Network Working Group
Request for Comments: 5228
Obsoletes: 3028
Category: Standards Track

P. Guenther, Ed.
Sendmail, Inc.
T. Showalter, Ed.

January 2008

Sieve: An Email Filtering Language

Status of This Memo

Abstract

This document describes a language for filtering email 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 the base language 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 graphical user interfaces (GUIs) for filter creation and manipulation. The base language was not designed to be Turing-complete: it does not have a loop control structure or functions.

Scripts written in Sieve are executed during final delivery, when the message is moved to the user-accessible mailbox. In systems where the Mail Transfer Agent (MTA) does final delivery, such as traditional Unix mail, it is reasonable to filter 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 email, 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", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [KEYWORDS].

Each section on a command (test, action, or control) has a line labeled "Usage:". This line describes the usage 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 parentheses are used for grouping, similar to [ABNF].

In the "Usage:" 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 is defined in section 8 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, or 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.com
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.

With the exceptions of strings and comments, the language is limited to US-ASCII characters. Strings and comments may contain octets outside the US-ASCII range. Specifically, they will normally be in UTF-8, as specified in [UTF-8]. NUL (US-ASCII 0) is never permitted in scripts, while CR and LF can only appear as the CRLF line ending.

> Note: While this specification permits arbitrary octets to appear in Sieve scripts inside strings and comments, this has made it difficult to robustly handle Sieve scripts in programs that are sensitive to the encodings used. The "encoded-character" capability (section 2.4.2.4) provides an alternative means of representing such octets in strings using just US-ASCII characters. As such, the use of non-UTF-8 text in scripts should be considered a deprecated feature that may be abandoned.

Tokens other than strings 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, strings, and string lists.

2.4.1. Numbers

Numbers are given as ordinary decimal numbers. As a shorthand for expressing larger values, such as message sizes, a suffix of "K", "M", or "G" MAY be 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 gibi-, or 1,073,741,824 (2^30) times the value of the number [BINARY-SI].

Implementations MUST support integer values in the inclusive range zero to 2,147,483,647 (2^31 - 1), but MAY support larger values.

Only non-negative integers are permitted by this specification.

2.4.2. Strings

Scripts involve large numbers of string values 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, US-ASCII 34). A backslash ("\", US-ASCII 92) inside of a quoted string is followed by either another backslash or a double quote. These two-character sequences represent a single backslash or double quote within the value, respectively.

Scripts SHOULD NOT escape other characters with a 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"), though that may be changed by extensions.

Non-printing characters such as tabs, CRLF, and control characters are permitted in quoted strings. Quoted strings MAY span multiple lines. An unencoded NUL (US-ASCII 0) is not allowed in strings; see section 2.4.2.4 for how it can be encoded.

As message header data is converted to [UTF-8] for comparison (see section 2.7.2), most string values will use the UTF-8 encoding. However, implementations MUST accept all strings that match the grammar in section 8. The ability to use non-UTF-8 encoded strings matches existing practice and has proven to be useful both in tests for invalid data and in arguments containing raw MIME parts for extension actions that generate outgoing messages.

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. The CRLF before the final period is considered part of the value. 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 that 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 as 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.

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

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], 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.

Similarly, no header will match a syntactically invalid header name. An implementation MUST NOT cause an error for syntactically invalid header names in tests.

Header lines are unfolded as described in [IMAIL] section 2.2.3. Interpretation of header data SHOULD be done according to [MIME3] section 6.2 (see section 2.7.2 below for details).

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 within this document. Using the symbols defined in [IMAIL], section 3, 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 permitted.

It is an error for a script to execute an action with a value for use as an outbound address that doesn't match the "sieve-address" syntax.

2.4.2.4. Encoding Characters Using "encoded-character"

When the "encoded-character" extension is in effect, certain character sequences in strings are replaced by their decoded value. This happens after escape sequences are interpreted and dot-unstuffing has been done. Implementations SHOULD support "encoded-character".

Arbitrary octets can be embedded in strings by using the syntax encoded-arb-octets. The sequence is replaced by the octets with the hexadecimal values given by each hex-pair.

```
blank                = WSP / CRLF
encoded-arb-octets   = "${hex:" hex-pair-seq "}"
hex-pair-seq         = *blank hex-pair *(1*blank hex-pair) *blank
hex-pair             = 1*2HEXDIG
```

Where WSP and HEXDIG non-terminals are defined in Appendix B.1 of [ABNF].

