Chapter 16 - Unlang Policy Language

This chapter describes the Unlang policy language supported by FreeRADIUS servers. It discusses:

- Overview
- Data types
- Configuration file format
- Module statements
- String expansion
- Programming statements
- Conditions/conditional expressions
- Attribute editing statements
- Configurable failover

This chapter describes the syntax of the language, followed by examples.

16.0 Overview

The server supports a simple processing language called “Unlang”, which is short for “unlanguage”. The original intention of using an “unlanguage” was to avoid creating yet another programming language.

Unlang allows simple conditional checks and editing of attributes and attribute lists. Where more complicated functionality is required, Perl or Python modules rlm_perl or rlm_python are recommended.

The goal of Unlang is to allow simple policies to be written with minimal effort. Conditional checks can be performed by the policies, which can then update the request or response attributes based on the results of those checks. Unlang can only be used in a processing section (e.g., authorize, authenticate, post-auth, preacct, accounting, pre-proxy, post-proxy, and session); it cannot be used anywhere else, including in configuration sections for a client or a module. The reason for this limitation is that the language is intended to perform specific actions on requests and responses. The client and module sections contain definitions for a client or module; they do not define how a request is processed.

The Unlang syntax is based on reserved words, which dictate how the text that follows should be interpreted. The syntax for each word is line oriented. Subsections are sometimes allowed, just like with other portions of radiusd.conf. Where the lines are too long, the backslash character can be used to link multiple lines together.

Note that unlike most programming languages, whitespace is important in Unlang. For example, the following two statements are not identical. Although the only difference between the two statements is the placement of the first hard return, the first will parse correctly, and the second will return a parse error:

```
if (foo) {
        ...
}
```

versus:

```
if (foo)
{
        ...
}
```

16.1 Data Types

Unlang supports a number of data types. These data types can be used in conditional expressions or when assigning a value to an attribute.

16.1.1 Numbers

Examples:

```
0
563
```

Numbers are unsigned integers that are composed of decimal digits. Signed numbers, floating point, hex, and octal numbers are not supported in Unlang.

These limitations are largely a result of the limitations of the underlying protocol. There is no way to transport signed numbers or floating point numbers in RADIUS, so there is no need to perform signed or floating point calculations.

The maximum value for a number is machine-dependent but is at least 32 bits or 4,294,967,296. In some cases, it may be necessary to perform calculations on numbers larger than 32 bits, as when calculating total data traffic to or from a user. In those cases, the calculations should either be one carefully to avoid 32-bit overflow or be done using real programming language.

16.1.2 Simple Words

Examples:

```
Hello
User-Name
```

Where a series of characters cannot be parsed as a decimal number, they are interpreted as a simple string composed of one word. These words are interpreted as static strings and are usually equivalent to placing the string in single quotes. When they are used in the left-hand side of a conditional expression they can be interpreted as a reference to an attribute.

This syntax is allowed for simplicity, as there are many cases where simple text can be used, and it would be awkward to require the simple text to be quoted.

16.1.3 IP Addresses

Examples:

```
192.0.2.16
::1
example.com
```

Depending on the context, a “simple word”, as above, may be interpreted as an IPv4 or an IPv6 address. This interpretation is usually done when the string is used in the context of an attribute, or to compare two addresses or assign an address to an attribute.

16.1.4 Single Quoted Strings

```
'string'
```

A single quoted string is interpreted without any dynamic string expansion. The quotes allow the string to contain spaces, which are not allowed in the `word` form described in the previous section. The single quote character can be placed in such a string by escaping it with a backslash.

Examples:

```
'hello'
'foo bar'
'foo\'bar'
'this is a long string'
```

16.1.5 Double-Quoted Strings

```
"string"
```

A double-quoted string is interpreted via the usual rules in programming languages for double quoted strings. The double-quote character can be placed in a string by escaping it with a backslash. Carriage returns and line-feeds can also be used via the usual `\r` and `\n` syntax.

The main difference between the single and double quoted strings is that the double quoted strings can be dynamically expanded. The syntax `${...}` is used for parse-time expansion and `%{...}` is used for run-time expansion. The difference between the two methods is that the `${...}` form is expanded when the server loads the configuration files and is valid anywhere in the configuration files. The `%{...}` form is valid only in conditional expressions and attribute assignments and is otherwise used verbatim.

The output of the dynamic expansion can be interpreted as a string, a number, or an IP address, depending on its context. In general, it is safest to assume that the result will be interpreted as a string.

Note that the language is not strongly typed, so the text "`0000`" can be interpreted as a data type "integer", having value zero, or a data type "string", having value "`0000`".

Examples:

```
"word"
"a string"
"this has embedded\ncharacters"
```

16.1.6 The Backtick Operator

```
`string`
```

The backtick operator is used to perform a run-time expansion similar to what is done with the Unix shell. The contents of the string are split into one or more sub-strings, based on intermediate whitespace. Each substring is then expanded as described above for double quoted strings. The resulting set of strings is used to execute a program with the associated arguments.

