Network Working Group
Request for Comments: 5229
Updates: 5228
Category: Standards Track

K. Homme
University of Oslo

January 2008

Sieve Email Filtering: Variables Extension

Status of This Memo

Abstract

In advanced mail filtering rule sets, it is useful to keep state or configuration details across rules. This document updates the Sieve filtering language (RFC 5228) with an extension to support variables. The extension changes the interpretation of strings, adds an action to store data in variables, and supplies a new test so that the value of a string can be examined.

1. Introduction

This is an extension to the Sieve language defined by [SIEVE]. It adds support for storing and referencing named data. The mechanisms detailed in this document will only apply to Sieve scripts that include a require clause for the "variables" extension. The require clauses themselves are not affected by this extension.

Conventions for notations are as in [SIEVE] section 1.1, including use of [KEYWORDS] and [ABNF]. The grammar builds on the grammar of [SIEVE]. In this document, "character" means a character from the ISO 10646 coded character set [ISO10646], which may consist of multiple octets coded in [UTF-8], and "variable" is a named reference to data stored or read back using the mechanisms of this extension.

2. Capability Identifier

The capability string associated with the extension defined in this document is "variables".

3. Interpretation of Strings

This extension changes the semantics of quoted-string, multi-line-literal and multi-line-dotstuff found in [SIEVE] to enable the inclusion of the value of variables.

When a string is evaluated, substrings matching variable-ref SHALL be replaced by the value of variable-name. Only one pass through the string SHALL be done. Variable names are case insensitive, so "foo" and "FOO" refer to the same variable. Unknown variables are replaced by the empty string.

```
variable-ref        =  "${" [namespace] variable-name "}"
namespace           =  identifier "." *sub-namespace
sub-namespace       =  variable-name "."
variable-name       =  num-variable / identifier
num-variable        =  1*DIGIT
```

Examples:

```
"&%${}!"     => unchanged, as the empty string is an illegal
                identifier
"${doh!}"    => unchanged, as "!" is illegal in identifiers
```

The variable "company" holds the value "ACME". No other variables are set.

```
"${full}"         => the empty string
"${company}"      => "ACME"
```

```
"${BAD${Company}" => "${BADACME"
"${President, ${Company} Inc.}"
                  => "${President, ACME Inc.}"
```

The expanded string MUST use the variable values that are current when control reaches the statement the string is part of.

Strings where no variable substitutions take place are referred to as constant strings. Future extensions may specify that passing non-constant strings as arguments to its actions or tests is an error.

Namespaces are meant for future extensions that make internal state available through variables. These variables SHOULD be put in a namespace whose first component is the same as its capability string. Such extensions SHOULD state which, if any, of the variables in its namespace are modifiable with the "set" action.

References to namespaces without a prior require statement for the relevant extension MUST cause an error.

Tests or actions in future extensions may need to access the unexpanded version of the string argument and, e.g., do the expansion after setting variables in its namespace. The design of the implementation should allow this.

3.1. Quoting and Encoded Characters

The semantics of quoting using backslash are not changed: backslash quoting is resolved before doing variable substitution. Similarly, encoded character processing (see Section 2.4.2.4 of [SIEVE]) is performed before doing variable substitution, but after quoting.

Examples:

```
"${fo\o}"  => ${foo}  => the expansion of variable foo.
"${fo\\o}" => ${fo\o} => illegal identifier => left verbatim.
"\${foo}"  => ${foo}  => the expansion of variable foo.
"\\${foo}" => \${foo} => a backslash character followed by the
                         expansion of variable foo.
```

If it is required to include a character sequence such as "${beep}" verbatim in a text literal, the user can define a variable to circumvent expansion to the empty string.

Example:

```
set "dollar" "$";
set "text" "regarding ${dollar}{beep}";
```

Example:

```
require ["encoded-character", "variables"];
set "name" "Ethelbert"
if header :contains "Subject" "dear${hex:20 24 7b 4e}ame}" {
    # the test string is "dear Ethelbert"
}
```

3.2. Match Variables

A "match variable" has a name consisting only of decimal digits and has no namespace component.

The decimal value of the match variable name will index the list of matching strings from the most recently evaluated successful match of type ":matches". The list is empty if no match has been successful.

