Network Working Group
Request for Comments: 3028
Category: Standards Track

T. Showalter
Mirapoint, Inc.

January 2001

Sieve: A Mail Filtering Language

Status of this Memo

Copyright Notice

Abstract

This document describes a language for filtering e-mail messages at time of final delivery. It is designed to be implementable on either a mail client or mail server. It is meant to be extensible, simple, and independent of access protocol, mail architecture, and operating system. It is suitable for running on a mail server where users may not be allowed to execute arbitrary programs, such as on black box Internet Message Access Protocol (IMAP) servers, as it has no variables, loops, or ability to shell out to external programs.

Table of Contents

1. Introduction

This memo documents a language that can be used to create filters for electronic mail. It is not tied to any particular operating system or mail architecture. It requires the use of [IMAIL]-compliant messages, but should otherwise generalize to many systems.

The language is powerful enough to be useful but limited in order to allow for a safe server-side filtering system. The intention is to make it impossible for users to do anything more complex (and dangerous) than write simple mail filters, along with facilitating the use of GUIs for filter creation and manipulation. The language is not Turing-complete: it provides no way to write a loop or a function and variables are not provided.

Scripts written in Sieve are executed during final delivery, when the message is moved to the user-accessible mailbox. In systems where the MTA does final delivery, such as traditional Unix mail, it is reasonable to sort when the MTA deposits mail into the user's mailbox.

There are a number of reasons to use a filtering system. Mail traffic for most users has been increasing due to increased usage of e-mail, the emergence of unsolicited email as a form of advertising, and increased usage of mailing lists.

Experience at Carnegie Mellon has shown that if a filtering system is made available to users, many will make use of it in order to file messages from specific users or mailing lists. However, many others did not make use of the Andrew system's FLAMES filtering language [FLAMES] due to difficulty in setting it up.

Because of the expectation that users will make use of filtering if it is offered and easy to use, this language has been made simple enough to allow many users to make use of it, but rich enough that it can be used productively. However, it is expected that GUI-based editors will be the preferred way of editing filters for a large number of users.

1.1. Conventions Used in This Document

In the sections of this document that discuss the requirements of various keywords and operators, the following conventions have been adopted.

The key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", and "MAY" in this document are to be interpreted as defined in [KEYWORDS].

Each section on a command (test, action, or control structure) has a line labeled "Syntax:". This line describes the syntax of the command, including its name and its arguments. Required arguments are listed inside angle brackets ("<" and ">"). Optional arguments are listed inside square brackets ("[" and "]"). Each argument is followed by its type, so "<key: string>" represents an argument called "key" that is a string. Literal strings are represented with double-quoted strings. Alternatives are separated with slashes, and parenthesis are used for grouping, similar to [ABNF].

In the "Syntax" line, there are three special pieces of syntax that are frequently repeated, MATCH-TYPE, COMPARATOR, and ADDRESS-PART. These are discussed in sections 2.7.1, 2.7.3, and 2.7.4, respectively.

The formal grammar for these commands in section 10 and is the authoritative reference on how to construct commands, but the formal grammar does not specify the order, semantics, number or types of arguments to commands, nor the legal command names. The intent is to allow for extension without changing the grammar.

1.2. Example mail messages

The following mail messages will be used throughout this document in examples.

Message A

```
------------------------------------------------------------
Date: Tue, 1 Apr 1997 09:06:31 -0800 (PST)
From: coyote@desert.example.org
To: roadrunner@acme.example.com
Subject: I have a present for you

Look, I'm sorry about the whole anvil thing, and I really
didn't mean to try and drop it on you from the top of the
cliff.  I want to try to make it up to you.  I've got some
great birdseed over here at my place--top of the line
```

```
   stuff--and if you come by, I'll have it all wrapped up
   for you.  I'm really sorry for all the problems I've caused
   for you over the years, but I know we can work this out.
   --
   Wile E. Coyote   "Super Genius"   coyote@desert.example.org
   ------------------------------------------------------------

   Message B
   ------------------------------------------------------------
   From: youcouldberich!@reply-by-postal-mail.invalid
   Sender: b1ff@de.res.example.com
   To: rube@landru.example.edu
   Date:  Mon, 31 Mar 1997 18:26:10 -0800
   Subject: $$$ YOU, TOO, CAN BE A MILLIONAIRE! $$$

   YOU MAY HAVE ALREADY WON TEN MILLION DOLLARS, BUT I DOUBT
   IT!  SO JUST POST THIS TO SIX HUNDRED NEWSGROUPS!  IT WILL
   GUARANTEE THAT YOU GET AT LEAST FIVE RESPONSES WITH MONEY!
   MONEY! MONEY! COLD HARD CASH!  YOU WILL RECEIVE OVER
   $20,000 IN LESS THAN TWO MONTHS!  AND IT'S LEGAL!!!!!!!!!
   !!!!!!!!!!!!!!!!!!111111111!!!!!!!11111111111!!1  JUST
   SEND $5 IN SMALL, UNMARKED BILLS TO THE ADDRESSES BELOW!
   ------------------------------------------------------------
```

2. Design

2.1. Form of the Language

The language consists of a set of commands. Each command consists of a set of tokens delimited by whitespace. The command identifier is the first token and it is followed by zero or more argument tokens. Arguments may be literal data, tags, blocks of commands, or test commands.

The language is represented in UTF-8, as specified in [UTF-8].

Tokens in the ASCII range are considered case-insensitive.

2.2. Whitespace

Whitespace is used to separate tokens. Whitespace is made up of tabs, newlines (CRLF, never just CR or LF), and the space character. The amount of whitespace used is not significant.

2.3. Comments

Two types of comments are offered. Comments are semantically equivalent to whitespace and can be used anyplace that whitespace is (with one exception in multi-line strings, as described in the grammar).