The output of the program is recorded, and the resulting data is used in place of the input string value. Where the output is composed of multiple lines, any carriage returns and line feeds are replaced by spaces.

For safety reasons, the full path to the executed program should be given. In addition, the string is split into arguments prior to dynamic expansion in order to prevent the expanded strings from being erroneously interpreted as more command-line arguments.

For performance reasons, we recommend that the use of back-quoted strings be kept to a minimum. Executing external programs is relatively expensive, and executing a large number of programs for every

request can quickly use all of the CPU time in a server. If many programs need to be executed, it is suggested that alternative ways to achieve the same result be found. In some cases, using a real language may be sufficient.

This operator is permitted only in conditional expressions and when assigning values to an attribute. In versions 2.1.11 and later, using it in an invalid context will return a syntax error, and the server will refuse to start. In versions of the server prior to 2.1.11, the data was treated as a single quoted string, and no run-time expansion or program execution is performed.

Examples:

```
`/bin/echo hello`
```

16.1.7 Character Escaping

The quotation characters in the above string data types can be escaped by using the backslash, or `\`, character. The backslash character itself can be created by using `\\`. Carriage returns and line feeds can be created by using `\n` and `\r`.

Examples:

```
"I say \"hello\" to you"
"This is split\nacross two lines"
```

16.2 Configuration File Format

The Unlang syntax is a superset of the syntax of the main configuration files, which was discussed earlier.

16.2.1 Syntax

Long lines can be split:

```
foo = \
bar
```

Comments are ignored:

```
# this is a comment
```

Comments can be placed at the tail end of any valid line:

```
foo = bar # assign "bar" to variable "foo"
```

16.2.2 Variable Assignment

```
variable = value
```

Assigns `value` to `variable`. The value can be any one of the data types defined in the previous section. The name of the `variable` is context-specific and is built into the server at compile time.

The default configuration files contain examples of the known variables along with the context in which they are valid. The meaning of each variable is defined in the comments that are located nearby. In many cases, those comments are the only documentation that exists for the variables.

Examples:

```
prefix = "/usr/share/local"
max_request_time = 30
reject_delay = 1
```

16.2.3 Variable References

```
variable = "${reference}"
```

A variable reference causes the configuration file parser to expand the string when the configuration file is being parsed. Further expansion may be done at run time if the string also contains the dynamic expansion string `%{...}`.

The data inside the `${...}` string must be a variable that exists in the configuration files. If the variable does not exist, a parse error is returned and the server will not start.

The reference can be any one of the following; the meanings are given below each example:

- reference
  A reference to a named variable in the current section. If the variable does not exist, then the variable is looked for in the main (i.e., global) configuration.
- .reference
  A reference to a named variable in the current section. The global configuration is not used.
- ..reference
  A reference to the named variable found in the parent section instead of the current section. Any number of dots (.) can be used, up to the global configuration.
- reference1.reference2.reference3
  A reference to the section named `reference1`, which contains a subsection named `reference2`, which in turn contains a variable `reference3`. These references may be nested to any depth.
- reference1[name].reference2
  A reference to a section named `reference1`, which has an instance called name, which in turn contains a variable `reference2`.

The above capabilities allow any portion of the configuration files to reference values taken from any other portion.

One additional expansion is supported.

- `$ENV{variable}`
  A reference to the environment variable named `variable`.

Examples:

```
foo = bar
baz = "${foo}" # assigns "bar" to "baz"
var = "a ${baz} string" # assigns "a bar string" to "var"
name = "${client[localhost].secret}"
ipaddr = $ENV{HOSTNAME}
```

16.2.4 Including Files

```
$INCLUDE path
```

The `$INCLUDE` directive "includes" a file into the current location of the configuration files, with the contents of the file replacing the `$INCLUDE` directive. The path has one or more forms, with the following meanings:

- filename
  Include a new `filename` relative to the current file being parsed.
- /path/to/filename
  Include `filename` using the given absolute path.
- directory/
  Include all of the files in the given directory. Any "hidden" files (i.e., with a leading dot (.) in their name) are ignored. All other files are read, including common editor "backup" files, such as ones with a trailing ~ in their name.
  For that reason, this directive should be used with care.

The only limitation of `$INCLUDE` files is that a section cannot be split across multiple files.

Examples:

```
$INCLUDE clients.conf
$INCLUDE sites-enabled/
$INCLUDE ${raddbdir}/foo/bar/baz
```

16.2.5 Sections

```
section {
      [ statements ]
}
```

A section is a way to group multiple statements together. Sections can be nested to any depth and may contain any valid statement. Empty sections are allowed but serve no purpose.

Names of sections and variables are scoped within the context of their parent section. It is a syntax error to have a section and variable with the same name in the same scope. The server looks for and parses sections with "known" names. These known sections have pre-defined meanings, which cannot be changed.