It may be inconvenient or undesirable to enter Unicode characters verbatim, and for these cases the syntax encoded-unicode-char can be used. The sequence is replaced by the UTF-8 encoding of the specified Unicode characters, which are identified by the hexadecimal value of unicode-hex.

```
encoded-unicode-char = "${unicode:" unicode-hex-seq "}"
unicode-hex-seq      = *blank unicode-hex
                       *(1*blank unicode-hex) *blank
unicode-hex          = 1*HEXDIG
```

It is an error for a script to use a hexadecimal value that isn't in either the range 0 to D7FF or the range E000 to 10FFFF. (The range D800 to DFFF is excluded as those character numbers are only used as part of the UTF-16 encoding form and are not applicable to the UTF-8 encoding that the syntax here represents.)

> Note: Implementations MUST NOT raise an error for an out-of-range Unicode value unless the sequence containing it is well-formed according to the grammar.

The capability string for use with the require command is "encoded-character".

In the following script, message B is discarded, since the specified test string is equivalent to "$$$".

```
Example:  require "encoded-character";
          if header :contains "Subject" "$${hex:24 24}" {
             discard;
          }
```

The following examples demonstrate valid and invalid encodings and how they are handled:

```
  "$${hex:40}"         -> "$@"
  "${hex: 40 }"        -> "@"
  "${HEX: 40}"         -> "@"
  "${hex:40"           -> "${hex:40"
  "${hex:400}"         -> "${hex:400}"
  "${hex:4${hex:30}}"  -> "${hex:40}"
  "${unicode:40}"      -> "@"
  "${ unicode:40}"     -> "${ unicode:40}"
  "${UNICODE:40}"      -> "@"
  "${UnICoDE:0000040}" -> "@"
  "${Unicode:40}"      -> "@"
  "${Unicode:Cool}"    -> "${Unicode:Cool}"
  "${unicode:200000}"  -> error
  "${Unicode:DF01}     -> error
```

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 to decide which block of code is run.

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 as a comma-separated list in parentheses.

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

It is an error for a script, on a single command, to use conflicting arguments or to use a tagged or optional argument more than once.

2.6.1. Positional Arguments

Positional arguments are given to a command that 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 literal data, 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.

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 comparator [COLLATION] will be used to compare two strings, since different languages may impose different orderings on UTF-8 [UTF-8] strings.

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 8.2 presents a parsable version of this: Arguments are string lists (string-lists), numbers, and tags, which may be followed by a test or a test list (test-list), which may be followed by a block of commands. No more than one test or test list, or more than one block of commands, may be used, and commands that end with a block 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 uses the comparators defined in the Internet Application Protocol Collation Registry [COLLATION].

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

Commands that perform string comparisons may have an optional match type argument. The three match types in this specification are ":contains", ":is", and ":matches".

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 empty 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 empty key ("") only ":is" matches with the empty value.

The ":matches" match type specifies a wildcard match using the characters "*" and "?"; the entire value must be matched. "*" matches zero or more characters in the value and "?" matches a single character in the value, where the comparator that is used (see section 2.7.3) defines what a character is. For example, the comparators "i;octet" and "i;ascii-casemap" define a character to be a single octet, so "?" will always match exactly one octet when one of those comparators is in use. In contrast, a Unicode-based comparator would define a character to be any UTF-8 octet sequence encoding one Unicode character and thus "?" may match more than one octet. "?" 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 arguments ":matches", ":is", and ":contains". Commands default to using ":is" matching if no match type argument is supplied. Note that these modifiers interact with comparators; in particular, only comparators that support the "substring match" operation are 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

Messages may involve a number of character sets. In order for comparisons to work across character sets, implementations SHOULD implement the following behavior:

> Comparisons are performed on octets. Implementations convert text from header fields in all charsets [MIME3] to Unicode, encoded as UTF-8, as input to the comparator (see section 2.7.3). Implementations MUST be capable of converting US-ASCII, ISO-8859-1, the US-ASCII subset of ISO-8859-* character sets, and UTF-8. Text that the implementation cannot convert to Unicode for any reason MAY be treated as plain US-ASCII (including any [MIME3] syntax) or processed according to local conventions. An encoded NUL octet (character zero) SHOULD NOT cause early termination of the header content being compared against.

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. The Internet Application Protocol Collation Registry [COLLATION] provides the framework for describing and naming comparators.

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 US-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 to use a comparator with ":matches" or ":contains" that is not compatible with it.