Hash comments begin with a "#" character that is not contained within a string and continue until the next CRLF.

```
Example:  if size :over 100K { # this is a comment
             discard;
          }
```

Bracketed comments begin with the token "/*" and end with "*/" outside of a string. Bracketed comments may span multiple lines. Bracketed comments do not nest.

```
Example:  if size :over 100K { /* this is a comment
             this is still a comment */ discard /* this is a comment
             */ ;
          }
```

2.4. Literal Data

Literal data means data that is not executed, merely evaluated "as is", to be used as arguments to commands. Literal data is limited to numbers and strings.

2.4.1. Numbers

Numbers are given as ordinary decimal numbers. However, those numbers that have a tendency to be fairly large, such as message sizes, MAY have a "K", "M", or "G" appended to indicate a multiple of a power of two. To be comparable with the power-of-two-based versions of SI units that computers frequently use, K specifies kibi-, or 1,024 (2^10) times the value of the number; M specifies mebi-, or 1,048,576 (2^20) times the value of the number; and G specifies tebi-, or 1,073,741,824 (2^30) times the value of the number [BINARY-SI].

Implementations MUST provide 31 bits of magnitude in numbers, but MAY provide more.

Only positive integers are permitted by this specification.

2.4.2. Strings

Scripts involve large numbers of strings as they are used for pattern matching, addresses, textual bodies, etc. Typically, short quoted strings suffice for most uses, but a more convenient form is provided for longer strings such as bodies of messages.

A quoted string starts and ends with a single double quote (the <"> character, ASCII 34). A backslash ("\", ASCII 92) inside of a quoted string is followed by either another backslash or a double quote. This two-character sequence represents a single backslash or double-quote within the string, respectively.

No other characters should be escaped with a single backslash.

An undefined escape sequence (such as "\a" in a context where "a" has no special meaning) is interpreted as if there were no backslash (in this case, "\a" is just "a").

Non-printing characters such as tabs, CR and LF, and control characters are permitted in quoted strings. Quoted strings MAY span multiple lines. NUL (ASCII 0) is not allowed in strings.

For entering larger amounts of text, such as an email message, a multi-line form is allowed. It starts with the keyword "text:", followed by a CRLF, and ends with the sequence of a CRLF, a single period, and another CRLF. In order to allow the message to contain lines with a single-dot, lines are dot-stuffed. That is, when composing a message body, an extra '.' is added before each line which begins with a '.'. When the server interprets the script, these extra dots are removed. Note that a line that begins with a dot followed by a non-dot character is not interpreted dot-stuffed; that is, ".foo" is interpreted as ".foo". However, because this is potentially ambiguous, scripts SHOULD be properly dot-stuffed so such lines do not appear.

Note that a hashed comment or whitespace may occur in between the "text:" and the CRLF, but not within the string itself. Bracketed comments are not allowed here.

2.4.2.1. String Lists

When matching patterns, it is frequently convenient to match against groups of strings instead of single strings. For this reason, a list of strings is allowed in many tests, implying that if the test is true using any one of the strings, then the test is true. Implementations are encouraged to use short-circuit evaluation in these cases.

For instance, the test ‘header :contains ["To", "Cc"] ["me@example.com", "me00@landru.example.edu"]’ is true if either the To header or Cc header of the input message contains either of the e-mail addresses "me@example.com" or "me00@landru.example.edu".

Conversely, in any case where a list of strings is appropriate, a single string is allowed without being a member of a list: it is equivalent to a list with a single member. This means that the test ‘exists "To"’ is equivalent to the test ‘exists ["To"]’.

2.4.2.2. Headers

Headers are a subset of strings. In the Internet Message Specification [IMAIL] [RFC1123], each header line is allowed to have whitespace nearly anywhere in the line, including after the field name and before the subsequent colon. Extra spaces between the header name and the ":" in a header field are ignored.

A header name never contains a colon. The "From" header refers to a line beginning "From:" (or "From :", etc.). No header will match the string "From:" due to the trailing colon.

Folding of long header lines (as described in [IMAIL] 3.4.8) is removed prior to interpretation of the data. The folding syntax (the CRLF that ends a line plus any leading whitespace at the beginning of the next line that indicates folding) are interpreted as if they were a single space.

2.4.2.3. Addresses

A number of commands call for email addresses, which are also a subset of strings. When these addresses are used in outbound contexts, addresses must be compliant with [IMAIL], but are further constrained. Using the symbols defined in [IMAIL], section 6.1, the syntax of an address is:

```
sieve-address = addr-spec                ; simple address
              / phrase "<" addr-spec ">" ; name & addr-spec
```

That is, routes and group syntax are not permitted. If multiple addresses are required, use a string list. Named groups are not used here.

Implementations MUST ensure that the addresses are syntactically valid, but need not ensure that they actually identify an email recipient.

2.4.2.4. MIME Parts

In a few places, [MIME] body parts are represented as strings. These parts include MIME headers and the body. This provides a way of embedding typed data within a Sieve script so that, among other things, character sets other than UTF-8 can be used for output messages.

2.5. Tests

Tests are given as arguments to commands in order to control their actions. In this document, tests are given to if/elsif/else to decide which block of code is run.

Tests MUST NOT have side effects. That is, a test cannot affect the state of the filter or message. No tests in this specification have side effects, and side effects are forbidden in extension tests as well.

The rationale for this is that tests with side effects impair readability and maintainability and are difficult to represent in a graphic interface for generating scripts. Side effects are confined to actions where they are clearer.

2.5.1. Test Lists

Some tests ("allof" and "anyof", which implement logical "and" and logical "or", respectively) may require more than a single test as an argument. The test-list syntax element provides a way of grouping tests.

```
Example:  if anyof (not exists ["From", "Date"],
                    header :contains "from" "fool@example.edu") {
             discard;
          }
```

2.6. Arguments

In order to specify what to do, most commands take arguments. There are three types of arguments: positional, tagged, and optional.