Examples:

```
foo {
    bar = baz
}
```

16.2.6 Instance Names

```
section-type instance-name {
    [ statements ]
}
```

An instance-name is a way to define a specific `instance` for a particular type of section. For example, the `client` section is used to define information about a client. When multiple clients are defined, they are distinguished by their `instance-name`.

The same `instance-name` applies to modules. The `sql` module defines interation with an SQL database. Individual instances of the `sql` module can define interactions with different databases.

Examples:

```
client localhost {
      ipaddr = 127.0.0.1
      ...
}
sql backup {
      ...
}
```

16.3 Module Statements

The Unlang syntax allows for policies to directly refer to modules and for modules to be placed in fail-over groups.

16.3.1 The module-name Statement

```
module-name
```

The `module-name` statement is a reference to the named module. Common module names include pap, `chap`, `files`, `eap`, and `sql`. The valid module names are determined by the modules that are defined in the `modules` subsection of `radiusd.conf`.

When processing reaches this point, the pre-compiled module is called. The module may succeed or fail and will return a status code to the Unlang interpreter detailing success or failure.

Examples:

```
chap
sql
```

Module Return Codes

When a module is called, it returns one of the following codes to the interpreter; the meaning of each code is detailed to the right of the code, below:.

Module Return Codes	
Name	**Meaning**
reject	the module rejected the request
fail	the module failed
ok	the module succeeded
handled	the module has handled the request itself

Module Return Codes	
Name	**Meaning**
invalid	the configuration was invalid
userlock	the user was locked out
notfound	information was not found
noop	the module did nothing
updated	the module updated the request

Table 16.3.1 Module Return Codes

These return codes can be used in a subsequent conditional expression (see the processing section for a more detailed description), thus allowing policies to perform different actions based on the behavior of the modules.

Return Codes as Modules

Some of the module return codes given above can be used in place of a `module-name`. This capability is implemented in the default configuration by the `always` module and exists so that a success or failure can be forced during the processing of a policy. The names and meanings are given below:

- **`fail`** Causes the request to be treated as if a database failure had occurred.
- **`noop`** Do nothing. This also serves as an instruction to the configurable failover tracking that nothing was done in the current section.
- **`ok`** Instructs the server that the request was processed properly. This keyword can be used to override earlier failures, if the local administrator determines that the failures are not catastrophic.
- **`reject`** Causes the request to be immediately rejected.

16.3.2 The module-name.section Statement

```
module-name.section
```

This variant of `module-name` is used in one processing section to cause the module to be called with the method of another processing section. It most often used to call a module's `authorize` method while processing the `post-auth` section.

The `module-name` portion must refer to an existing module. The `section` portion must refer to a processing section.

Examples:

```
sql.authorize
files.authorize
```

16.3.3 The redundant Statement

```
redundant {
       [ module-1 ]
```

```
        [ module-2 ]
}
```

The `redundant` section is used in place of a single module name and is used to configure inter-module fail-over. When the `redundant` statement is reached, the first module in the section is called. If that module succeeds, then the server returns from the `redundant` section and continues processing the rest of the policy.

However, if that module fails, then the next module in the list is called and checked for failure as described above. The processing continues until either a module succeeds or the end of the list has been reached.

There is no limit to the number of modules that can be listed inside of a `redundant` section.

Examples:

```
redundant {
        sql1
        sql2
        ok
}
```

16.3.4 The load-balance Statement

```
load-balance {
        [ module-1 ]
        [ module-2 ]
}
```

The `load-balance` section is similar to the `redundant` section except that only one module in the subsection is ever called. The module is chosen randomly, in a "load balanced" manner.

In general, the `redundant-load-balance` statement (see next section) is more useful than this one.

Examples:

```
load-balance {
        sql1
        sql2
}
```

16.3.5 The redundant-load-balance Statement

```
redundant-load-balance {
        [ module-1 ]
        [ module-2 ]
}
```

The `redundant-load-balance` section operates as a combination of the `redundant` and `load-balance` sections. When the section is entered, one module is chosen at random to process the request. If that module succeeds, then the server stops processing the section. If, however, the module fails, then one of the remaining modules is chosen at random to process the request. This process repeats until one module succeeds or until the list has been exhausted.

All of the modules in the list should be the same type (e.g., `ldap` or `sql`). All of the modules in the list should behave identically, otherwise different requests will be processed through different modules and will give different results.

Examples:

```
redundant-load-balance {
      sql1
      sql2
      sql3
}
```

16.3.6 The group Statement

```
group {
      [ statement ]
      [ statement ]
}
```

The `group` statement collects a series of statements into a list. The default processing sections of the server (`authorize`, `accounting`, etc.) are each based on a group statement.

The difference between a `redundant` and `group` statement is that the entries in a `group` statement are all processed, even if one of the returns `fail`.

The contents of a `group` subsection can be any valid statement.