Note: Extra leading zeroes are allowed and ignored.

The list will contain one string for each wildcard ("?" and "*") in the match pattern. Each string holds the substring from the source value that the corresponding wildcard expands to, possibly the empty string. The wildcards match as little as possible (non-greedy matching).

The first string in the list has index 1. If the index is out of range, the empty string will be substituted. Index 0 contains the matched part of the source value.

The interpreter MUST short-circuit tests, i.e., not perform more tests than necessary to find the result. Evaluation order MUST be left to right. If a test has two or more list arguments, the implementation is free to choose which to iterate over first.

An extension describing a new match type (e.g., [REGEX]) MAY specify that match variables are set as a side effect when the match type is used in a script that has enabled the "variables" extension.

Example:

```
require ["fileinto", "variables"];

if header :matches "List-ID" "*<*@*" {
    fileinto "INBOX.lists.${2}"; stop;
}
```

```
# Imagine the header
# Subject: [acme-users] [fwd] version 1.0 is out
if header :matches "Subject" "[*] *" {
    # ${1} will hold "acme-users",
    # ${2} will hold "[fwd] version 1.0 is out"
    fileinfo "INBOX.lists.${1}"; stop;
}

# Imagine the header
# To: coyote@ACME.Example.COM
if address :matches ["To", "Cc"] ["coyote@**.com",
        "wile@**.com"] {
    # ${0} is the matching address
    # ${1} is always the empty string
    # ${2} is part of the domain name ("ACME.Example")
    fileinto "INBOX.business.${2}"; stop;
} else {
    # Control wouldn't reach this block if any match was
    # successful, so no match variables are set at this
    # point.
}

if anyof (true, address :domain :matches "To" "*.com") {
    # The second test is never evaluated, so there are
    # still no match variables set.
    stop;
}
```

4. Action set

Usage: set [MODIFIER] <name: string> <value: string>

The "set" action stores the specified value in the variable identified by name. The name MUST be a constant string and conform to the syntax of variable-name. Match variables cannot be set. A namespace cannot be used unless an extension explicitly allows its use in "set". An invalid name MUST be detected as a syntax error.

Modifiers are applied on a value before it is stored in the variable. See the next section for details.

Variables are only visible to the currently running script. Note: Future extensions may provide different scoping rules for variables.

Variable names are case insensitive.

Example:

```
set "honorific"  "Mr";
set "first_name" "Wile";
set "last_name"  "Coyote";
set "vacation" text:
Dear ${HONORIFIC} ${last_name},
I'm out, please leave a message after the meep.
.
;
```

"set" does not affect the implicit keep. It is compatible with all actions defined in [SIEVE].

4.1. Modifiers

Usage: ":lower" / ":upper" / ":lowerfirst" / ":upperfirst" / ":quotewildcard" / ":length"

Modifier names are case insensitive. Unknown modifiers MUST yield a syntax error. More than one modifier can be specified, in which case they are applied according to this precedence list, largest value first:

Precedence	Modifier
40	:lower
	:upper
30	:lowerfirst
	:upperfirst
20	:quotewildcard
10	:length

It is an error to use two or more modifiers of the same precedence in a single "set" action.

Examples:

```
# The value assigned to the variable is printed after the arrow
set "a" "juMBlEd lETteRS";             => "juMBlEd lETteRS"
set :length "b" "${a}";                => "15"
set :lower "b" "${a}";                 => "jumbled letters"
set :upperfirst "b" "${a}";            => "JuMBlEd lETteRS"
set :upperfirst :lower "b" "${a}";     => "Jumbled letters"
set :quotewildcard "b" "Rock*";        => "Rock\*"
```

4.1.1. Modifier ":length"

The value is the decimal number of characters in the expansion, converted to a string.

4.1.2. Modifier ":quotewildcard"

This modifier adds the necessary quoting to ensure that the expanded text will only match a literal occurrence if used as a parameter to :matches. Every character with special meaning ("*", "?", and "\") is prefixed with "\" in the expansion.

4.1.3. Case Modifiers

These modifiers change the letters of the text from upper to lower case or vice versa. Characters other than "A"-"Z" and "a"-"z" from US-ASCII are left unchanged.