Sieve treats a comparator result of "undefined" the same as a result of "no-match". That is, this base specification does not provide any means to directly detect invalid comparator input.

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 section 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.

If an address is not syntactically valid, then it will not be matched by tests specifying ":localpart" or ":domain".

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 and supplied as the final argument to a command. Such a command is a control structure: when executed it has control over the number of times the commands in the block are executed.

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.

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. The actions referenced in this document are:

- keep, to save the message in the default location
- fileinto, to save the message in a specific mailbox
- redirect, to forward the message to another address
- discard, to silently throw away the message

A control command is a command that affects the parsing or the flow of execution of the Sieve script in some way. A control structure is a control command that 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 section 4.3) 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 may.

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 mailbox 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 the number 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 allow at least one keep or one fileinto. If fileinto is not implemented, implementations MUST allow at least one keep.

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. This protects scripts from having their behavior altered by extensions that the script author might not have even been aware of.

Implementations MUST NOT execute any Sieve script test or command subsequent to "require" if one of the required extensions is unavailable.

> 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.

Extensions that 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 MUST perform syntactic, semantic, and run-time checks on code that is actually executed. Implementations MAY perform those checks or any part of them on code that is not reached during execution.

When an error happens, implementations MUST notify the user that an error occurred and 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 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 or else MUST only follow an if or elsif. An error occurs if these conditions are not met.

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

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

Usage: else <block3: block>

The semantics are similar to those of any of the many other programming languages these control structures 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's block.

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

In the following example, both messages 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.com; message B, to postmaster@example.com; any other message is redirected to field@example.com.

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

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 require

Usage: 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 stop

Usage: stop

The "stop" action ends all processing. If the implicit keep has not been cancelled, then it is taken.

4. Action Commands

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

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

Implementations SHOULD support "fileinto".

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

4.1. Action fileinto

Usage: fileinto <mailbox: string>

The "fileinto" action delivers the message into the specified mailbox. Implementations SHOULD support fileinto, but in some environments this may be impossible. Implementations MAY place restrictions on mailbox names; use of an invalid mailbox name MAY be treated as an error or result in delivery to an implementation-defined mailbox. If the specified mailbox doesn't exist, the implementation MAY treat it as an error, create the mailbox, or deliver the message to an implementation-defined mailbox. If the implementation uses a different encoding scheme than UTF-8 for mailbox names, it SHOULD reencode the mailbox name from UTF-8 to its encoding scheme. For example, the Internet Message Access Protocol [IMAP] uses modified UTF-7, such that a mailbox argument of "odds & ends" would appear in IMAP as "odds &- ends".

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

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

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

4.2. Action redirect

Usage: 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. In particular, existing Received headers MUST be preserved and the count of Received headers in the outgoing message MUST be larger than the same count on the message as received by the implementation. (An implementation that adds a Received header before processing the message does not need to add another when redirecting.)

The message is sent back out with the address from the redirect command as an envelope recipient. Implementations MAY combine separate redirects for a given message into a single submission with multiple envelope recipients. (This is not a Mail User Agent (MUA)-style forward, which creates a new message with a different sender and message ID, wrapping the old message in a new one.)

The envelope sender address on the outgoing message is chosen by the sieve implementation. It MAY be copied from the message being processed. However, if the message being processed has an empty envelope sender address the outgoing message MUST also have an empty envelope sender address. This last requirement is imposed to prevent loops in the case where a message is redirected to an invalid address when then returns a delivery status notification that also ends up being redirected to the same invalid address.

A simple script can be used for redirecting all mail:

Example: redirect "bart@example.com";

Implementations MUST take measures to implement loop control, possibly including adding headers to the message or counting Received headers as specified in section 6.2 of [SMTP]. If an implementation detects a loop, it causes an error.

Implementations MUST provide means of limiting the number of redirects a Sieve script can perform. See section 10 for more details.

Implementations MAY ignore a redirect action silently due to policy reasons. For example, an implementation MAY choose not to redirect to an address that is known to be undeliverable. Any ignored redirect MUST NOT cancel the implicit keep.

