The Open Source Solution to SPAM

Covers SpamAssassin Version 3.0

SpamAssassin

O'REILLY®

Alan Schwartz

SpamAssassin

Alan Schwartz

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

SpamAssassin
by Alan Schwartz

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Jonathan Gennick
Production Editor:	Darren Kelly
Cover Designer:	Ellie Volckhausen
Interior Designer:	Melanie Wang
Production Services:	Nancy Crumpton

Printing History:

July 2004:	First Edition.

RepKover™ This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-00707-8
[M]

Table of Contents

Preface

If you use email, it's likely that you've recently been visited by a piece of spam—an unsolicited, unwanted message, sent to you without your permission.* If you manage an email system, it's almost certain that you've had to help your users avoid the deluge of unwanted email.

System administrators pay for spam with their time. The Internet's email system was designed to make it difficult to lose email messages: when a computer can't deliver a message to the intended recipient, it does its best to return that message to the sender. If it can't send the message to the sender, it sends it to the computer's postmaster—because something must be seriously wrong if both the email addresses of the sender and the recipient of a message are invalid.

The well-meaning nature of Internet mail software becomes a positive liability when spammers come into the picture. In a typical bulk mailing, anywhere from a few hundred to tens of thousands of email addresses might be invalid. Under normal circumstances these email messages would bounce back to the sender. But the spammer doesn't want them! To avoid being overwhelmed, spammers often use invalid return addresses. The result: the email messages end up in the mailboxes of the Internet postmasters, who are usually living, breathing system administrators.

System administrators at large sites are now receiving hundreds to thousands of bounced spam messages each day. Unfortunately, each of these messages has to be carefully examined, because mixed in with these messages are the occasional bounced mail messages from misconfigured computers that actually should be fixed.

As the spam problem grows worse and worse, system administrators are increasingly taking themselves off their computers' "postmaster" mailing lists. The result is predictable: they're deluged with less email, but problems that they would normally dis-

* Spam is also a registered trademark of Hormel Foods, which uses the word to describe a canned luncheon meat. In this book, the word "spam" is used exclusively to refer to Internet spam and not the meat.

cover by receiving postmaster email are being missed as well. The Internet as a whole suffers as a result.

Although there are many important ways to reduce spam—including obscuring email addresses, complaining to spammers' service providers, and seeking legal and legislative relief—few remedies are as immediately effective as filtering email messages on the basis of content and format, and few filtering systems are as widely used and well maintained as SpamAssassin™.

This book is for mail system administrators, network administrators, and Internet service providers who are concerned about the growing toll that spam is taking on their systems and their users and are looking for a way to regain some control or reduce the burden on their users.

Scope of This Book

This book is divided into nine chapters and one appendix. The first four chapters deal with core SpamAssassin concepts that are independent of the underlying mail system.

Chapter 1, *Introducing SpamAssassin*
: Explains what SpamAssassin does, and provides a conceptual overview of its organization and features.

Chapter 2, *SpamAssassin Basics*
: Covers the installation, testing, and basic operation of SpamAssassin.

Chapter 3, *SpamAssassin Rules*
: Details the configuration of SpamAssassin, and focuses particularly on SpamAssassin's spam-detection rules. It explains how to increase or decrease the impact of rules, write new rules, and add addresses to blacklists and whitelists.

Chapter 4, *SpamAssassin as a Learning System*
: Reviews the learning features of SpamAssassin: automatic whitelisting and Bayesian filtering. It provides the theory behind these features and discusses how to configure, train, and tune them.

The remaining five chapters detail the integration of SpamAssassin with several popular mail transport agents (MTAs) to provide sitewide spam-checking. They also explain how to set up a SpamAssassin gateway to check all incoming mail before delivery to an internal mail host.

Chapter 5, *Integrating SpamAssassin with sendmail*
: Explains how to integrate SpamAssassin with the sendmail MTA, using the milter interface. As an example of this approach, the installation and configuration of MIMEDefang is described.

Chapter 6, *Integrating SpamAssassin with Postfix*
: Explains how to integrate SpamAssassin with the Postfix MTA, using the *content_filter* interface. As an example of this approach, the installation and configuration of amavisd-new, a daemonized content filter, is described.

Chapter 7, *Integrating SpamAssassin with qmail*
: Explains how to integrate SpamAssassin with the qmail MTA.

Chapter 8, *Integrating SpamAssassin with Exim*
: Explains how to integrate SpamAssassin with the Exim MTA using several different popular approaches including custom transports, exiscan, and sa-exim.

Chapter 9, *Using SpamAssassin as a Proxy*
: Explains how to set up a SpamAssassin POP mail proxy to support users who download their email with POP clients.

The Appendix lists useful resources for more information about SpamAssassin and other antispam approaches.

Versions Covered in This Book

At the time this book went to press, SpamAssassin 2.63 was the latest released version of SpamAssassin and was in wide use. The next-generation release of SpamAssassin, SpamAssassin 3.0, was available for beta-testing and is expected to be released at about the time this book appears in stores. SpamAssassin 3.0 introduces several important new features and changes parts of the Perl API.