Examples:

```
group {
      sql
      ldap
      file
      if (updated) {
            ...
      }
}
```

16.4 String Expansion

String expansion is a feature that allows strings to dynamically define their value at run time. This expansion is done via the following syntax:

```
%{string}
```

This feature is used to create policies that refer to concepts rather than to specific values. For example, a policy can be created that refers to the User-Name in a request, via:

```
%{User-Name}
```

This string expansion is done only for double-quoted strings and for the back-tick operator, as described above. Unlike other languages, there is no way to define new variables. All of the string expansions must refer to attributes that already exist or to modules that will return a string value.

16.4.1 Attribute References

Attributes in a list may be referenced via one of the following two syntaxes:

```
%{Attribute-Name}
```

```
%{<list>:Attribute-Name}
```

The `<list>:` prefix is optional. If given, it must be one of `request`, `reply`, `proxy-request`, `proxy-reply`, `coa`, `disconnect`, or `control`. If the `<list>:` prefix is omitted, then the `request` list is assumed.

For EAP methods with tunneled authentication sessions (i.e. PEAP and EAP-TTLS), the inner tunnel session can refer to a list for the outer session by prefixing the list name with `outer.`; for example, `outer.request`.

When a reference is encountered, the given list is examined for an attribute of the given name. If found, the variable reference in the string is replaced with the value of that attribute. Otherwise, the reference is replaced with an empty string.

Examples:

```
%{User-Name}
%{request:User-Name}          # same as above
%{reply:User-Name}
%{outer.request:User-Name}    # from inside of a TTLS/PEAP tunnel
```

Examples of using references inside of a string:

```
"Hello %{User-Name}"
"You, %{User-Name} are not allowed to use %{NAS-IP-Address}"
```

16.4.2 Character Escaping

Some characters need to be escaped within a dynamically expanded string %{...}. The % character is used for variable expansion, so a literal % character can be created by using %%.

Other than within a dynamically expanded string, very little character escaping is needed. The rules of the enclosing string context determine whether or not a space or " character needs to be escaped.

16.4.3 Pre-defined Expansions

In addition to storing attribute references, the server has a number of pre-defined references, as follows:

- `%{conf:...}`
  Refers to a variable in the configuration file. See the section for documentation on configuration file references.
- `%{client:...}`
  Refers to a variable that was defined in the client section for the current client.
- `%{listen:...}`
  Refers to a variable that was defined in the listen section that received the packet. This definition is only available in version 2.1.11 and later.
- `%{0}`
  Refers to the string that was last used to match a regular expression. The variables `%{1}` through `%{8}` refer to the matched substring in the regular expression.
- `%{md5:...}`
  Dynamically expands the string and performs an MD5 hash on it. The result is 32 hex digits.
- `%{Packet-Type}`
  The packet type (Access-Request, etc.)
- `%{Packet-SRC-IP-Address}` `%{Packet-SRC-IPv6-Address}`
  The source IPv4 or IPv6 address of the packet. See also the expansions `%{client:ipaddr}` and `%{client:ipv6addr}`. The two expansions should be identical, unless `%{client:ipaddr}` contains a DNS hostname.
- `%{Packet-DST-IP-Address}` `%{Packet-DST-IPv6-Address}`
  The destination IPv4 or IPv6 address of the packet. See also the expansions `%{listen:ipaddr}` and `%{listen:ipv6addr}`. If the socket is listening on a "wildcard" address, then these two expansions will be different, as follows: the `%{listen:ipaddr}` will be the wildcard address and `%{Packet-DST-IP-Address}` will be the unicast address to which the packet was sent.
- `%{Packet-SRC-Port}` `%{Packet-DST-Port}`
  The source/destination ports associated with the packet.
- `%{tolower:...}`
  Dynamically expands the string and returns the lowercase version of it. This definition is only available in version 2.1.10 and later.
- `%{toupper:...}`
  Dynamically expands the string and returns the uppercase version of it. This definition is only available in version 2.1.10 and later.

16.4.4 Single Letter Expansions

The following are single letter expansions. A number of additional single-letter expansions are not documented here as they are duplicates of the form `%{Attribute-Name}`.

- `%%`
  Returns `%`.
- `%d`
  Two-digit day of the month when the request was received.
- `%l`
  The Unix timestamp of when the request was received. This is an unsigned decimal number. It should be used with time-based calculations.
- `%m`
  Two-digit month describing when the request was received.
- `%t`
  Timestamp in `ctime` format.
- `%D`
  Request date (YYYYMMDD)
- `%H`
  Two digit hour of the day describing when the request was received.
- `%S`
  Request timestamp in SQL format, YYYY-mmm-ddd HH:MM:SS
- `%T`
  Request timestamp in database format, YYYY-mmm-ddd HH.MM.SS.000000
- `%Y`
  Four digit request year.