2.6.1. Positional Arguments

Positional arguments are given to a command which discerns their meaning based on their order. When a command takes positional arguments, all positional arguments must be supplied and must be in the order prescribed.

2.6.2. Tagged Arguments

This document provides for tagged arguments in the style of CommonLISP. These are also similar to flags given to commands in most command-line systems.

A tagged argument is an argument for a command that begins with ":" followed by a tag naming the argument, such as ":contains". This argument means that zero or more of the next tokens have some particular meaning depending on the argument. These next tokens may be numbers or strings but they are never blocks.

Tagged arguments are similar to positional arguments, except that instead of the meaning being derived from the command, it is derived from the tag.

Tagged arguments must appear before positional arguments, but they may appear in any order with other tagged arguments. For simplicity of the specification, this is not expressed in the syntax definitions with commands, but they still may be reordered arbitrarily provided they appear before positional arguments. Tagged arguments may be mixed with optional arguments.

To simplify this specification, tagged arguments SHOULD NOT take tagged arguments as arguments.

2.6.3. Optional Arguments

Optional arguments are exactly like tagged arguments except that they may be left out, in which case a default value is implied. Because optional arguments tend to result in shorter scripts, they have been used far more than tagged arguments.

One particularly noteworthy case is the ":comparator" argument, which allows the user to specify which [ACAP] comparator will be used to compare two strings, since different languages may impose different orderings on UTF-8 [UTF-8] characters.

2.6.4. Types of Arguments

Abstractly, arguments may be literal data, tests, or blocks of commands. In this way, an "if" control structure is merely a command that happens to take a test and a block as arguments and may execute the block of code.

However, this abstraction is ambiguous from a parsing standpoint. The grammar in section 9.2 presents a parsable version of this: Arguments are string-lists, numbers, and tags, which may be followed

by a test or a test-list, which may be followed by a block of commands. No more than one test or test list, nor more than one block of commands, may be used, and commands that end with blocks of commands do not end with semicolons.

2.7. String Comparison

When matching one string against another, there are a number of ways of performing the match operation. These are accomplished with three types of matches: an exact match, a substring match, and a wildcard glob-style match. These are described below.

In order to provide for matches between character sets and case insensitivity, Sieve borrows ACAP's comparator registry.

However, when a string represents the name of a header, the comparator is never user-specified. Header comparisons are always done with the "i;ascii-casemap" operator, i.e., case-insensitive comparisons, because this is the way things are defined in the message specification [IMAIL].

2.7.1. Match Type

There are three match types describing the matching used in this specification: ":is", ":contains", and ":matches". Match type arguments are supplied to those commands which allow them to specify what kind of match is to be performed.

These are used as tagged arguments to tests that perform string comparison.

The ":contains" match type describes a substring match. If the value argument contains the key argument as a substring, the match is true. For instance, the string "frobnitzm" contains "frob" and "nit", but not "fbm". The null key ("") is contained in all values.

The ":is" match type describes an absolute match; if the contents of the first string are absolutely the same as the contents of the second string, they match. Only the string "frobnitzm" is the string "frobnitzm". The null key ":is" and only ":is" the null value.

The ":matches" version specifies a wildcard match using the characters "*" and "?". "*" matches zero or more characters, and "?" matches a single character. "?" and "*" may be escaped as "\\?" and "*" in strings to match against themselves. The first backslash escapes the second backslash; together, they escape the "*". This is awkward, but it is commonplace in several programming languages that use globs and regular expressions.

In order to specify what type of match is supposed to happen, commands that support matching take optional tagged arguments ":matches", ":is", and ":contains". Commands default to using ":is" matching if no match type argument is supplied. Note that these modifiers may interact with comparators; in particular, some comparators are not suitable for matching with ":contains" or ":matches". It is an error to use a comparator with ":contains" or ":matches" that is not compatible with it.

It is an error to give more than one of these arguments to a given command.

For convenience, the "MATCH-TYPE" syntax element is defined here as follows:

Syntax: ":is" / ":contains" / ":matches"

2.7.2. Comparisons Across Character Sets

All Sieve scripts are represented in UTF-8, but messages may involve a number of character sets. In order for comparisons to work across character sets, implementations SHOULD implement the following behavior:

> Implementations decode header charsets to UTF-8. Two strings are considered equal if their UTF-8 representations are identical. Implementations should decode charsets represented in the forms specified by [MIME] for both message headers and bodies. Implementations must be capable of decoding US-ASCII, ISO-8859-1, the ASCII subset of ISO-8859-* character sets, and UTF-8.

If implementations fail to support the above behavior, they MUST conform to the following:

> No two strings can be considered equal if one contains octets greater than 127.

2.7.3. Comparators

In order to allow for language-independent, case-independent matches, the match type may be coupled with a comparator name. Comparators are described for [ACAP]; a registry is defined for ACAP, and this specification uses that registry.

ACAP defines multiple comparator types. Only equality types are used in this specification.

All implementations MUST support the "i;octet" comparator (simply compares octets) and the "i;ascii-casemap" comparator (which treats uppercase and lowercase characters in the ASCII subset of UTF-8 as the same). If left unspecified, the default is "i;ascii-casemap".

Some comparators may not be usable with substring matches; that is, they may only work with ":is". It is an error to try and use a comparator with ":matches" or ":contains" that is not compatible with it.

A comparator is specified by the ":comparator" option with commands that support matching. This option is followed by a string providing the name of the comparator to be used. For convenience, the syntax of a comparator is abbreviated to "COMPARATOR", and (repeated in several tests) is as follows:

Syntax: ":comparator" <comparator-name: string>

So in this example,

```
Example:  if header :contains :comparator "i;octet" "Subject"
             "MAKE MONEY FAST" {
                discard;
          }
```

would discard any message with subjects like "You can MAKE MONEY FAST", but not "You can Make Money Fast", since the comparator used is case-sensitive.