4.3. Action keep

Usage: 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; }

4.4. Action discard

Usage: 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.com" is thrown out.

```
Example:  if header :contains ["from"] ["idiot@example.com"] {
             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

Usage: address [COMPARATOR] [ADDRESS-PART] [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. Whether there are other addresses present in the header doesn't affect this test; this test does not provide any way to determine whether an address is the only address in a header.

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

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 or 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, and Resent-To, and it 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

Usage: 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

Usage: 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

Usage: 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. This specification defines the interpretation of the (case insensitive) "from" and "to" envelope-parts. Additional envelope-parts may be defined by other extensions; implementations SHOULD consider unknown envelope parts an error.

If one of the envelope-part strings is (case insensitive) "from", then matching occurs against the FROM address used in the SMTP MAIL command. The null reverse-path is matched against as the empty string, regardless of the ADDRESS-PART argument specified.

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 list and key-list arguments match and returns false otherwise.

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

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

5.5. Test exists

Usage: 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

Usage: false

The "false" test always evaluates to false.

5.7. Test header

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

The "header" test evaluates to true if the value of any of the named headers, ignoring leading and trailing whitespace, 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 header-names list and key-list arguments match and returns false otherwise.

If a header listed in the header-names argument exists, it contains the empty key (""). However, if the named header is not present, it does not match any key, including the empty 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
```

Testing whether a given header is either absent or doesn't contain any non-whitespace characters can be done using a negated "header" test:

```
not header :matches "Cc" "?*"
```

5.8. Test not

Usage: not <test1: 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

Usage: 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 in the [IMAIL] representation of the message.

Note that for a message that is exactly 4,000 octets, the message is neither ":over" nor ":under" 4000 octets.

5.10. Test true

Usage: true

The "true" test always evaluates to true.

6. Extensibility

New control commands, 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. Capability string are case-sensitive; for example, "foo" and "FOO" are different capabilities. If a new version of an extension changes the functionality of a previously defined extension, it MUST use a different name. Extensions may register a set of related capabilities by registering just a unique prefix for them. The "comparator-" prefix is an example of this. The prefix MUST end with a "-" and MUST NOT overlap any existing registrations.

In a situation where there is a script 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. Extensions MUST NOT change the behavior of the "require" control command or alter the interpretation of the argument to the "require" control.

Extensions that can submit new email messages or otherwise generate new protocol requests MUST consider loop suppression, at least to document any security considerations.

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:

encoded-character The string "encoded-character" indicates that the implementation supports the interpretation of "${hex:...}" and "${unicode:...}" in strings.

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

fileinto The string "fileinto" indicates that the implementation supports the "fileinto" 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 maintained by IANA. This registry contains both vendor-controlled capability names (beginning with "vnd.") and IETF-controlled capability names. Vendor-controlled capability names may be registered on a first-come, first-served basis, by applying to IANA with the form in the following section. Registration of capability prefixes that do not begin with "vnd." REQUIRES a standards track or IESG-approved experimental RFC.

Extensions designed for interoperable use SHOULD use IETF-controlled capability names.

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: [the string for use in the 'require' statement]
Description: [a brief description of what the extension adds or changes]
RFC number: [for extensions published as RFCs]
Contact address: [email and/or physical address to contact for additional information]

6.2.2. Handling of Existing Capability Registrations

In order to bring the existing capability registrations in line with the new template, IANA has modified each as follows:

1. The "capability name" and "capability arguments" fields have been eliminated
2. The "capability keyword" field have been renamed to "Capability name"
3. An empty "Description" field has been added
4. The "Standards Track/IESG-approved experimental RFC number" field has been renamed to "RFC number"
5. The "Person and email address to contact for further information" field should be renamed to "Contact address"

6.2.3. Initial Capability Registrations

This RFC updates the following entries in the IANA registry for Sieve extensions.

Capability name: encoded-character
Description: changes the interpretation of strings to allow arbitrary octets and Unicode characters to be represented using US-ASCII
RFC number: RFC 5228 (Sieve base spec)
Contact address: The Sieve discussion list <ietf-mta-filters@imc.org>

Capability name: fileinto
Description: adds the 'fileinto' action for delivering to a mailbox other than the default
RFC number: RFC 5228 (Sieve base spec)
Contact address: The Sieve discussion list <ietf-mta-filters@imc.org>

```
Capability name: envelope
Description:     adds the 'envelope' test for testing the message
                 transport sender and recipient address
RFC number:      RFC 5228 (Sieve base spec)
Contact address: The Sieve discussion list <ietf-mta-filters@imc.org>

Capability name: comparator-* (anything starting with "comparator-")
Description:     adds the indicated comparator for use with the
                 :comparator argument
RFC number:      RFC 5228 (Sieve base spec) and [COLLATION]
Contact address: The Sieve discussion list <ietf-mta-filters@imc.org>
```

6.3. Capability Transport

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 the 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 updated 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 this RFC.
Interoperability considerations: Discussed in section 2.10.5
   of this RFC.
Published specification: this RFC.
Applications that use this media type: sieve-enabled mail
  servers and clients
Additional information:
  Magic number(s):
  File extension(s): .siv .sieve
  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:
The SIEVE WG, delegated by the IESG.