4.1.3.1. Modifier ":upper"

All lower case letters are converted to their upper case counterparts.

4.1.3.2. Modifier ":lower"

All upper case letters are converted to their lower case counterparts.

4.1.3.3. Modifier ":upperfirst"

The first character of the string is converted to upper case if it is a letter and set in lower case. The rest of the string is left unchanged.

4.1.3.4. Modifier ":lowerfirst"

The first character of the string is converted to lower case if it is a letter and set in upper case. The rest of the string is left unchanged.

5. Test string

Usage: string [MATCH-TYPE] [COMPARATOR]
<source: string-list> <key-list: string-list>

The "string" test evaluates to true if any of the source strings matches any key. The type of match defaults to ":is".

In the "string" test, both source and key-list are taken from the script, not the message, and whitespace stripping MUST NOT be done unless the script explicitly requests this through some future mechanism.

Example:

```
set "state" "${state} pending";
if string :matches " ${state} " "* pending *" {
    # the above test always succeeds
}
```

The "relational" extension [RELATIONAL] adds a match type called ":count". The count of a single string is 0 if it is the empty string, or 1 otherwise. The count of a string list is the sum of the counts of the member strings.

6. Implementation Limits

An implementation of this document MUST support at least 128 distinct variables. The supported length of variable names MUST be at least 32 characters. Each variable MUST be able to hold at least 4000 characters. Attempts to set the variable to a value larger than what the implementation supports SHOULD be reported as an error at compile-time if possible. If the attempt is discovered during run-time, the value SHOULD be truncated, and it MUST NOT be treated as an error.

Match variables ${1} through ${9} MUST be supported. References to higher indices than those the implementation supports MUST be treated as a syntax error, which SHOULD be discovered at compile-time.

7. Security Considerations

When match variables are used, and the author of the script isn't careful, strings can contain arbitrary values controlled by the sender of the mail.

Since values stored by "set" that exceed implementation limits are silently truncated, it's not appropriate to store large structures with security implications in variables.

The introduction of variables makes advanced decision making easier to write, but since no looping construct is provided, all Sieve scripts will terminate in an orderly manner.

Sieve filtering should not be relied on as a security measure against hostile mail messages. Sieve is designed to do simple, mostly static tests, and is not suitable for use as a spam or virus checker, where

the perpetrator has a motivation to vary the format of the mail in order to avoid filtering rules. See also [SPAMTEST].

8. IANA Considerations

The following template specifies the IANA registration of the variables Sieve extension specified in this document:

To: iana@iana.org
Subject: Registration of new Sieve extension

Capability name: variables
Description: Adds support for variables to the Sieve filtering language.
RFC number: RFC 5229
Contact address: The Sieve discussion list <ietf-mta-filters@imc.org>

9. Acknowledgments

Thanks to Cyrus Daboo, Jutta Degener, Ned Freed, Lawrence Greenfield, Jeffrey Hutzelman, Mark E. Mallett, Alexey Melnikov, Peder Stray, and Nigel Swinson for valuable feedback.

10. References

10.1. Normative References

[ABNF] Crocker, D., Ed., and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005.

[KEYWORDS] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RELATIONAL] Segmuller, W. and B. Leiba, "Sieve Email Filtering: Relational Extension", RFC 5231, January 2008.

[SIEVE] Guenther, P., Ed., and T. Showalter, Ed., "Sieve: An Email Filtering Language", RFC 5228, January 2008.

[UTF-8] Yergeau, F., "UTF-8, a transformation format of Unicode and ISO 10646", RFC 3629, November 2003.

10.2. Informative References

[ISO10646] ISO/IEC, "Information Technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane", May 1993, with amendments.

[REGEX] Murchison, K., "Sieve Email Filtering -- Regular Expression Extension", Work in Progress, February 2006.

[SPAMTEST] Daboo, C., "Sieve Email Filtering: Spamtest and Virustest Extensions", RFC 5235, January 2008.

Author's Address

Kjetil T. Homme
University of Oslo
PO Box 1080
0316 Oslo, Norway

Phone: +47 9366 0091
EMail: kjetilho@ifi.uio.no

Full Copyright Statement

Intellectual Property