Comparators other than i;octet and i;ascii-casemap must be declared with require, as they are extensions. If a comparator declared with require is not known, it is an error, and execution fails. If the comparator is not declared with require, it is also an error, even if the comparator is supported. (See 2.10.5.)

Both ":matches" and ":contains" match types are compatible with the "i;octet" and "i;ascii-casemap" comparators and may be used with them.

It is an error to give more than one of these arguments to a given command.

2.7.4. Comparisons Against Addresses

Addresses are one of the most frequent things represented as strings. These are structured, and being able to compare against the local-part or the domain of an address is useful, so some tests that act

exclusively on addresses take an additional optional argument that specifies what the test acts on.

These optional arguments are ":localpart", ":domain", and ":all", which act on the local-part (left-side), the domain part (right-side), and the whole address.

The kind of comparison done, such as whether or not the test done is case-insensitive, is specified as a comparator argument to the test.

If an optional address-part is omitted, the default is ":all".

It is an error to give more than one of these arguments to a given command.

For convenience, the "ADDRESS-PART" syntax element is defined here as follows:

Syntax: ":localpart" / ":domain" / ":all"

2.8. Blocks

Blocks are sets of commands enclosed within curly braces. Blocks are supplied to commands so that the commands can implement control commands.

A control structure is a command that happens to take a test and a block as one of its arguments; depending on the result of the test supplied as another argument, it runs the code in the block some number of times.

With the commands supplied in this memo, there are no loops. The control structures supplied--if, elsif, and else--run a block either once or not at all. So there are two arguments, the test and the block.

2.9. Commands

Sieve scripts are sequences of commands. Commands can take any of the tokens above as arguments, and arguments may be either tagged or positional arguments. Not all commands take all arguments.

There are three kinds of commands: test commands, action commands, and control commands.

The simplest is an action command. An action command is an identifier followed by zero or more arguments, terminated by a semicolon. Action commands do not take tests or blocks as arguments.

A control command is similar, but it takes a test as an argument, and ends with a block instead of a semicolon.

A test command is used as part of a control command. It is used to specify whether or not the block of code given to the control command is executed.

2.10. Evaluation

2.10.1. Action Interaction

Some actions cannot be used with other actions because the result would be absurd. These restrictions are noted throughout this memo.

Extension actions MUST state how they interact with actions defined in this specification.

2.10.2. Implicit Keep

Previous experience with filtering systems suggests that cases tend to be missed in scripts. To prevent errors, Sieve has an "implicit keep".

An implicit keep is a keep action (see 4.4) performed in absence of any action that cancels the implicit keep.

An implicit keep is performed if a message is not written to a mailbox, redirected to a new address, or explicitly thrown out. That is, if a fileinto, a keep, a redirect, or a discard is performed, an implicit keep is not.

Some actions may be defined to not cancel the implicit keep. These actions may not directly affect the delivery of a message, and are used for their side effects. None of the actions specified in this document meet that criteria, but extension actions will.

For instance, with any of the short messages offered above, the following script produces no actions.

Example: if size :over 500K { discard; }

As a result, the implicit keep is taken.

2.10.3. Message Uniqueness in a Mailbox

Implementations SHOULD NOT deliver a message to the same folder more than once, even if a script explicitly asks for a message to be written to a mailbox twice.

The test for equality of two messages is implementation-defined.

If a script asks for a message to be written to a mailbox twice, it MUST NOT be treated as an error.

2.10.4. Limits on Numbers of Actions

Site policy MAY limit numbers of actions taken and MAY impose restrictions on which actions can be used together. In the event that a script hits a policy limit on the number of actions taken for a particular message, an error occurs.

Implementations MUST prohibit more than one reject.

Implementations MUST allow at least one keep or one fileinto. If fileinto is not implemented, implementations MUST allow at least one keep.

Implementations SHOULD prohibit reject when used with other actions.

2.10.5. Extensions and Optional Features

Because of the differing capabilities of many mail systems, several features of this specification are optional. Before any of these extensions can be executed, they must be declared with the "require" action.

If an extension is not enabled with "require", implementations MUST treat it as if they did not support it at all.

If a script does not understand an extension declared with require, the script must not be used at all. Implementations MUST NOT execute scripts which require unknown capability names.

Note: The reason for this restriction is that prior experiences with languages such as LISP and Tcl suggest that this is a workable way of noting that a given script uses an extension.

Experience with PostScript suggests that mechanisms that allow a script to work around missing extensions are not used in practice.

Extensions which define actions MUST state how they interact with actions discussed in the base specification.

2.10.6. Errors

In any programming language, there are compile-time and run-time errors.

Compile-time errors are ones in syntax that are detectable if a syntax check is done.

Run-time errors are not detectable until the script is run. This includes transient failures like disk full conditions, but also includes issues like invalid combinations of actions.

When an error occurs in a Sieve script, all processing stops.

Implementations MAY choose to do a full parse, then evaluate the script, then do all actions. Implementations might even go so far as to ensure that execution is atomic (either all actions are executed or none are executed).

Other implementations may choose to parse and run at the same time. Such implementations are simpler, but have issues with partial failure (some actions happen, others don't).

Implementations might even go so far as to ensure that scripts can never execute an invalid set of actions (e.g., reject + fileinto) before execution, although this could involve solving the Halting Problem.

This specification allows any of these approaches. Solving the Halting Problem is considered extra credit.

When an error happens, implementations MUST notify the user that an error occurred, which actions (if any) were taken, and do an implicit keep.

2.10.7. Limits on Execution

Implementations may limit certain constructs. However, this specification places a lower bound on some of these limits.

Implementations MUST support fifteen levels of nested blocks.

Implementations MUST support fifteen levels of nested test lists.

3. Control Commands

Control structures are needed to allow for multiple and conditional actions.