Accordingly, this book covers both versions of SpamAssassin. When a topic or setting is specific to one version, I so note it.

Conventions Used in This Book

The following conventions are used in this book:

Italic
: Used for Unix file, directory, user, and group names and for Perl modules, objects, method names, and method options. It is also used for URLs (uniform resource locators) and to emphasize new terms and concepts when they are introduced.

`Constant Width`
: Used for Unix commands, code examples, and system output. It is also used for scripts, process names, and SpamAssassin directives.

`Constant Width Italic`
: Used in examples for variable input (e.g., a filename you must provide).

$
: The Unix Bourne shell or Korn shell prompt.

#
: The Unix superuser prompt. I use this symbol for examples that should be executed by root.

This icon designates a note, which is an important aside to the nearby text.

This icon designates a warning related to the nearby text.

Using Code Examples

All the code in this book is available for download from *http://www.oreilly.com/catalog/spamassassin*. See the file *readme.txt* in the download for installation instructions.

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, and publisher; for example: "*SpamAssassin*, by Alan Schwartz (O'Reilly)."

If you feel your use of code examples falls outside fair use or the permission given previously, feel free to contact us at *permissions@oreilly.com*.

Comments and Questions

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (U.S. and Canada)
(707) 827-7000 (international/local)
(707) 829-0104 (fax)

You can also contact O'Reilly by email. To be put on the mailing list or request a catalog, send a message to:

info@oreilly.com

We have a web page for this book, which lists errata, examples, and additional information. You can access this page at:

http://www.oreilly.com/catalog/spamassassin

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about O'Reilly books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com/

Acknowledgments

Bob Amen, Justin Mason, and Matt Riffle served as technical reviewers for this book. Any remaining errors, of course, are mine.

I have once again had the pleasure of collaborating with an excellent O'Reilly editor, Jonathan Gennick. The O'Reilly production crew for this book included Darren Kelly, Ellie Volckhausen, and Nancy Crumpton.

This book is dedicated to the developers and user community of SpamAssassin, for their fine work in helping to stem the flood of unwanted email.

Never-ending thanks to M.G. and Ari, who make it all worthwhile.

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 827-7000 (international/local)
(707) 829-0104 (fax)

You can also contact O'Reilly by email. To be put on the mailing list or request a catalog, send email to:

info@oreilly.com

We have a web page for this book, which lists errata, examples, and additional information. You can access this page at:

http://www.oreilly.com/catalog/pythonxml

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about O'Reilly books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com

Acknowledgments

Bob Aniello, Justin Mason, and Brad Robinson served as technical reviewers for this book. Any remaining errors or typos are mine.

I have once again had the pleasure of collaborating with an excellent O'Reilly editor, Jonathan Gennick. The O'Reilly production team for this book included Darren Kelly, Ellie Volckhausen, and Nancy Crumpton.

This book is dedicated to the developers and users of numerous open source tools, for their fine work in helping to make the field of software worthwhile.

Neverending thanks to M.G. and An, who make it all worthwhile.

CHAPTER 1

Introducing SpamAssassin

The SpamAssassin system is software for analyzing email messages, determining how likely they are to be spam, and reporting its conclusions. It is a rule-based system that compares different parts of email messages with a large set of rules. Each rule adds or removes points from a message's spam score. A message with a high enough score is reported to be spam.

SpamAssassin was a trademark of Deersoft, and Deersoft has been acquired by Network Associates. In this book, I won't write SpamAssassin™ each time I mention it because that would be distracting, but you should assume that the trademark symbol is there.

Many spam-checking systems are available. SpamAssassin has become popular for several reasons:

- It uses a large number of different kinds of rules and weights them according to their diagnosticity. Rules that have been demonstrated to be more effective at discriminating spam from non-spam email are given higher weightings.
- It is easy to tune the scores associated with each rule or to add new rules based on regular expressions.
- SpamAssassin can adapt to each system's email environment, learning to recognize which senders are to be trusted and to identify new kinds of spam.
- It can report spam to several different spam clearinghouses and can be configured to create *spam traps*—email addresses that are used only to forward spam to a clearinghouse.
- It is free software, distributed under either the GNU Public License or the Artistic License. Either license allows users to freely modify the software and redistribute their modifications under the same terms.

Example 1-1 shows a message that has been tagged as spam by SpamAssassin. Elements added by SpamAssassin appear in bold.