16.4.5 Module References

Individual modules may be referenced via the following syntax:

```
%{module:string}
```

These references are allowed only by a small number of modules that usually perform database lookups. The module name is the actual name of the module, as described earlier. The string portion is specific to each module and is not documented here. It is, however, usually dynamically expanded to allow for additional flexibility.

Examples:

```
%{sql:SELECT name FROM mytable WHERE username = %{User-Name}}
```

16.4.6 Conditional Syntax

Conditional syntax similar to that used in Unix shells may also be used:

```
%{%{Foo}:-bar}
```

This code returns the value of `%{Foo}`, if it has a value. Otherwise, it returns a literal string bar.

```
%{%{Foo}:-%{Bar}}
```

This code returns the value of `%{Foo}`, if it has a value. Otherwise, it returns the expansion of `%{Bar}`.

These conditional expansions can be nested to almost any depth, such as with `%{%{One}:-%{%{Two}:-%{Three}}}`.

16.4.7 String Manipulation

There are a number of other operators that manipulate the expanded string prior to returning a value.

String Length

```
%{#string}
```

The string length operator returns the number of characters in the given string as a decimal number. It can be used with attribute or module references. If the string has no value, then the length evaluates to zero.

Printing as Integers

```
%{Attribute-Name#}
```

Placing the hash character after an attribute name causes the attribute to be printed as an integer. In normal operation, `integer` attributes are printed using the name given by a `VALUE` statement in a dictionary. Similarly, date attributes are printed as dates, i.e., "January 1 2010".

This operator applies only to attributes of type `date`, `integer`, `byte`, and `short`. It has no effect on other attribute types and cannot be used with module references. It is most commonly used to perform calculations on dates, where the dates are treated as integers.

For example, if a request contains `Service-Type = Login-User`, the expansion of `%{Service-Type#}` will yield `1`, which is the value associated with the `Login-User` name. Using `%{Event-Timestamp#}` will return the event timestamp as an unsigned 32-bit decimal number.

Indexed Offsets

```
%{Attribute-Name[index]}
```

References the indexed occurrence of the given attribute. The indexes start at zero. This feature is NOT available for non-attribute dynamic translations, like `%{sql:...}`.

For example, `%{User-Name[0]}` is the same as `%{User-Name}`. The reference `%{Cisco-AVPair[2]}` will reference the value of the third `Cisco-AVPair` attribute (if it exists) in the request packet.

In practical usage, this kind of reference is of limited use. A plugin for a full language such as Perl or Python is recommended if this functionality is required.

Number of Attributes

```
%{Attribute-Name[#]}
```

Returns the total number of attributes of that name in the relevant attribute list. The number will usually be between 0 and 200.

For most requests, `%{User-Name[#]}` will have a value of 1. As with the previous section, this reference is of limited use.

All Named Attributes

```
%{Attribute-Name[*]}
```

Expands to a single string, with the value of each array member separated by a new line. This reference should be used with care, as the result may have dozens of lines of text. As with the previous section, this reference is of limited use.

Length of an Indexed Attribute

```
%{#Attribute-Name[index]}
```

Expands to the length of the string `%{Attribute-Name[index]}`. As with the previous section, this reference is of limited use.

16.5 Programming Statements

Unlang provides the ability to conditionally evaluate statements based on attributes in a request, in a response, or in data taken from a database.

16.5.1 The if Statement

```
if (condition) {
       [ statements ]
}
```

The primary conditional statement is the `if` statement. If `condition` evaluates to `true`, the statements within the subsection are executed. If `condition` evaluates to `false`, those statements are skipped.

16.5.2 The else Clause

```
if (condition) {
       [ statements ]
}
else {
       [ statements ]
}
```

An `if` statement can have an `else` clause. If `condition` evaluates to `false`, the statements in the `if` subsection are skipped and the statements within the `else` subsection are executed.

16.5.3 The elsif Clause

```
if (condition-1) {
       [ statements ]
}
elsif (condition-2) {
       [ statements ]
}
```

```
else {
      [ statements ]
}
```

A series of `if` statements can be used so that only one subsection is executed. In the preceding example, if `condition-1` evaluates to false, then the statements in the `if` subsection are skipped and `condition-2` is checked. If `condition-2` evaluates true, then the statements in the `elsif` subsection are executed. If `condition-2` evaluates false, then the statements in the `elsif` subsection are skipped, and the statements in the `else` subsection (if it exists) are executed.

An `elsif` clause does not need to be followed by an `else` clause. However, any `else` clause must be the last clause in the chain. An arbitrary number of `elsif` clauses can be chained together to create a series of conditional checks and statements.

16.5.4 The switch Statement

```
switch "string" {
      case string-1 {
            [ statements ]
      }
      case string-2 {
            [ statements ]
      }
      case {
            [ statements ]
      }
}
```

A `switch` statement causes the server to evaluate `string`, which may be dynamically expanded. The resulting fixed string is compared against `string-1` and `string-2` to find a match. If no string matches, the server looks for the default `case` statement, which has no associated string.