3.1. Control Structure If

There are three pieces to if: "if", "elsif", and "else". Each is actually a separate command in terms of the grammar. However, an elsif MUST only follow an if, and an else MUST follow only either an if or an elsif. An error occurs if these conditions are not met.

Syntax: if <test1: test> <block1: block>

Syntax: elsif <test2: test> <block2: block>

Syntax: else <block>

The semantics are similar to those of any of the many other programming languages these control commands appear in. When the interpreter sees an "if", it evaluates the test associated with it. If the test is true, it executes the block associated with it.

If the test of the "if" is false, it evaluates the test of the first "elsif" (if any). If the test of "elsif" is true, it runs the elsif's block. An elsif may be followed by an elsif, in which case, the interpreter repeats this process until it runs out of elsifs.

When the interpreter runs out of elsifs, there may be an "else" case. If there is, and none of the if or elsif tests were true, the interpreter runs the else case.

This provides a way of performing exactly one of the blocks in the chain.

In the following example, both Message A and B are dropped.

Example:
```
require "fileinto";
if header :contains "from" "coyote" {
   discard;
} elsif header :contains ["subject"] ["$$$"] {
   discard;
} else {
   fileinto "INBOX";
}
```

When the script below is run over message A, it redirects the message to acm@example.edu; message B, to postmaster@example.edu; any other message is redirected to field@example.edu.

```
Example:  if header :contains ["From"] ["coyote"] {
             redirect "acm@example.edu";
          } elsif header :contains "Subject" "$$$" {
             redirect "postmaster@example.edu";
          } else {
             redirect "field@example.edu";
          }
```

Note that this definition prohibits the "... else if ..." sequence used by C. This is intentional, because this construct produces a shift-reduce conflict.

3.2. Control Structure Require

Syntax: require <capabilities: string-list>

The require action notes that a script makes use of a certain extension. Such a declaration is required to use the extension, as discussed in section 2.10.5. Multiple capabilities can be declared with a single require.

The require command, if present, MUST be used before anything other than a require can be used. An error occurs if a require appears after a command other than require.

```
Example:  require ["fileinto", "reject"];
```

```
Example:  require "fileinto";
          require "vacation";
```

3.3. Control Structure Stop

Syntax: stop

The "stop" action ends all processing. If no actions have been executed, then the keep action is taken.

4. Action Commands

This document supplies five actions that may be taken on a message: keep, fileinto, redirect, reject, and discard.

Implementations MUST support the "keep", "discard", and "redirect" actions.

Implementations SHOULD support "reject" and "fileinto".

Implementations MAY limit the number of certain actions taken (see section 2.10.4).

4.1. Action reject

Syntax: reject <reason: string>

The optional "reject" action refuses delivery of a message by sending back an [MDN] to the sender. It resends the message to the sender, wrapping it in a "reject" form, noting that it was rejected by the recipient. In the following script, message A is rejected and returned to the sender.

```
Example:  if header :contains "from" "coyote@desert.example.org" {
             reject "I am not taking mail from you, and I don't want
             your birdseed, either!";
          }
```

A reject message MUST take the form of a failure MDN as specified by [MDN]. The human-readable portion of the message, the first component of the MDN, contains the human readable message describing the error, and it SHOULD contain additional text alerting the original sender that mail was refused by a filter. This part of the MDN might appear as follows:

```
---------------------------------------------------------------
Message was refused by recipient's mail filtering program.  Reason
given was as follows:

I am not taking mail from you, and I don't want your birdseed,
either!
---------------------------------------------------------------
```

The MDN action-value field as defined in the MDN specification MUST be "deleted" and MUST have the MDN-sent-automatically and automatic-action modes set.

Because some implementations can not or will not implement the reject command, it is optional. The capability string to be used with the require command is "reject".

4.2. Action fileinto

Syntax: fileinto <folder: string>

The "fileinto" action delivers the message into the specified folder. Implementations SHOULD support fileinto, but in some environments this may be impossible.

The capability string for use with the require command is "fileinto".

In the following script, message A is filed into folder "INBOX.harassment".

```
Example:  require "fileinto";
          if header :contains ["from"] "coyote" {
             fileinto "INBOX.harassment";
          }
```

4.3. Action redirect

Syntax: redirect <address: string>

The "redirect" action is used to send the message to another user at a supplied address, as a mail forwarding feature does. The "redirect" action makes no changes to the message body or existing headers, but it may add new headers. The "redirect" modifies the envelope recipient.

The redirect command performs an MTA-style "forward"--that is, what you get from a .forward file using sendmail under UNIX. The address on the SMTP envelope is replaced with the one on the redirect command and the message is sent back out. (This is not an MUA-style forward, which creates a new message with a different sender and message ID, wrapping the old message in a new one.)

A simple script can be used for redirecting all mail:

```
Example:  redirect "bart@example.edu";
```

Implementations SHOULD take measures to implement loop control, possibly including adding headers to the message or counting received headers. If an implementation detects a loop, it causes an error.

4.4. Action keep

Syntax: keep

The "keep" action is whatever action is taken in lieu of all other actions, if no filtering happens at all; generally, this simply means to file the message into the user’s main mailbox. This command provides a way to execute this action without needing to know the name of the user’s main mailbox, providing a way to call it without needing to understand the user’s setup, or the underlying mail system.

For instance, in an implementation where the IMAP server is running scripts on behalf of the user at time of delivery, a keep command is equivalent to a fileinto "INBOX".

Example: if size :under 1M { keep; } else { discard; }

Note that the above script is identical to the one below.

Example: if not size :under 1M { discard; }

4.5. Action discard

Syntax: discard

Discard is used to silently throw away the message. It does so by simply canceling the implicit keep. If discard is used with other actions, the other actions still happen. Discard is compatible with all other actions. (For instance fileinto+discard is equivalent to fileinto.)

Discard MUST be silent; that is, it MUST NOT return a non-delivery notification of any kind ([DSN], [MDN], or otherwise).