Example 1-1. A message tagged by SpamAssassin

```
From riverol5380503@jubii.dk Fri Nov  7 18:26:05 2003
Received: from localhost [127.0.0.1] by localhost
        with SpamAssassin (2.60 1.212-2003-09-23-exp);
        Sun, 09 Nov 2003 12:24:22 -0600
From: "brianj" <riverol5380503@jubii.dk>
To: <Undisclosed.Recipients@mailin-2.priv.cc.uic.edu>
Subject: Live your dream life!!                MPNWSTU
Date: Fri, 07 Nov 2003 15:32:41 -0800
Message-Id: <00016646728$00007347$00000042@mail3.mailnara.net>
X-Spam-Status: Yes, hits=12.9 required=5.0 tests=CLICK_BELOW,
        FORGED_MUA_EUDORA,FROM_ENDS_IN_NUMS,MISSING_OUTLOOK_NAME,
        MSGID_OUTLOOK_INVALID,MSGID_SPAM_ZEROES,NORMAL_HTTP_TO_IP,
        SUBJ_HAS_SPACES,SUBJ_HAS_UNIQ_ID autolearn=no version=2.60
X-Spam-Flag: YES
X-Spam-Checker-Version: SpamAssassin 2.60 (1.212-2003-09-23-exp)
X-Spam-Level: ************
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="----------=_3FAE8656.371BED4D"

This is a multi-part message in MIME format.

------------=_3FAE8656.371BED4D
Content-Type: text/plain
Content-Disposition: inline
Content-Transfer-Encoding: 8bit

Spam detection software, running on the system has
identified this incoming email as possible spam.  The original message
has been attached to this so you can view it (if it isn't spam) or block
similar future email.  If you have any questions, see
the administrator of that system for details.

Content preview:  Do you owe large sums of money? Are you stuck with high
  interest ra{tes? We can help! You can do what tens of thousands of
  americans have done, consolidate your high interest bills into one
  easy, low interest, monthly payment. [...]

Content analysis details:   (12.9 points, 5.0 required)

 pts rule name              description
---- ---------------------- --------------------------------------------------
 1.0 SUBJ_HAS_SPACES        Subject contains lots of white space
 4.3 MSGID_SPAM_ZEROES      Spam tool Message-Id: (12-zeroes variant)
 0.9 FROM_ENDS_IN_NUMS      From: ends in numbers
 0.2 NORMAL_HTTP_TO_IP      URI: Uses a dotted-decimal IP address in URL
 0.2 SUBJ_HAS_UNIQ_ID       Subject contains a unique ID
 4.3 MSGID_OUTLOOK_INVALID  Message-Id is fake (in Outlook Express format)
 0.1 MISSING_OUTLOOK_NAME   Message looks like Outlook, but isn't
 1.9 FORGED_MUA_EUDORA      Forged mail pretending to be from Eudora
 0.0 CLICK_BELOW            Asks you to click below

The original message was not completely plain text, and may be unsafe to
open with some email clients; in particular, it may contain a virus,
```

Example 1-1. A message tagged by SpamAssassin (continued)

```
or confirm that your address can receive spam.  If you wish to view
it, it may be safer to save it to a file and open it with an editor.


------------=_3FAE8656.371BED4D
Content-Type: message/rfc822; x-spam-type=original
Content-Description: original message before SpamAssassin
Content-Disposition: attachment
Content-Transfer-Encoding: 8bit

Received: (qmail 25515 invoked from network); 7 Nov 2003 18:26:02 -0600
Received: from mailin-2.cc.uic.edu (HELO mailin-2.priv.cc.uic.edu) (128.248.155.213)
  by email0.cc.uic.edu with SMTP; 7 Nov 2003 18:26:02 -0600
Received: from mail3.mailnara.net (c-24-98-136-187.atl.client2.attbi.com [24.98.136.187])
        by mailin-2.priv.cc.uic.edu (8.12.10/8.12.9) with ESMTP id hA80PxJk011669;
        Fri, 7 Nov 2003 18:26:00 -0600
Message-ID: <000016646728$00007347$00000042@mail3.mailnara.net>
To: <Undisclosed.Recipients@mailin-2.priv.cc.uic.edu>
From: "brianj" <riverol5380503@jubii.dk>
Subject: Live your dream life!!                 MPNWSTU
Date: Fri, 07 Nov 2003 15:32:41 -0800
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="----------=_1068251164-2528-687"
X-Priority: 3
X-MSMail-Priority: Normal
X-Mailer: QUALCOMM Windows Eudora Version 5.1
X-MimeOLE: Produced By Microsoft MimeOLE V5.00.3018.1300
Content-Length: 2290
Lines: 72


Do you owe large sums of money? Are you stuck with
high interest ra{tes? We can help!

You can do what tens of thousands of americans have
done, consolidate your high interest bills into one
easy, low interest, monthly payment.

By first reducing, and then completely removing your
d+ebts, you will be able to start fresh. Why keep
dealing with the stress, headaches, and wasted money,
when you can consolidate your d+ebt and pay them off
much sooner!

Click below to learn more:

http://61.186.254.9?affiliateid=mailer10



hjeuubnfs

------------=_3FAE8656.371BED4D--
```

The SpamAssassin report is revealing. Despite the fact that this message includes several tricks to fool spam-checkers, such as random characters at the end and breaking up the words "rates" and "debt" with symbols, SpamAssassin identifies several suspicious characteristics and assigns a high spam score.