8. Parsing

The Sieve grammar is separated into tokens and a separate grammar as most programming languages are. Additional rules are supplied here for common arguments to various language facilities.

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 US-ASCII.

The following are tokens in Sieve:

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

Identifiers, tags, and numbers are case-insensitive, while quoted strings and multi-line strings are case-sensitive.

Blanks, horizontal tabs, CRLFs, and comments ("whitespace") are ignored except as they separate tokens. Some whitespace is required to separate otherwise adjacent tokens and in specific places in the multi-line strings. CR and LF can only appear in CRLF pairs.

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

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

```
bracket-comment    = "/*" *not-star 1*STAR
                     *(not-star-slash *not-star 1*STAR) "/"
                       ; 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.)
```

```
   comment            = bracket-comment / hash-comment

   hash-comment       = "#" *octet-not-crlf CRLF

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

   multi-line         = "text:" *(SP / HTAB) (hash-comment / CRLF)
                        *(multiline-literal / multiline-dotstart)
                        "." CRLF

   multiline-literal  = [ octet-not-period *octet-not-crlf ] CRLF

   multiline-dotstart = "." 1*octet-not-crlf CRLF
                          ; A line containing only "." ends the
                          ; multi-line.  Remove a leading '.' if
                          ; followed by another '.'.

   not-star           = CRLF / %x01-09 / %x0B-0C / %x0E-29 / %x2B-FF
                            ; either a CRLF pair, OR a single octet
                            ; other than NUL, CR, LF, or star

   not-star-slash     = CRLF / %x01-09 / %x0B-0C / %x0E-29 / %x2B-2E /
                        %x30-FF
                            ; either a CRLF pair, OR a single octet
                            ; other than NUL, CR, LF, star, or slash

   number             = 1*DIGIT [ QUANTIFIER ]

   octet-not-crlf     = %x01-09 / %x0B-0C / %x0E-FF
                            ; a single octet other than NUL, CR, or LF

   octet-not-period   = %x01-09 / %x0B-0C / %x0E-2D / %x2F-FF
                            ; a single octet other than NUL,
                            ; CR, LF, or period

   octet-not-qspecial = %x01-09 / %x0B-0C / %x0E-21 / %x23-5B / %x5D-FF
                            ; a single octet other than NUL,
                            ; CR, LF, double-quote, or backslash

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

   quoted-other       = "\" octet-not-qspecial
                            ; represents just the octet-no-qspecial
                            ; character.  SHOULD NOT be used

   quoted-safe        = CRLF / octet-not-qspecial
                            ; either a CRLF pair, OR a single octet other
                            ; than NUL, CR, LF, double-quote, or backslash
```

```
quoted-special     = "\" (DQUOTE / "\")
                       ; represents just a double-quote or backslash

quoted-string      = DQUOTE quoted-text DQUOTE

quoted-text        = *(quoted-safe / quoted-special / quoted-other)

STAR               = "*"

tag                = ":" identifier

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

8.2. Grammar

The following is the grammar of Sieve after it has been lexically interpreted. No whitespace 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) ")"
```

8.3. Statement Elements

These elements are collected from the "Syntax" sections elsewhere in this document, and are provided here in [ABNF] syntax so that they can be modified by extensions.

```
ADDRESS-PART = ":localpart" / ":domain" / ":all"
```

```
COMPARATOR   = ":comparator" string

MATCH-TYPE   = ":is" / ":contains" / ":matches"
```

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"];