In the following script, any mail from "idiot@example.edu" is thrown out.

```
Example: if header :contains ["from"] ["idiot@example.edu"] {
            discard;
         }
```

While an important part of this language, "discard" has the potential to create serious problems for users: Students who leave themselves logged in to an unattended machine in a public computer lab may find their script changed to just "discard". In order to protect users in this situation (along with similar situations), implementations MAY keep messages destroyed by a script for an indefinite period, and MAY disallow scripts that throw out all mail.

5. Test Commands

Tests are used in conditionals to decide which part(s) of the conditional to execute.

Implementations MUST support these tests: "address", "allof", "anyof", "exists", "false", "header", "not", "size", and "true".

Implementations SHOULD support the "envelope" test.

5.1. Test address

Syntax: address [ADDRESS-PART] [COMPARATOR] [MATCH-TYPE]
<header-list: string-list> <key-list: string-list>

The address test matches Internet addresses in structured headers that contain addresses. It returns true if any header contains any key in the specified part of the address, as modified by the comparator and the match keyword.

Like envelope and header, this test returns true if any combination of the header-list and key-list arguments match.

Internet email addresses [IMAIL] have the somewhat awkward characteristic that the local-part to the left of the at-sign is considered case sensitive, and the domain-part to the right of the at-sign is case insensitive. The "address" command does not deal with this itself, but provides the ADDRESS-PART argument for allowing users to deal with it.

The address primitive never acts on the phrase part of an email address, nor on comments within that address. It also never acts on group names, although it does act on the addresses within the group construct.

Implementations MUST restrict the address test to headers that contain addresses, but MUST include at least From, To, Cc, Bcc, Sender, Resent-From, Resent-To, and SHOULD include any other header that utilizes an "address-list" structured header body.

```
Example:  if address :is :all "from" "tim@example.com" {
             discard;
```

5.2. Test allof

Syntax: allof <tests: test-list>

The allof test performs a logical AND on the tests supplied to it.

```
Example:  allof (false, false)  =>   false
          allof (false, true)   =>   false
          allof (true,  true)   =>   true
```

The allof test takes as its argument a test-list.

5.3. Test anyof

Syntax: anyof <tests: test-list>

The anyof test performs a logical OR on the tests supplied to it.

```
Example:  anyof (false, false)  =>   false
          anyof (false, true)   =>   true
          anyof (true,  true)   =>   true
```

5.4. Test envelope

Syntax: envelope [COMPARATOR] [ADDRESS-PART] [MATCH-TYPE]
<envelope-part: string-list> <key-list: string-list>

The "envelope" test is true if the specified part of the SMTP (or equivalent) envelope matches the specified key.

If one of the envelope-part strings is (case insensitive) "from", then matching occurs against the FROM address used in the SMTP MAIL command.

If one of the envelope-part strings is (case insensitive) "to", then matching occurs against the TO address used in the SMTP RCPT command that resulted in this message getting delivered to this user. Note that only the most recent TO is available, and only the one relevant to this user.

The envelope-part is a string list and may contain more than one parameter, in which case all of the strings specified in the key-list are matched against all parts given in the envelope-part list.

Like address and header, this test returns true if any combination of the envelope-part and key-list arguments is true.

All tests against envelopes MUST drop source routes.

If the SMTP transaction involved several RCPT commands, only the data from the RCPT command that caused delivery to this user is available in the "to" part of the envelope.

If a protocol other than SMTP is used for message transport, implementations are expected to adapt this command appropriately.

The envelope command is optional. Implementations SHOULD support it, but the necessary information may not be available in all cases.

```
Example:  require "envelope";
          if envelope :all :is "from" "tim@example.com" {
             discard;
          }
```

5.5. Test exists

Syntax: exists <header-names: string-list>

The "exists" test is true if the headers listed in the header-names argument exist within the message. All of the headers must exist or the test is false.

The following example throws out mail that doesn't have a From header and a Date header.

```
Example:  if not exists ["From","Date"] {
             discard;
          }
```

5.6. Test false

Syntax: false

The "false" test always evaluates to false.

5.7. Test header

Syntax: header [COMPARATOR] [MATCH-TYPE]
<header-names: string-list> <key-list: string-list>

The "header" test evaluates to true if any header name matches any key. The type of match is specified by the optional match argument, which defaults to ":is" if not specified, as specified in section 2.6.

Like address and envelope, this test returns true if any combination of the string-list and key-list arguments match.

If a header listed in the header-names argument exists, it contains the null key (""). However, if the named header is not present, it does not contain the null key. So if a message contained the header

```
X-Caffeine: C8H10N4O2
```

these tests on that header evaluate as follows:

```
header :is ["X-Caffeine"] [""]         => false
header :contains ["X-Caffeine"] [""]   => true
```

5.8. Test not

Syntax: not <test>

The "not" test takes some other test as an argument, and yields the opposite result. "not false" evaluates to "true" and "not true" evaluates to "false".

5.9. Test size

Syntax: size <":over" / ":under"> <limit: number>

The "size" test deals with the size of a message. It takes either a tagged argument of ":over" or ":under", followed by a number representing the size of the message.

If the argument is ":over", and the size of the message is greater than the number provided, the test is true; otherwise, it is false.

If the argument is ":under", and the size of the message is less than the number provided, the test is true; otherwise, it is false.

Exactly one of ":over" or ":under" must be specified, and anything else is an error.

The size of a message is defined to be the number of octets from the initial header until the last character in the message body.

Note that for a message that is exactly 4,000 octets, the message is neither ":over" 4000 octets or ":under" 4000 octets.

5.10. Test true

Syntax: true

The "true" test always evaluates to true.

6. Extensibility

New control structures, actions, and tests can be added to the language. Sites must make these features known to their users; this document does not define a way to discover the list of extensions supported by the server.