How SpamAssassin Works

There are several ways that SpamAssassin makes up its mind about a message:

- The message headers can be checked for consistency and adherence to Internet standards (e.g., is the date formatted properly?).
- The headers and body can be checked for phrases or message elements commonly found in spam (e.g., "MAKE MONEY FAST" or instructions on how to be removed from future mailings)—in several languages.
- The headers and body can be looked up in several online databases that track message checksums of verified spam messages.
- The sending system's IP address can be looked up in several online lists of sites that have been used by spammers or are otherwise suspicious.
- Specific addresses, hosts, or domains can be blacklisted or whitelisted. A whitelist can be automatically constructed based on the sender's past history of messages.
- SpamAssassin can be trained to recognize the types of spam that you receive by learning from a set of messages that you consider spam and a set that you consider non-spam. (SpamAssassin and the spam-filtering community often refer to non-spam messages as *ham*.)
- The sending system's IP address can be compared to the sender's domain name using the Sender Policy Framework (SPF) protocol (*http://spf.pobox.com*) to determine if that system is permitted to send messages from users at that domain. This feature requires SpamAssassin 3.0.
- SpamAssassin can privilege senders who are willing to expend some extra computational power in the form of Hashcash (*http://www.hashcash.org*). Spammers cannot do these computations and still send out huge amounts of mail rapidly. This feature requires SpamAssassin 3.0.

Organization of SpamAssassin

At heart, SpamAssassin is a set of modules written in the Perl programming language, along with a Perl script that accepts a message on standard input and checks it using the modules. For higher-performance applications, SpamAssassin also includes a daemonized version of the spam-checker and a client program in C that can accept a message on standard input and check it with the daemon.

Other Antispam Approaches

SpamAssassin combines message format validation, content-filtering, and the ability to consult network-based blacklists. Filtering systems require little user intervention and introduce little delay into the process of sending and receiving email. There are other approaches to preventing spam, each of which comes with its own advantages and disadvantages (and many of which can be used in addition to, as well as in place of, SpamAssassin).

In a *challenge/response* system, the system holds all messages from unknown senders and sends them a reply message with a unique code or set of instructions (the *challenge*). The senders must reply to the challenge in some fashion that verifies their email addresses and (generally speaking) proves that they are human beings, rather than an automated bulk mail program (the *response*). After a successful response, the system allows messages from the sender to be accepted, rather than holding them.

In *greylisting* systems, the mail server initially returns a temporary SMTP (Simple Mail Transfer Protocol) failure code to messages from new senders or sending systems. If the sending system attempts to resend the message after a reasonable time period, the mail server accepts the message and subsequent messages from the sending host. Because spammers are likely either to treat the temporary failure as a permanent failure, or to attempt to deliver messages continually during the greylisting time period, their messages are not received.

In time-limited address systems, users generate unique variations of their email address to include in different web forms, email messages, newsgroup postings, etc. Addresses may be valid only for a limited time or may be valid until revoked by the user. In these systems, if a user receives spam at one of his addresses, he can usually identify the company that spammed him (or sold his address to a spammer), and he can quickly invalidate the address to prevent further spam.

In micropayment systems, senders must pay a small fee for each message they send, making large-scale spam runs costly. In some of these systems, the micropayment is refunded when the recipient determines that the message is in fact non-spam. (SpamAssassin 3.0 supports a variation of micropayments in the form of Hashcash, in which the payment is made in processing time rather than money.)

Most of SpamAssassin's behavior is controlled through a systemwide configuration file and a set of per-user configuration files. The per-user configuration can also be stored in an SQL database.

For a great deal more about Perl, check out *Learning Perl*, by Randal L. Schwartz and Tom Phoenix, or *Programming Perl*, by Larry Wall, Tom Christiansen, and Jon Orwant, both from O'Reilly.

Mailers and SpamAssassin

Although it's possible to run SpamAssassin manually on a single message, SpamAssassin becomes really useful when all incoming messages are scanned automatically. There are several ways that this can be done.

Figure 1-1 shows a typical mail transmission. The sending system connects to the recipient's mail transport agent (MTA) and transmits the message. If the message is destined for a user on the MTA's system, the MTA hands the message off to the local mail delivery agent (MDA), which is responsible for storing the message in a user's mailbox. Users may log into the system and read their mail directly from their mailboxes (as is typical on multiuser Unix systems), or, if the system runs the appropriate servers, users may download their mail using a mail client that supports the POP (Post Office Protocol) or IMAP (Internet Message Access Protocol) protocols.

