first occurrence of an unequal character causes the function to stop the comparison. If the characters are equal, comparison continues until the end of the string is reached.

<table>
<thead>
<tr>
<th>strcasecmp()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>strcasecmp(string s1, string s2);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcasecmp(&quot;abc&quot;, &quot;aBc&quot;) == 0</td>
</tr>
<tr>
<td>strcasecmp(&quot;abc&quot;, &quot;def&quot;) == -1</td>
</tr>
<tr>
<td>strcasecmp(&quot;abc&quot;, &quot;ABC&quot;) == 1</td>
</tr>
<tr>
<td>strcasecmp(&quot;abc&quot;, &quot;Abc&quot;) == 0</td>
</tr>
<tr>
<td>strcasecmp(&quot;abc&quot;, &quot;abd&quot;) == -1</td>
</tr>
</tbody>
</table>
strcasecmp()

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>• string s1 - the first string to be compared</td>
</tr>
<tr>
<td>• string s2 - the second string, which will be compared against the first</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>• int &lt;0 - the first string is less than the second string</td>
</tr>
<tr>
<td>• int 0 - the two strings are equal</td>
</tr>
<tr>
<td>• int &gt;0 - the first string is greater than the second string</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (strcasecmp($ftp_user, &quot;anonymous&quot;) == 0</td>
</tr>
</tbody>
</table>

strerror()

The strerror() function returns a Boolean string indicating whether an error occurred.

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>strerror(any arg);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>• any arg - the expression being evaluated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>string - a string indicating whether the argument evaluated as &quot;no error&quot; or &quot;unknown error&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$one = 1;$zero = 0;$strerror($one + $zero) == &quot;no error&quot;$strerror($one / $zero) == &quot;unknown error&quot;$div = $one / $zero;$strerror($div) == &quot;unknown error&quot;</td>
</tr>
</tbody>
</table>

strlen()

The strlen() function returns the length, in bytes, of a string or blob.

Unlike other programming languages, CPcode does not use a trailing NULL character.

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>strlen(string str);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>• string str - the string whose length is to be determined</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int - the length of str</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>strlen(&quot;123&quot;) == 3</td>
</tr>
</tbody>
</table>

strstr()

See index() (on page 67).
### strtype()

The `strtype()` function returns an integer representing the type of data that is in a string. The `strtype()` function is useful for performing one type of processing on data that contains only ASCII characters and another type of processing on binary data.

<table>
<thead>
<tr>
<th><strong>strtype()</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>strtype(string str);</code></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• string str - the string whose data is to be evaluated</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>int, where:</td>
</tr>
<tr>
<td></td>
<td>• 1 - NULL</td>
</tr>
<tr>
<td></td>
<td>• 2 - the string contains only digits</td>
</tr>
<tr>
<td></td>
<td>• 3 - the string contains only alphabetic characters, including ISO 8859-1 Latin-1 characters 192 through 255, except 215 and 247</td>
</tr>
<tr>
<td></td>
<td>• 4 - the string contains only spaces</td>
</tr>
<tr>
<td></td>
<td>• 5 - the string contains only white space, including ISO 8859-1 Latin-1 characters 10, 11, 12, 13, 32, and 160</td>
</tr>
<tr>
<td></td>
<td>• 6 - the string contains only alphanumeric characters</td>
</tr>
<tr>
<td></td>
<td>• 7 - the string contains only printable characters, including ISO 8859-1 Latin-1 characters 32 through 126 and 160 through 255</td>
</tr>
<tr>
<td></td>
<td>• 8 - the string contains only ASCII characters between 0 and 127</td>
</tr>
<tr>
<td></td>
<td>• 9 - the string contains only binary data</td>
</tr>
</tbody>
</table>
**strtype()**

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| func foo 
| { 
| type = strtype($1);
| if (type == 9) 
| echo("The data has a string type of ", type, "\n");
| else 
| echo("The data ", $1, " has a string type of ", type, "\n");
| 
| foo("21167");
| foo("Company");
| foo("Company123");
| foo(""");
| foo("Generic Company, Inc.");
| foo("www.company.com\00");
| foo("\xff \x01");
| The output is: 
| The data 21167 has a string type of 2
| The data Company has a string type of 3
| The data Company123 has a string type of 6
| The data has a string type of 1
| The data Generic Company, Inc. has a string type of 7
| The data www.company.com has a string type of 9
| The data has a string type of 9 |

**sublist()**

The sublist() function returns a list that is a subset of another list. It creates a new list, starting with the element specified as the start, and continuing for the number of elements specified for the length.