If a matching `case` is found, the statements within that subsection are executed. If no matching `case` is found, the `switch` statement does nothing.

Each string for the `case` statement must be constant. They are not expanded or evaluated at run time. No statement other than `case` can appear in a `switch` section, and the `case` statement cannot appear outside of a `switch` section.

16.6 Conditions/Conditional Expressions

Conditions are evaluated when parsing `if` and `elsif` statements. These conditions allow the server to make complex decisions based on one of a number of possible criteria.

16.6.1 Intermediate Operators

This section defines the operators that are used to return results to a conditional expression.

The ( ) Operator

```
( condition )
```

The `( )` operator returns the result of evaluating the given `condition`. It is used to clarify policies or to explicitly define conditional precedence.

Examples:

```
(foo)
(bar || (baz && dub))
```

The attribute-name Operator

```
attribute-name
```

If an `attribute-name` appears inside of a condition, the contents of that attribute are returned. If no such attribute exists, an empty value is returned.

Where the `attribute-name` is used in a condition as a test for existence, the condition evaluates to `true` if the named attribute exists. Otherwise, the condition evaluates to `false`.

Examples:

```
User-Name
NAS-IP-Address
```

The return-code Operator

```
return-code
```

The Unlang interpreter tracks the return code of any module that has been called. This return code can be checked in any condition. If the saved return code matches the `code` given here, the condition evaluates to `true`. Otherwise, it evaluates to `false`.

The list of valid return codes is as follows:

Valid Return Codes	
Name	**Meaning**
notfound	information was not found
noop	the module did nothing
ok	the module succeeded
updated	the module updated the request
fail	the module failed
reject	the module rejected the request
userlock	the user was locked out
invalid	the configuration was invalid
handled	the module has handled the request itself

Table 16.3.1 Module Return Codes

Examples:

```
if (notfound) {
```

```
        ...
}
```

The data Operator

```
data
```

Any text not matching `attribute-name` or `return-code` is interpreted as given previously in 16.1 Data Types on page 111. Where the `data` is used in a condition as a test for existence, the condition evaluates to `true` if the resulting string is not empty. Otherwise, it evaluates to `false`.

Examples:

```
foo
bar
```

16.6.2 Boolean Operators

This section defines the operators that return boolean values

The Existence Operator

```
(expression)
```

The existence operator returns the result of evaluating `expression`. Note that the language is not strongly typed, so the text "`0000`" can be interpreted as a data type "integer", having value zero, or a data type "string", having value "`0000`". This issue can be avoided by comparing strings to an empty string rather than by evaluating the string by itself.

Examples:

```
if (foo) {...
if (User-name) { ...
```

The Negation Operator

```
(!expression)
```

This operator returns `true` if `expression` evaluates to `false` and returns `false` if `expression` evaluates to `true`.

Examples:

```
if (!User-Name) { ...
```

The && Operator

```
(expression-1 && expression-2)
```

The `&&` operator performs a short-circuit "and" evaluation of the two expressions. This operator evaluates `expression-1` and returns `false` if `expression-1` returns `false`. Only if `expression-1` returns `true` is `expression-2` evaluated and its result returned.

Examples:

```
if (User-Name && EAP-Message) { ...
```

The || Operator

```
(expression-1 || expression-2)
```

The || operator performs a short-circuit "or" evaluation of the two expressions. This operator evaluates `expression-1` and returns `true` if `expression-1` returns true. Only if `expression-1` returns `false` is `expression-2` evaluated and its result returned.

Examples:

```
if (User-Name || NAS-IP-Address) { ...
```

The == Operator

```
(data-1 == data-2)
```

The == operator compares the result of evaluating `data-1` and `data-2`. As discussed in **16.1 Data Types** on page 111, the `data-1` field may be interpreted as a reference to an attribute.

The `data-2` field is interpreted in a type-specific manner. For example, if `data-1` refers to an attribute of type `ipaddr`, then `data-2` is evaluated as an IP address. If `data-1` refers to an attribute of type `integer`, then `data-2` is evaluated as an integer or as a named enumeration defined by a `VALUE` statement in a dictionary. Similarly, if `data-1` refers to an attribute of type `date`, `data-2` will be interpreted as a date string.

If the resulting data evaluates to be the same, then the operator returns `true`; otherwise, it returns `false`.

Examples:

```
if (User-Name == "bob") { ...
```

Other Comparison Operators

```
(data-1 < data-1)
(data-1 <= data-1)
(data-1 > data-1)
(data-1 >= data-1)
```

The other comparison operators act in a fashion similar to the == operator except that the result of the evaluation is the result of the given comparison. These operators are type-safe for integers, IP addresses, dates, and strings.