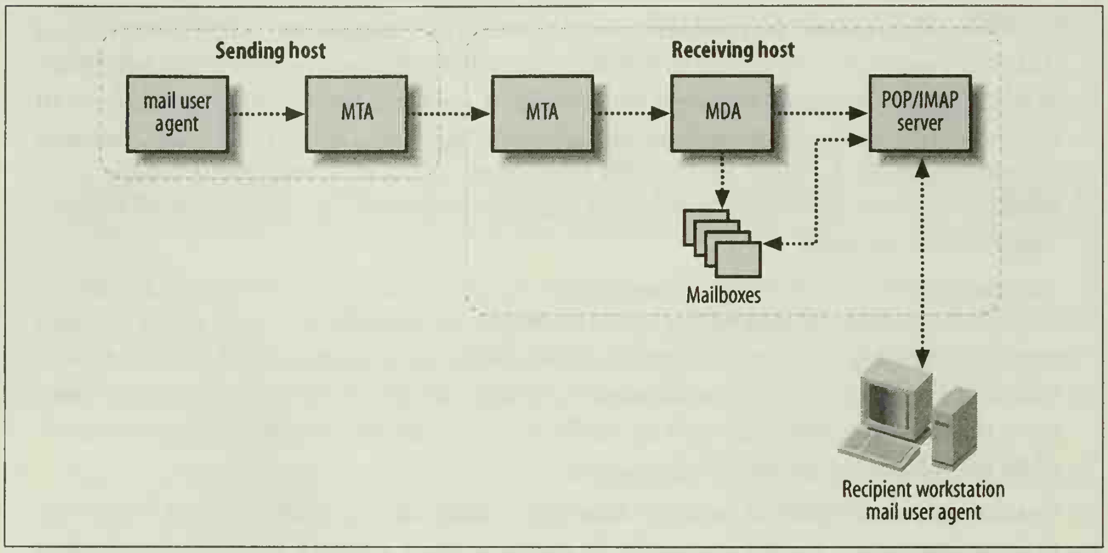


Figure 1-1. A typical mail transmission

SpamAssassin can be run in three fundamental places: at the MTA, at the MDA, and as a POP proxy. Each has advantages and disadvantages.

Scanning at the MTA

Some MTAs provide a way for incoming messages to be passed through a filter during the SMTP transaction; others can pass messages through a filter after the SMTP transaction is complete. Spam-checking is one kind of filtering that can be usefully performed at the MTA; virus-checking is another. In many cases, sophisticated filtering daemons have been developed for specific MTAs, and these daemons are capable of calling SpamAssassin to perform spam checks.

Because all email destined for users on the system must pass through the MTA, it is a natural place for centralized spam-checking. If you run a gateway MTA that delivers mail to several internal systems, you can perform spam-checking at the gateway MTA to limit the amount of spam that any internal server will receive.

In addition to tagging messages that appear to be spam, MTA-based filters can often take other actions, such as blocking a message (either refusing to complete the SMTP transaction or discarding it after the SMTP transaction has taken place) or redirecting it to quarantine area. If the MTA is already running a filtering system to do virus-checking, spam-checking can usually be performed by the same filter and share some of the overhead associated with filtering.

A disadvantage of scanning at the MTA alone is that the MTA filtering system may not be able to access per-user preferences for scanning if the filter does not have access to the recipient information, if the recipient is at another host, or if the message is destined for multiple users on the same system.

Scanning at the MDA

On many Unix systems, the mail delivery agent is procmail, which can submit messages to SpamAssassin and act on the results. This is the most typical way that SpamAssassin is installed alone, as it does not require any MTA-specific filter interfaces.

This configuration maximizes flexibility. Systemwide SpamAssassin rules can be applied to all incoming messages, and users can supplement or modify them with their own per-user SpamAssassin configuration, because, by definition, the MDA always knows the recipient to which it is delivering the message. Users who are proficient in writing procmail recipes gain complete control over the disposition of messages marked as likely spam; procmail can be instructed to discard them, file them in a separate mailbox, modify message headers, or take many other actions.

The downside of this configuration is that spam-checking is applied only after a message has been received by the system and has consumed some system resources. Another disadvantage is that spam-checking must be set up on every system that has local recipients, rather than at a single centralized MTA gateway.

Scanning with a POP Proxy

POP mail users who want the benefits of SpamAssassin on mail servers that don't provide it can use a proxy to perform spam-checking. The proxy runs on the client computer and integrates with the POP mail reader to scan messages as they are downloaded via POP.

The best known POP proxy for SpamAssassin on Windows systems is SAproxy by Stata Labs. SAproxy Pro is a commercial product, but the source code is freely available under the same terms as SpamAssassin itself for administrators who wish to compile it and provide it to their users.

Proxies are the most decentralized approach to spam-checking and require the mail server to be liberal in accepting messages so that each user's proxy can apply their own standards. This may increase the storage load on the mail server. On the other hand, proxies completely remove the computational load from the mail server, as all spam-checking is performed by the client.

Scanning at Multiple Places

It's entirely possible to run SpamAssassin at two or even all three of the places discussed in the previous sections. An MTA-based filter could use SpamAssassin with conservative settings to refuse messages that are highly suspicious. An MDA filter on the same system could apply a more liberal (and per-user) definition of spam in order to tag messages for users who read their mail on the server itself. Finally, POP users could apply their own spam-checking by running SAproxy on their client machines.