When using the sublist() function, remember that the first character in the string is always at zero. If a length is not included, the new list starts at the offset and uses the remaining elements in the old list. If the starting offset is past the end of the list, the new list is empty. If the length reaches past the end of the list, the new list uses the remaining elements in the list.

<table>
<thead>
<tr>
<th>sublist()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
</tbody>
</table>
| **Parameters** | • list li - the list to be searched
| | • int start - the offset in the list for the beginning of the new list
| | • int length - the number of elements of the first list to use in the second list |
| **Return Value** | list - the sublist |
### sublist()

**Example**

```
ourIP = listadd(our_IPs, 208.217.179.1, 208.217.179.2, 208.217.179.3);
ourIP = listadd(our_IPs, 208.217.179.4, 208.217.179.5, 208.217.179.6);
ourIP = listadd(our_IPs, 208.217.179.7, 208.217.179.8, 208.217.179.9);
echo(sublist(ourIP, 3, 3));
echo(sublist(ourIP, 8));
Displays:
[208.217.179.4, 208.217.179.5, 208.217.179.6]
[208.217.179.9]
```

### substr()

The `substr()` function creates a new string from another string, starting with the character specified at the offset start, and continuing for the number of characters specified for length.

When using the `substr()` function, remember that the first element in the list is always zero.

If a length is not included, the new string starts at the offset and uses the remaining characters in the old string.

**Example**

```
substr("Network Security Inc", 8, 8) == "Security"
substr("foo bar", 4) == "bar"
```

### succ()

The `succ()` function finds and returns the logical successor of the indexed item in a passed array.

The `succ()` function allows foreach-like behavior, but at a larger performance cost.

**Example**

```
succ(array arr, any index);
```
**succ()**

Example

```c
array[0] = "zero";
array[1] = "one";
array[2] = "two";
succ(array, 0) == 1
succ(array, 1) == 2
succ(array, 2) == NULL
succ(array, 500) == NULL
```

**suffix()**

The `suffix()` function compares two strings and returns true if the second string is a suffix of the first string. Use the `casesuffix()` function for a case-insensitive comparison.

**suffix()**

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>suffix(string big, string little);</code></td>
</tr>
</tbody>
</table>

**Parameters**

- `string big` - the string against which `little` will be compared
- `string little` - the smaller string, which will be compared to the end of `big`

**Return Value**

`int`, where:

- `1` - `little` is the same as the end of `big`
- `0` - `little` is not the same as the end of `big`

**Example**

```c
suffix("ABCDEFG", "EFG") == 1
suffix("ABCDEFG", "eFg") == 0
suffix("ABCDEFG", "ABC") == 0
```

**tolower()**

The `tolower()` function converts all English alphabetic characters from uppercase to lowercase. Other characters, including alphabetic characters in other character sets, are not converted.

The `tolower()` function has a pre-optimized fastpath, where `str` is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The `tolower()` function may be impacted by the change in the behavior of string variables. See `str` for additional information about this behavior.

The overall return string for the `tolower()` function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

**tolower()**

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tolower(string in);</code></td>
</tr>
</tbody>
</table>

**Parameters**

- `string in` - the string to be converted to lowercase

**Return Value**

`string` - the same string that was passed in, with all uppercase characters (A-Z) converted to lowercase characters (a-z).
### tolower()

The `tolower()` function converts all English alphabetic characters from lowercase to uppercase. Other characters, including alphabetic characters in other character sets, are not converted.

The `tolower()` function has a pre-optimized fastpath, where `str` is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The `tolower()` function may be impacted by the change in the behavior of string variables. Refer to the description of the string data type on page B-5 for additional information about this behavior.

The overall return string for the `tolower()` function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

<table>
<thead>
<tr>
<th>tolower()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
<tr>
<td><code>tolower(&quot;AB12fj4&quot;)</code> == &quot;ab12fj4&quot;</td>
<td></td>
</tr>
<tr>
<td><code>tolower(&quot;ABCdefG&quot;)</code> == &quot;abcdefg&quot;</td>
<td></td>
</tr>
<tr>
<td><code>$foo = tolower($foo);</code> # FAST</td>
<td></td>
</tr>
</tbody>
</table>

### toupper()

The `toupper()` function converts all English alphabetic characters from lowercase to uppercase. Other characters, including alphabetic characters in other character sets, are not converted.