Examples:

```
if (reply:Session-Timeout < 3600) { ...
```

Regular Expression Operators

```
(data =~ /regex/)
(data !~ /regex/)
```

The regular expression operators perform regular expression matching on the data. The `data` field is interpreted as given above for the `==` operator. The `/regex/` field is interpreted as a regular expression.

The `/regex/` field is valid only after a regular-expression operator and is implemented by the local regular expression library on the system. These are usually Posix regular expressions.

A trailing `i` may be given, i.e., `/regex/i`. The trailing `i` indicates that the regular expression match should be done in a case-insensitive fashion.

When operator `=~` is used, then any parentheses in the regular expression will define variables that contain the matching text.

The special variable `%{0}` contains a copy of the result of evaluating the data field. The variables `%{1}` through `%{8}` will contain the substring matches, starting from the left-most parentheses. If there are more than eight parentheses, the additional results will be discarded and will not be placed into any variables.

Note that any `$` character will have to be escaped or doubled, owing to limitations of the parser.

Examples:

```
if (User-Name =~ /@example\.com$$/) { ...
```

16.7 Attribute Editing Statements

One of reasons to use Unlang is the ability to edit attributes. This capability allows policies to add, delete, or modify any attribute.

16.7.1 The update Statement

```
update <list> {
      [ attributes ]
}
```

The `update` statement adds attributes to or edits the attributes in a named `<list>`. The `<list>` should be one of `request`, `reply`, `proxy-request`, `proxy-reply`, `coa`, `disconnect`, or `control`.

For EAP methods with tunneled authentication sessions (i.e. PEAP and EAP-TTLS), the inner tunnel session can refer to a list for the outer session by prefixing the list name with `outer.` For example, `outer.request`.

The `update` statement can only be used with the `coa` and `disconnect` lists when the server receives an Access-Request or Accounting-Request. Updating the `coa` or `disconnect` list will cause the server to send a CoA-Request of Disconnect-Request packet to the NAS.

When the server receives a CoA-Request or Disconnect-Request packet, the update sections should reference the request and reply lists, as the server has received a request and will be sending a reply.

16.7.2 Attribute Statement

```
Attribute-Name operator value
```

The `attribute` statement can be used only with an `update` section. It allows for the addition, deletion, or editing of an attribute in the named list.

Each individual attribute and its associated value have to be placed in a single line in the policy. There is no need for commas or semi-colons after the value.

Examples:

```
User-Name = "bob"
Reply-Message += "hello"
```

Attribute Names

The `Attribute-Name`, described above in the Operator section, must be a name defined in a dictionary. If an undefined name is used, the server will return an error and will not start. The `Attribute-Name` must be a simple word, as described above in section **16.1 Data Types** on page 111.

The Unlang interpreter does not perform any string expansion on the name, so it is not possible to define names that refer to other attributes.

Examples:

```
User-Name
Reply-Message
```

Operators

The `operator` is used to define how the attribute is handled. Different operators allow attributes to be added, deleted, or replaced.

- `=` Add the attribute to the list, if and only if an attribute of the same name is not already present in that list.
- `:=` Add the attribute to the list. If any attribute of the same name is already present in that list, its value is replaced with the value of the current attribute.
- `+=` Add the attribute to the tail of the list, even if attributes of the same name are already present in the list.

Examples:

```
User-Name = "bob"
NAS-IP-Address := 192.0.2.17
Reply-Message += "hello"
```

Enforcement and Filtering Operators

The following operators may also be used in addition to the ones listed above. These operators use the `Attribute-Name` and `value` fields to enforce limits on all attributes in the list and to edit attributes with a matching `Attribute-Name`. All other attributes are ignored.

- `-=` Remove all attributes from the list that match the given value.
- `==` Keep only the attributes in the list that match the given value.

Note that this operator is very different from the = operator listed above. The = operator is used to add new attributes to the list, while the == operator removes all attributes that do not match the given value.

- `<` Keep only the attributes in the list that have values less than the value given herein. Any larger value is replaced by the value given. If no such `Attribute-Name` exists, the attribute is added with the value given here, as with the `+=` operator.
  This operator is valid only for attributes that can be represented as an integer.
- `<=` Keep only the attributes in the list that have values less than or equal to the value given here. Any larger value is replaced by the value given here. If no such `Attribute-Name` exists, the attribute is added with the value given here, as with the `+=` operator.
  This operator is valid only for attributes that can be represented as an integer.
- `>` Keep only the attributes in the list that have values greater than the value given here. Any smaller value is replaced by the value given here. If no such `Attribute-Name` exists, the attribute is added with the value given here, as with the `+=` operator.
  This operator is valid only for attributes that can be represented as an integer.
- `>=` Keep only the attributes in the list that have values greater than or equal to the value given here. Any smaller value is replaced by the value given here. If no such `Attribute-Name` exists, the attribute is added with the value given here, as with the `+=` operator.
  This operator is valid only for attributes that can be represented as an integer.
- `!*` Delete all occurrences of the named attribute, no matter what value they have.