The Politics of Scanning

If you're an ISP that provides email service, many of your users will want—perhaps even demand—spam-tagging or spam-filtering of their incoming email. Other users, however, may not want their email tagged or filtered, either because they don't get much spam, don't perceive the spam they receive to be a problem, or are concerned about the possibility of a real message being mistakenly tagged as spam.

Before you implement systemwide or sitewide spam-checking, consider carefully the needs of your users and your responsibilities toward them. At minimum, you must inform users (and would-be users) of any unconditional spam-checking you perform on their email. Better yet is to provide spam-tagging only for those users who opt to turn it on. Best of all is to enable each user to configure their own settings and threshold for how spam is recognized. This is doubly important if you not only tag messages for users but actually filter or block spam for them.

SpamAssassin is an excellent tool for distinguishing spam and non-spam email, but only if you've determined that your users want you to distinguish the two.

CHAPTER 2

SpamAssassin Basics

This chapter explains how to get and install SpamAssassin and its components, perform basic configuration, test the system, and start using it for spam-checking. It covers the basics of using SpamAssassin from the shell or from procmail, and discusses the setup of the daemonized version of the spam-checker. The configuration examples in this chapter provide only the basic functionality. The following chapters cover rule-tweaking, white- and blacklisting, and learning.

Prerequisites

SpamAssassin is written for a Unix or Unix-like environment that includes Perl Version 5, preferably 5.6.1 or later. Perl is now standard on most Unix systems, but if you don't have it, the source code for Perl can be downloaded at *http://www.cpan.org*.

SpamAssassin requires several Perl modules to be installed. If you install SpamAssassin using CPAN (the Comprehensive Perl Archive Network), as described in the next section, these modules will be automatically downloaded and installed as well. If you install SpamAssassin manually, you'll need to be sure that you also have up-to-date versions of the Perl modules *ExtUtils::MakeMaker*, *File::Spec*, *Pod::Usage*, *HTML::Parser*, *Sys::Syslog*, *DB_File*, *Digest::SHA1*, and *Net::DNS*. You may also want *Net::Ident* and *IO::Socket::SSL* if you plan to use the daemonized checker (`spamd`) and its client (`spamc`) and you will allow remote clients to access your daemon.

SpamAssassin can consult several spam checksum clearinghouses. A spam clearinghouse is a server (or a distributed network of servers) that gathers spam messages reported by thousands of users around the world and provides a mechanism for a client to check a new message to see if it matches a message in the clearinghouse. These clearinghouses are known as *checksum*-based clearinghouses because rather than transmit and store complete email messages, they work with cryptographic checksums of messages. A cryptographic checksum is a much smaller data string (typically no more than 256 bits) that is, for all practical purposes, unique to the message from which it is computed.

As of version 3.0, SpamAssassin can consult three clearinghouses: *Vipul's Razor* (*http://razor.sourceforge.net*), *Pyzor* (*http://pyzor.sourceforge.net*), and *DCC* (*http://www.rhyolite.com/anti-spam/dcc/*). SpamAssassin can also be used to report spam to the clearinghouses. Each clearinghouse uses its own client software, and you should install these clients before you install SpamAssassin. In most cases, each SpamAssassin user will have to manually run the clearinghouse's client program to initialize it before SpamAssassin can use it.

> In many sitewide SpamAssassin configurations, you will create a dedicated special user account to run SpamAssassin. If you do and you intend to use spam clearinghouses, be sure that you follow the client software instructions for initialization and that you do so *as the dedicated user*, rather than as *root*.