The `toupper()` function has a pre-optimized fastpath, where `str` is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The `toupper()` function may be impacted by the change in the behavior of string variables. Refer to the description of the string data type on page B-5 for additional information about this behavior.

The overall return string for the `toupper()` function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

<table>
<thead>
<tr>
<th>toupper()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td></td>
</tr>
<tr>
<td><code>toupper(string in);</code></td>
<td></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>- string in - the string to be converted to uppercase</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>string - the same string that was passed in, with all lowercase characters (a-z) converted to uppercase characters (A-Z).</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
<tr>
<td><code>toupper(&quot;AB12fj4&quot;)</code> == &quot;AB12FJ4&quot;</td>
<td></td>
</tr>
<tr>
<td><code>toupper(&quot;ABCdefG&quot;)</code> == &quot;ABCDEFG&quot;</td>
<td></td>
</tr>
<tr>
<td><code>$foo = toupper($foo);</code> # FAST</td>
<td></td>
</tr>
</tbody>
</table>

### typeof()

The `typeof()` function returns the data type of an expression.

You can use the `typeof()` function for error checking or for conditional operations. However, there is a performance penalty associated with this usage.

<table>
<thead>
<tr>
<th>typeof()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td></td>
</tr>
<tr>
<td><code>typeof(any argument);</code></td>
<td></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>- any argument - the variable whose data type will be returned</td>
</tr>
</tbody>
</table>
typeof()

**Return Value**

<table>
<thead>
<tr>
<th>string</th>
<th>a string indicating the data type of the argument, where:</th>
</tr>
</thead>
<tbody>
<tr>
<td>alert</td>
<td>the argument is an alert</td>
</tr>
<tr>
<td>array</td>
<td>the argument is an array</td>
</tr>
<tr>
<td>ethmac</td>
<td>the argument is an ethmac address</td>
</tr>
<tr>
<td>filter tag</td>
<td>the argument is filter tag</td>
</tr>
<tr>
<td>function pointer</td>
<td>the argument is a function pointer</td>
</tr>
<tr>
<td>error</td>
<td>the argument is an error code</td>
</tr>
<tr>
<td>int</td>
<td>the argument is an integer</td>
</tr>
<tr>
<td>ipv4host</td>
<td>the argument is an IPv4 host address</td>
</tr>
<tr>
<td>ipv4net</td>
<td>the argument is an IPv4 network address</td>
</tr>
<tr>
<td>ipv6host</td>
<td>the argument is an IPv6 host address</td>
</tr>
<tr>
<td>ipv6net</td>
<td>the argument is an IPv6 network address</td>
</tr>
<tr>
<td>list</td>
<td>the argument is a list</td>
</tr>
<tr>
<td>recorder</td>
<td>the argument is a recorder</td>
</tr>
<tr>
<td>str</td>
<td>the argument is a string, a blob, or a pattern</td>
</tr>
<tr>
<td>unknown</td>
<td>a processing error occurred</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
if (typeof($1) != "str")
    return 0;
```

ubyte()

The ubyte() function returns a one-byte unsigned integer from the specified offset within a string.

<table>
<thead>
<tr>
<th>ubyte()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td>ubyte(string str, int offset);</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>• string str - the string to be examined, beginning at byte 0</td>
</tr>
<tr>
<td>• int offset - the offset in the string for the byte to be returned</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td>• int - the offset integer, as an unsigned value</td>
</tr>
<tr>
<td>• error - the offset is out of bounds</td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>#the eighth byte is the unsigned type code</td>
</tr>
<tr>
<td>$code = ubyte( $struct_data, 7 );</td>
</tr>
</tbody>
</table>

ulong()

The ulong() function returns a signed four-byte integer from a specified offset within a string. It converts the value from network byte order in the string to the host's byte order. Network byte order is big endian; therefore, the most significant byte is stored at the lowest address in memory.

The ulong() function is useful for extracting values from packets.

The ulong() function returns the same result as the long() function. It is provided for backwards compatibility. The internal representation of all CPcode integers is signed. Therefore, the ulong() function yields a signed integer.
**ulong()**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>ulong(string str, int offset);</th>
</tr>
</thead>
</table>
| Parameters | • string str - the string to be examined, beginning at byte 0  
• int offset - the offset in the string for the first byte to be returned |
| Return Value | • int - the offset integer, as a signed value  
• error - the offset is out of bounds |
| Example | # Structure starts with a 4-byte integer  
# in network byte order.  
# Converted to host byte order  
# before placing it in $data  
$data = ulong($struct_data, 0);  
# Caveat  
$blob = cat(-1);  
echo(ulong($blob, 0));  # -1  
echo(ulong($blob, 0) >> 2);  # -1  
$blob = cat(-4);  
echo(ulong($blob, 0) >> 1);  # -2  
# Be _very_ careful with  
# shifting signed integers.  
# Their sign bit is extended. |

**unpack()**

The unpack() function extracts the individual variables from a structure that was packed using the pack() function.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>int unpack(list schema, string packed, var1, var2, ...);</th>
</tr>
</thead>
</table>
| Parameters | • list schema - the result of a call to the packing_schema() function  
• string packed - a packed structure  
• var1, var2, varn - the variables in which to store data elements from the unpacked structure |
| Return Value | int - !NULL if the unpacking was successful |
**unpack()**

**Example**