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

#
# 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*"])
        {
        fileinto "spam";   # move to "spam" mailbox
        }
else
        {
        # Move all other (non-company) mail to "personal"
        # mailbox.
        fileinto "personal";
        }
```

10. Security Considerations

Users must get their mail. It is imperative that whatever implementations use to store the user-defined filtering scripts protect them from unauthorized modification, to preserve the integrity of the mail system. An attacker who can modify a script can cause mail to be discarded, rejected, or forwarded to an unauthorized recipient. In addition, it's possible that Sieve scripts might expose private information, such as mailbox names, or email addresses of favored (or disfavored) correspondents. Because of that, scripts SHOULD also be protected from unauthorized retrieval.

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

Use of the "redirect" command to generate notifications may easily overwhelm the target address, especially if it was not designed to handle large messages.

Allowing a single script to redirect to multiple destinations can be used as a means of amplifying the number of messages in an attack. Moreover, if loop detection is not properly implemented, it may be possible to set up exponentially growing message loops. Accordingly, Sieve implementations:

(1) MUST implement facilities to detect and break message loops. See section 6.2 of [SMTP] for additional information on basic loop detection strategies.

(2) MUST provide the means for administrators to limit the ability of users to abuse redirect. In particular, it MUST be possible to limit the number of redirects a script can perform. Additionally, if no use cases exist for using redirect to multiple destinations, this limit SHOULD be set to 1. Additional limits, such as the ability to restrict redirect to local users, MAY also be implemented.

(3) MUST provide facilities to log use of redirect in order to facilitate tracking down abuse.

(4) MAY use script analysis to determine whether or not a given script can be executed safely. While the Sieve language is sufficiently complex that full analysis of all possible scripts is computationally infeasible, the majority of real-world scripts are amenable to analysis. For example, an implementation might

allow scripts that it has determined are safe to run unhindered, block scripts that are potentially problematic, and subject unclassifiable scripts to additional auditing and logging.

Allowing redirects at all may not be appropriate in situations where email accounts are freely available and/or not trackable to a human who can be held accountable for creating message bombs or other abuse.

As with any filter on a message stream, if the Sieve implementation and the mail agents 'behind' Sieve in the message stream differ in their interpretation of the messages, it may be possible for an attacker to subvert the filter. Of particular note are differences in the interpretation of malformed messages (e.g., missing or extra syntax characters) or those that exhibit corner cases (e.g., NUL octets encoded via [MIME3]).

11. Acknowledgments

This document has been revised in part based on comments and discussions that took place on and off the SIEVE mailing list. Thanks to Sharon Chisholm, Cyrus Daboo, Ned Freed, Arnt Gulbrandsen, Michael Haardt, Kjetil Torgrim Homme, Barry Leiba, Mark E. Mallett, Alexey Melnikov, Eric Rescorla, Rob Siemborski, and Nigel Swinson for reviews and suggestions.

12. Normative References

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

[COLLATION] Newman, C., Duerst, M., and A. Gulbrandsen, "Internet Application Protocol Collation Registry", RFC 4790, March 2007.

[IMAIL] Resnick, P., Ed., "Internet Message Format", RFC 2822, April 2001.

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

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

[MIME3] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, November 1996.

[SMTP] Klensin, J., Ed., "Simple Mail Transfer Protocol", RFC 2821, April 2001.

[UTF-8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

13. Informative References

[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 3464, January 2003.

[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.

[IMAP] Crispin, M., "Internet Message Access Protocol - version 4rev1", RFC 3501, March 2003.

[MDN] Hansen, T., Ed., and G. Vaudreuil, Ed., "Message Disposition Notification", RFC 3798, May 2004.

[RFC3028] Showalter, T., "Sieve: A Mail Filtering Language", RFC 3028, January 2001.

14. Changes from RFC 3028

This following list is a summary of the changes that have been made in the Sieve language base specification from [RFC3028].

1. Removed ban on tests having side-effects
2. Removed reject extension (will be specified in a separate RFC)
3. Clarified description of comparators to match [COLLATION], the new base specification for them
4. Require stripping of leading and trailing whitespace in "header" test
5. Clarified or tightened handling of many minor items, including:
   - invalid [MIME3] encoding
   - invalid addresses in headers
   - invalid header field names in tests
   - 'undefined' comparator result
   - unknown envelope parts
   - null return-path in "envelope" test
6. Capability strings are case-sensitive
7. Clarified that fileinto should reencode non-ASCII mailbox names to match the mailstore's conventions
8. Errors in the ABNF were corrected
9. The references were updated and split into normative and informative
10. Added encoded-character capability and deprecated (but did not remove) use of arbitrary binary octets in Sieve scripts.
11. Updated IANA registration template, and added IANA considerations to permit capability prefix registrations.
12. Added .sieve as a valid extension for Sieve scripts.

Editors' Addresses

Philip Guenther
Sendmail, Inc.
6425 Christie St. Ste 400
Emeryville, CA 94608
EMail: guenther@sendmail.com

Tim Showalter
EMail: tjs@psaux.com

Full Copyright Statement

Intellectual Property