Any extensions to this language MUST define a capability string that uniquely identifies that extension. If a new version of an extension changes the functionality of a previously defined extension, it MUST use a different name.

In a situation where there is a submission protocol and an extension advertisement mechanism aware of the details of this language, scripts submitted can be checked against the mail server to prevent use of an extension that the server does not support.

Extensions MUST state how they interact with constraints defined in section 2.10, e.g., whether they cancel the implicit keep, and which actions they are compatible and incompatible with.

6.1. Capability String

Capability strings are typically short strings describing what capabilities are supported by the server.

Capability strings beginning with "vnd." represent vendor-defined extensions. Such extensions are not defined by Internet standards or RFCs, but are still registered with IANA in order to prevent conflicts. Extensions starting with "vnd." SHOULD be followed by the name of the vendor and product, such as "vnd.acme.rocket-sled".

The following capability strings are defined by this document:

envelope The string "envelope" indicates that the implementation supports the "envelope" command.

fileinto The string "fileinto" indicates that the implementation supports the "fileinto" command.

reject The string "reject" indicates that the implementation supports the "reject" command.

comparator- The string "comparator-elbonia" is provided if the implementation supports the "elbonia" comparator. Therefore, all implementations have at least the "comparator-i;octet" and "comparator-i;ascii-casemap" capabilities. However, these comparators may be used without being declared with require.

6.2. IANA Considerations

In order to provide a standard set of extensions, a registry is provided by IANA. Capability names may be registered on a first-come, first-served basis. Extensions designed for interoperable use SHOULD be defined as standards track or IESG approved experimental RFCs.

6.2.1. Template for Capability Registrations

The following template is to be used for registering new Sieve extensions with IANA.

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

Capability name:
Capability keyword:
Capability arguments:
Standards Track/IESG-approved experimental RFC number:
Person and email address to contact for further information:

6.2.2. Initial Capability Registrations

The following are to be added to the IANA registry for Sieve extensions as the initial contents of the capability registry.

Capability name: fileinto
Capability keyword: fileinto
Capability arguments: fileinto <folder: string>
Standards Track/IESG-approved experimental RFC number:
RFC 3028 (Sieve base spec)
Person and email address to contact for further information:
Tim Showalter
tjs@mirapoint.com

Capability name: reject
Capability keyword: reject
Capability arguments: reject <reason: string>
Standards Track/IESG-approved experimental RFC number:
RFC 3028 (Sieve base spec)
Person and email address to contact for further information:
Tim Showalter
tjs@mirapoint.com

```
Capability name:        envelope
Capability keyword:     envelope
Capability arguments:
        envelope [COMPARATOR] [ADDRESS-PART] [MATCH-TYPE]
        <envelope-part: string-list> <key-list: string-list>
Standards Track/IESG-approved experimental RFC number:
        RFC 3028 (Sieve base spec)
Person and email address to contact for further information:
        Tim Showalter
        tjs@mirapoint.com

Capability name:        comparator-*
Capability keyword:
        comparator-* (anything starting with "comparator-")
Capability arguments:   (none)
Standards Track/IESG-approved experimental RFC number:
        RFC 3028, Sieve, by reference of
        RFC 2244, Application Configuration Access Protocol
Person and email address to contact for further information:
        Tim Showalter
        tjs@mirapoint.com
```

6.3. Capability Transport

As the range of mail systems that this document is intended to apply to is quite varied, a method of advertising which capabilities an implementation supports is difficult due to the wide range of possible implementations. Such a mechanism, however, should have property that the implementation can advertise the complete set of extensions that it supports.

7. Transmission

The MIME type for a Sieve script is "application/sieve".

The registration of this type for RFC 2048 requirements is as follows:

```
 Subject: Registration of MIME media type application/sieve

 MIME media type name: application
 MIME subtype name: sieve
 Required parameters: none
 Optional parameters: none
 Encoding considerations: Most sieve scripts will be textual,
    written in UTF-8.  When non-7bit characters are used,
    quoted-printable is appropriate for transport systems
    that require 7bit encoding.
```

Security considerations: Discussed in section 10 of RFC 3028.
Interoperability considerations: Discussed in section 2.10.5
of RFC 3028.
Published specification: RFC 3028.
Applications which use this media type: sieve-enabled mail servers
Additional information:
Magic number(s):
File extension(s): .siv
Macintosh File Type Code(s):
Person & email address to contact for further information:
See the discussion list at ietf-mta-filters@imc.org.
Intended usage:
COMMON
Author/Change controller:
See Author information in RFC 3028.

8. Parsing

The Sieve grammar is separated into tokens and a separate grammar as most programming languages are.

8.1. Lexical Tokens

Sieve scripts are encoded in UTF-8. The following assumes a valid UTF-8 encoding; special characters in Sieve scripts are all ASCII.

The following are tokens in Sieve:

- identifiers
- tags
- numbers
- quoted strings
- multi-line strings
- other separators

Blanks, horizontal tabs, CRLFs, and comments ("white space") are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent tokens and in specific places in the multi-line strings.

The other separators are single individual characters, and are mentioned explicitly in the grammar.