```plaintext
{  
  schema = packing_schema("ip", "ip", "port");  
  ...
  $struct = pack(schema, $src_ip, $dst_ip, $dst_port);
  if (ALERTS[$struct] == NULL) ALERTS[$struct] = 1;
  else ALERTS[$struct] = ALERTS[$struct] + 1;
  ...
  unpack(schema, $struct, $src, $dst, $dport);
  msg = cat(ALERTS[$struct], " attacks from ", $src, " against ", $dst, ": ", $dport);
  echo( $msg);
}
```

**url_flush()**

The `url_flush()` function removes a specified URL from cache.

Cache control parameters are inherited by the CPcode runtime from the calling application for use with the `url_*` functions.

**url_flush()**

**Syntax**

```plaintext
url_flush( URL )
```

**Parameters**


**Return Value**

None

**url_needs_refresh()**

The `url_needs_refresh()` function indicates whether or not the specified URL needs to be updated because it has been modified and/or the cached representation is stale.

Cache control parameters are inherited by the CPcode runtime from the calling application for use with the `url_*` functions.

**url_needs_refresh()**

**Syntax**

```plaintext
int url_needs_refresh ( URL )
```

**Parameters**


**Return Value**

Int – an integer with one of the following possible values:

- 0 - the cached URL is up to date.
- 1 – the cached URL is stale.
- 2 – the URL is not cached.
url_open()

The url_open() function returns a handle to the resource which can be used with the I/O builtins (io_*) to retrieve data. The user must call io_close on the handle to release it.

If the cached representation is stale and the resource has been modified, the cached representation will be refreshed before the handle is returned.

Cache control parameters are inherited by the CPcode runtime from the calling application for use with the url_* functions.

<table>
<thead>
<tr>
<th>url_open()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
</tbody>
</table>
• <username> - optional parameter supporting basic HTTP authentication,  
• <password> - optional parameter supporting basic HTTP authentication. |
| **Return Value** | int – Returns a handle to the cached resource if successful, otherwise NULL if the not successful. |

url_readall()

The url_readall() function returns all of the contents of the specified URL.

Cache control parameters are inherited by the CPcode runtime from the calling application for use with the url_* functions.

<table>
<thead>
<tr>
<th>url_readall()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
</tbody>
</table>
• <username> - optional parameter supporting basic HTTP authentication,  
• <password> - optional parameter supporting basic HTTP authentication. |
| **Return Value** | string - A string containing the contents of the specified URL. |

url_readlines()

The url_readlines() function returns the contents of the specified URL as a list of strings with each entry containing a single line. Line terminators are not removed.

Cache control parameters are inherited by the CPcode runtime from the calling application for use with the url_* functions.

<table>
<thead>
<tr>
<th>url_readlines()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
</tbody>
</table>
url_readlines()

• <username> - optional parameter supporting basic HTTP authentication,  
• <password> - optional parameter supporting basic HTTP authentication. |
| Return Value | list - A list of strings. Each entry is a single line from the URL. |

ushort()

The ushort() function returns a two-byte value as an unsigned integer from a specified offset within a string. It converts the value from network byte order in the string to the host's byte order. Network byte order is big endian; therefore, the most significant byte is stored at the lowest address in memory.

The ushort() function is useful for extracting values from packets.

| Syntax | ushort(string str, int offset); |
| Parameters | • string str - the string to be examined, beginning byte 0  
• int offset - the offset in the string for the first byte to be returned |
| Return Value | • int - the offset integer, as an unsigned value  
• error - the offset is out of bounds |
| Example | # Structure starts with a 2-byte integer  
# in network byte order.  
# Converted to host byte order  
# before placing it in $data  
$data = ushort($struct_data, 0); |