Examples:

```
Session-Timeout <= 3600
Reply-Message != *
```

Values

The `value` field contains data as given above in **16.1 Data Types** on page 111. The format of the value is attribute-specific and is usually a string, integer, IP address, or other.

Prior to the attribute being instantiated, the value may be expanded as described above in section **16.1 Data Types** on page 111. The expansion of the value allows for greater flexibility; for example, an IP address value can be assigned to an attribute by specifying the IP address directly, by having the address returned from a database query, or by having the address returned as the output of an executed program.

When `string` values are assigned to an attribute, they can have a maximum length of 253 characters. Any extra data is silently discarded, causing the string to be truncated.

This truncating behavior is due to limitations of the RADIUS protocol, which can only transport strings of 253 characters or less; thus, attributes can only contain strings of that length. The Unlang parser has no

such limit, however, so strings within the configuration files can have nearly arbitrary length. For example, it is possible to write an SQL `SELECT` statement that is more than 1000 characters long. The result of the `SELECT` statement will, however, be truncated to 253 characters.

Examples:

```
0
192.0.2.16
'foo bar'
"hello"
"%{sql: SELECT ... }"
```

16.8 Configurable Failover

Configurable failover provides the ability to control exactly how the processing flows through a section and to override or rewrite the return value of any module. The Unlang interpreter provides for complicated failover. The return codes from each module or subsection can be modified or updated based on complex criteria.

This section describes how to create more complicated module processing and failover configurations, and is thus intended only for advanced users.

16.8.1 Normal Processing

In normal operation, the interpreter processes lists of statements, such as the example given below:

```
authorize {
      preprocess
      files
}
```

In the above configuration, when the `authorize` section is executed, the `preprocess` module is also called, followed by the `files` module. The processing is top to bottom and follows the algorithm given earlier in this chapter.

The configurable failover function allows for more flexibility when the interpreter processes a section. As stated previously, configurable failover provides the ability to control exactly how the processing flows through a section and to override or rewrite the return value of any module. This capability allows administrators to create policies such as "try `sql1`, if it's down, try `sql2`, otherwise pretend everything succeeded":

```
accounting {
      detail            # always log to detail, stopping if it fails
      redundant {
            sql1        # try module sql1
            sql2        # if that's down, try module sql2
            ok          # otherwise treat it as success
      }
}
```

In comparison to configurable failover, the Unlang interpreter is even more flexible.

16.8.2 Rewriting Return Code

Normally, when a module fails, the intepretor stops processing the current section and returns. In some cases, allowing an "early success" may be preferable, which means the server is instructed to stop processing the list on a success. This instruction is accomplished by adding a subsection after the module name. For example, rather than:

```
accounting {
      detail
      ...
}
```

the configuration would read as follows:

```
accounting {
      detail {
            ok = return
      }
      ...
}
```

In this example, the default action and the priority in the action table are replaced with the specified action and/or priority. The above configuration would cause the interpreter to stop processing the `accounting` section if the `detail` module returned `ok`.

On the other hand, if the goal is to have a "soft failure", then the interpreter should continue processing the `accounting` section even if the `detail` module returned `fail`. In that situation, the following configuration should be used:

```
accounting {
      detail {
            fail = 1
      }
      ...
}
```

The `fail = 1` text tells the interpetor to remember the `fail` code with priority `1` in place of the default action for `fail`, which is `return`. This configuration allows the `accounting` section to return `ok` if another module succeeds and to return `fail` only if all of the other modules return `fail`.

16.8.3 Overriding the Action Table

Any module return code can be over-written as described above. The syntax is:

```
return-code = value
```

The `return-code` can be any one of the previously defined module return codes (e.g. `ok`, `fail`, etc.), or the word `default` can be used to set values for all of the return codes that have not yet been specified.

The `value` can be one of:

- **decimal number** The priority associated with this return code. The number may have any value between 1 and 999999.
- **return** Stop processing the current list, and return the current code.
- **reject** Stop processing the current list, and return with a reject.

Examples:

```
files {
      notfound = 1
      noop = 2
      ok = 3
      updated = 4
      default = return # all other codes are set to "return"
}
files {
      default = return # all codes are set to "return"
      notfound = 1 # these settings over-ride the "return"
      noop = 2
      ok = 3
      updated = 4
}
```

16.8.4 Setting Actions for a Subsection

The actions may also be over-ridden for a subsection:

```
authorize {
      preprocess
      redundant {
            sql1
            sql2
            notfound = return
      }
      files
}
```

This configuration has a `redundant` block that over-rides the normal behavior. If both `sql1` and `sql2` gave the return code `notfound`, the redundant block would normally also return `notfound`. However, by setting the handling of `notfound` to return, the parent `authorize` section will see the `notfound = return` code from the redundant block and will stop processing the `authorize` section.

The above configuration therefore prevents the `files` module from being called if both of the `sql` modules fail to find the user.