Building SpamAssassin

The easiest way to download and install SpamAssassin is through CPAN. Here's what a CPAN-install of SpamAssassin looks like:

```
$ su
Password: XXXXXXX
# perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.61)
ReadLine support enabled

cpan> o conf prerequisites_policy ask

  prerequisites_policy ask

cpan> install Mail::SpamAssassin
CPAN: Storable loaded ok
CPAN: LWP::UserAgent loaded ok
Fetching with LWP:
  ftp://ftp.perl.org/pub/CPAN/authors/01mailrc.txt.gz
...
Running install for module Mail::SpamAssassin
Running make for J/JM/JMASON/Mail-SpamAssassin-2.60.tar.gz
Fetching with LWP:
ftp://ftp.perl.org/pub/CPAN/authors/id/J/JM/JMASON/Mail-SpamAssassin-2.60.tar.gz
CPAN: Digest::MD5 loaded ok
Fetching with LWP:
ftp://ftp.perl.org/pub/CPAN/authors/id/J/JM/JMASON/CHECKSUMS
Checksum for /root/.cpan/sources/authors/id/J/JM/JMASON/Mail-SpamAssassin-2.60.tar.gz
ok
Scanning cache /root/.cpan/build for sizes
Mail-SpamAssassin-2.60/
Mail-SpamAssassin-2.60/ninjabutton.png
...
Mail-SpamAssassin-2.60/sample-spam.txt
```

```
  CPAN.pm: Going to build J/JM/JMASON/Mail-SpamAssassin-2.60.tar.gz

What email address or URL should be used in the suspected-spam report
text for users who want more information on your filter installation?
(In particular, ISPs should change this to a local Postmaster contact)
default text: [the administrator of that system] postmaster@example.com

Checking if your kit is complete...
Looks good
Writing Makefile for Mail::SpamAssassin
Makefile written by ExtUtils::MakeMaker 6.03
/usr/bin/perl build/preprocessor -Mconditional -Mbytes -DPERL_VERSION=5.8.0 -Mvars -
DVERSION=2.60 -DPREFIX=/usr <lib/Mail/SpamAssassin/AutoWhitelist.pm >blib/lib/Mail/
SpamAssassin/AutoWhitelist.pm
...
gcc  -g -O2 spamd/spamc.c spamd/libspamc.c spamd/utils.c \
    -o spamd/spamc   -ldl
...
Manifying blib/man3/Mail::SpamAssassin::PerMsgLearner.3pm
  /usr/bin/make  -- OK
Running make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0,
'blib/lib', 'blib/arch')" t/*.t
t/basic_lint...............ok
...
t/zz_cleanup...............ok
All tests successful, 1 test skipped.
Files=40, Tests=301, 426 wallclock secs (238.53 cusr + 14.19 csys = 252.72 CPU)
  /usr/bin/make test -- OK
Running make install
Installing /usr/lib/perl5/site_perl/5.8.0/Mail/SpamAssassin.pm
Installing /usr/lib/perl5/site_perl/5.8.0/Mail/SpamAssassin/PerMsgLearner.pm
...
Installing /usr/bin/spamc
Installing /usr/bin/spamd
Installing /usr/bin/sa-learn
Installing /usr/bin/spamassassin
Writing /usr/lib/perl5/site_perl/5.8.0/i586-linux-thread-multi/auto/Mail/
SpamAssassin/.packlist
Appending installation info to /usr/lib/perl5/5.8.0/i586-linux-thread-multi/
perllocal.pod
/usr/bin/perl "-MExtUtils::Command" -e mkpath /etc/mail/spamassassin
...
  /usr/bin/make install  -- OK

cpan> quit
```

It is also possible to install SpamAssassin manually by downloading the code as a *gzipped tar* archive from *http://www.spamassassin.org* and following these steps from the directory where you keep local source code (*/usr/local/src* on many systems):

```
$ gunzip -c Mail-SpamAssassin-2.60.tar.gz | tar xf -
$ cd Mail-SpamAssassin-2.60
$ perl Makefile.PL
```

```
What email address or URL should be used in the suspected-spam report
text for users who want more information on your filter installation?
(In particular, ISPs should change this to a local Postmaster contact)
default text: [the administrator of that system] postmaster@example.com

Checking if your kit is complete...
Looks good
Writing Makefile for Mail::SpamAssassin
$ make
...compilation mesages...
$ su
Password: XXXXXXXX
# make install
...installation messages...
```

If you install SpamAssassin manually, remember that you may need to install or update other Perl modules listed in the "Prerequisites" section, earlier in this chapter, prior to installing SpamAssassin.

FreeBSD users can install SpamAssassin from the *ports* collection, where it is available both as a traditional port (in which it downloads the source code and compiles it) and as a precompiled package. For example, SpamAssassin 2.63 is included in the collection as *p5-Mail-SpamAssassin-2.63*.

Finally, Linux users can install SpamAssassin in one of several packaged formats. SpamAssassin is available in the Debian GNU/Linux and Gentoo Linux packaging systems as the "spamassassin" and "Mail-SpamAssassin" packages, respectively. Many other distributions of Linux bundle SpamAssassin (although not always the latest version). The latest version of SpamAssassin is also distributed as a source *rpm* by one of its developers. The source *rpm* is used to build three platform-specific *rpm*s that are then installed in the usual way. Example 2-1 shows the process on a RedHat Linux system.

Example 2-1. Building SpamAssassin from source rpm

```
(download spamassassin-2.60-1.src.rpm from http;//w:w.spamassassin.org)
# rpm -Uvh spamassassin-2.60-1.src.rpm
   1:spamassassin          ########################################### [100%]
# cd /usr/src/redhat/SPECS
# rpm -bb spamassassin.spec
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.57624
...
# cd ../RPMS/i386
# ls -l
perl-Mail-SpamAssassin-2.60-1.i386.rpm  spamassassin-tools-2.60-1.i386.rpm
spamassassin-2.60-1.i386.rpm
# rpm -Uvh Perl-Mail-Spam*rpm spamassassin*2.6.0*.rpm
...installation messages...
```

Installing SpamAssassin for Personal Use

If you do not have superuser access on your mail server, but do have a shell account, it is possible to install SpamAssassin into private directories in your account.

Follow the instructions for manual installation and indicate the directory structure you'd like to use for the installation of the program and libraries, and for the configuration files. For example, if you have personal *bin*, *share*, *lib*, and *etc* directories under your home directory, you might use this build process:

```
$ perl Makefile.PL PREFIX=~ SYSCONFDIR=~/etc
$ make
$ make install
```

Note that you must still have the prerequisite Perl modules installed systemwide or you must install them into your private directories as well.