The lexical structure of sieve is defined in the following BNF (as described in [ABNF]):

```
   bracket-comment = "/*" *(CHAR-NOT-STAR / ("*" CHAR-NOT-SLASH)) "*/"
           ;; No */ allowed inside a comment.
           ;; (No * is allowed unless it is the last character,
           ;; or unless it is followed by a character that isn't a
           ;; slash.)

   CHAR-NOT-DOT = (%x01-09 / %x0b-0c / %x0e-2d / %x2f-ff)
           ;; no dots, no CRLFs

   CHAR-NOT-CRLF = (%x01-09 / %x0b-0c / %x0e-ff)

   CHAR-NOT-SLASH = (%x00-57 / %x58-ff)

   CHAR-NOT-STAR = (%x00-51 / %x53-ff)

   comment = bracket-comment / hash-comment

   hash-comment = ( "#" *CHAR-NOT-CRLF CRLF )

   identifier = (ALPHA / "_") *(ALPHA DIGIT "_")

   tag = ":" identifier

   number = 1*DIGIT [QUANTIFIER]

   QUANTIFIER = "K" / "M" / "G"

   quoted-string = DQUOTE *CHAR DQUOTE
           ;; in general, \ CHAR inside a string maps to CHAR
           ;; so \" maps to " and \\ maps to \
           ;; note that newlines and other characters are all allowed
           ;; strings

   multi-line          = "text:" *(SP / HTAB) (hash-comment / CRLF)
                         *(multi-line-literal / multi-line-dotstuff)
                         "." CRLF
   multi-line-literal  = [CHAR-NOT-DOT *CHAR-NOT-CRLF] CRLF
   multi-line-dotstuff = "." 1*CHAR-NOT-CRLF CRLF
           ;; A line containing only "." ends the multi-line.
           ;; Remove a leading '.' if followed by another '.'.

   white-space = 1*(SP / CRLF / HTAB) / comment
```

8.2. Grammar

The following is the grammar of Sieve after it has been lexically interpreted. No white space or comments appear below. The start symbol is "start".

```
argument = string-list / number / tag

arguments = *argument [test / test-list]

block = "{" commands "}"

command = identifier arguments ( ";" / block )

commands = *command

start = commands

string = quoted-string / multi-line

string-list = "[" string *("," string) "]" / string         ;; if
there is only a single string, the brackets are optional

test = identifier arguments

test-list = "(" test *("," test) ")"
```

9. Extended Example

The following is an extended example of a Sieve script. Note that it does not make use of the implicit keep.

```
 #
 # Example Sieve Filter
 # Declare any optional features or extension used by the script
 #
 require ["fileinto", "reject"];

 #
 # Reject any large messages (note that the four leading dots get
 # "stuffed" to three)
 #
 if size :over 1M
         {
         reject text:
 Please do not send me large attachments.
 Put your file on a server and send me the URL.
 Thank you.
 .... Fred
 .
 ;
         stop;
         }
 #
```

```
    # Handle messages from known mailing lists
    # Move messages from IETF filter discussion list to filter folder
    #
    if header :is "Sender" "owner-ietf-mta-filters@imc.org"
            {
            fileinto "filter";  # move to "filter" folder
            }
    #
    # Keep all messages to or from people in my company
    #
    elsif address :domain :is ["From", "To"] "example.com"
            {
            keep;               # keep in "In" folder
            }

    #
    # Try and catch unsolicited email.  If a message is not to me,
    # or it contains a subject known to be spam, file it away.
    #
    elsif anyof (not address :all :contains
                   ["To", "Cc", "Bcc"] "me@example.com",
                 header :matches "subject"
                   ["*make*money*fast*", "*university*dipl*mas*"])
            {
            # If message header does not contain my address,
            # it's from a list.
            fileinto "spam";   # move to "spam" folder
            }
    else
            {
            # Move all other (non-company) mail to "personal"
            # folder.
            fileinto "personal";
            }
```

10. Security Considerations

Users must get their mail. It is imperative that whatever method implementations use to store the user-defined filtering scripts be secure.

It is equally important that implementations sanity-check the user's scripts, and not allow users to create on-demand mailbombs. For instance, an implementation that allows a user to reject or redirect multiple times to a single message might also allow a user to create a mailbomb triggered by mail from a specific user. Site- or implementation-defined limits on actions are useful for this.

Several commands, such as "discard", "redirect", and "fileinto" allow for actions to be taken that are potentially very dangerous.

Implementations SHOULD take measures to prevent languages from looping.

11. Acknowledgments

I am very thankful to Chris Newman for his support and his ABNF syntax checker, to John Myers and Steve Hole for outlining the requirements for the original drafts, to Larry Greenfield for nagging me about the grammar and finally fixing it, to Greg Sereda for repeatedly fixing and providing examples, to Ned Freed for fixing everything else, to Rob Earhart for an early implementation and a great deal of help, and to Randall Gellens for endless amounts of proofreading. I am grateful to Carnegie Mellon University where most of the work on this document was done. I am also indebted to all of the readers of the ietf-mta-filters@imc.org mailing list.

12. Author's Address

Tim Showalter
Mirapoint, Inc.
909 Hermosa Court
Sunnyvale, CA 94085

EMail: tjs@mirapoint.com

13. References

[ABNF] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[ACAP] Newman, C. and J. G. Myers, "ACAP -- Application Configuration Access Protocol", RFC 2244, November 1997.

[BINARY-SI] "Standard IEC 60027-2: Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics", January 1999.

[DSN] Moore, K. and G. Vaudreuil, "An Extensible Message Format for Delivery Status Notifications", RFC 1894, January 1996.

[FLAMES] Borenstein, N, and C. Thyberg, "Power, Ease of Use, and Cooperative Work in a Practical Multimedia Message System", Int. J. of Man-Machine Studies, April, 1991. Reprinted in Computer-Supported Cooperative Work and Groupware, Saul Greenberg, editor, Harcourt Brace Jovanovich, 1991. Reprinted in Readings in Groupware and Computer-Supported Cooperative Work, Ronald Baecker, editor, Morgan Kaufmann, 1993.

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

[IMAP] Crispin, M., "Internet Message Access Protocol - version 4rev1", RFC 2060, December 1996.

[IMAIL] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, August 1982.

[MIME] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[MDN] Fajman, R., "An Extensible Message Format for Message Disposition Notifications", RFC 2298, March 1998.

[RFC1123] Braden, R., "Requirements for Internet Hosts -- Application and Support", STD 3, RFC 1123, November 1989.

[SMTP] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC 821, August 1982.

[UTF-8] Yergeau, F., "UTF-8, a transformation format of Unicode and ISO 10646", RFC 2044, October 1996.

14. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.