To use a personal installation of SpamAssassin, you will need to make sure that *<PREFIX>/bin* is on your PATH.

What Gets Installed

An installation of SpamAssassin includes the following components:

Perl modules

SpamAssassin's core functions are in a set of Perl modules. The most important of these are *Mail::SpamAssassin*, the top-level module that includes most of the others, and *Mail::SpamAssassin::Conf*, the module that includes documentation of the configuration files for SpamAssassin. These modules are usually installed under a directory with a name like */usr/lib/perl5/site_perl/5.8.1*, but you do not need to know their location, as the Perl installer will ensure that they are installed in a path that Perl will search when loading modules.

SpamAssassin 3.0 introduced a distinction between core SpamAssassin modules and *plug-ins*, modules that may be written for SpamAssassin by third parties and loaded in rulesets. Plug-in modules will have names in the *Mail::SpamAssassin::Plugin* hierarchy (e.g., *Mail::SpamAssassin::Plugin::URIDNSBL*).

Rulesets

The rules that SpamAssassin uses to help decide whether or not a message is spam are kept in a set of configuration files that are usually installed in */usr/share/spamassassin*. You can find the default location of these files by running `spamassassin --local --debug`, but you can always specify alternative locations.

A systemwide configuration file

The systemwide configuration file controls the default behavior of the `spamassassin` (and `spamd`) programs when not overridden by per-user preferences. The file is called *local.cf* and is installed in */etc/mail/spamassassin*. Other applications that use the *Mail::SpamAssassin* modules often put their systemwide configuration files in this directory as well. You can find the default location of these files by running `spamassassin --local --debug`, but you can always specify alternative locations.

`spamassassin`
: The `spamassassin` program is a Perl script that accepts a message on standard input, applies the functions of *Mail::SpamAssassin*, and returns the message on standard output with spam scores, reports, or other modifications added as warranted. It has several other functions as well, which are described in detail later in this chapter. It is usually installed in */usr/bin*.

`spamd` *and* `spamc`
: On sites that receive large amounts of mail, invoking the `spamassassin` script for each message is costly, due to the overhead associated with starting a new process and running the Perl interpreter. `spamd` is a daemon that is started once (at system boot) and remains in memory to perform spam-checking. It listens on either a Unix domain socket or a TCP port to receive requests to check messages, and performs checks; it returns the (possibly modified) messages to the client.

  `spamc` is the client program for sites that run the `spamd` daemon. It accepts a message on standard input, transmits it to `spamd`, and returns the response on standard output. Like `spamassassin`, it is invoked for each message, but it is written in C and compiled, and thus avoids the overhead associated with invoking Perl. It provides the most important functionality of `spamassassin`.

  `spamc` and `spamd` are usually installed in */usr/bin*. They are described in greater detail later in this chapter.

`sa-learn`
: The `sa-learn` script is used to train SpamAssassin's Bayesian spam classification system. It teaches SpamAssassin which messages you consider spam and which you consider non-spam. Eventually, SpamAssassin can use this information to make better judgments of whether or not you want a message marked as spam. SpamAssassin's learning systems are described in detail in Chapter 4.

Basic Configuration

Once SpamAssassin has been installed, it's a good idea to adjust the basic systemwide configuration before testing. A complete guide to the configuration directives is given in Chapter 3; only the most commonly adjusted systemwide directives are described here.

Configuration is usually controlled by the file */etc/mail/spamassassin/local.cf*. Example 2-2 shows a typical *local.cf* that might be used with SpamAssassin 2.63.

Example 2-2. A typical local.cf file

```
# This is the right place to customize your installation of SpamAssassin.
#
# See 'perldoc Mail::SpamAssassin::Conf' for details of what can be
# tweaked.
#
###########################################################################

# How high a score is considered spam?
required_hits 5

# How should spam reports be inserted into the message?
report_safe 1

# Should we tag the subject of spam messages?
rewrite_subject 1

# By default, SpamAssassin will run RBL checks.  If your ISP already
# does this, set this to 1.
skip_rbl_checks 0
```

Blank lines and lines beginning with a number sign (#) are ignored in configuration files. Other lines begin with a configuration directive (e.g., `required_score`), followed by whitespace and then the value for the directive (e.g., `5`).

The directives you will most want to adjust are:

`required_hits` *(SpamAssassin 2.63) or* `required_score` *(SpamAssassin 3.0)*
: Each SpamAssassin rule that matches a message adds (or subtracts) points from the message's total spam score. When the total score reaches the value of this directive, SpamAssassin reports the message as spam. The default value, 5, is suitable for most installations. If you are particularly worried about false positives, you can increase this value, which will also have the effect of reducing the number of true positives (i.e., some spam will be missed).

`report_safe`
: This directive determines how SpamAssassin modifies messages that it determines are spam.

  No matter how `report_safe` is set, SpamAssassin adds three headers to spam mail: *X-Spam-Level* (set to a number of asterisks representing the spam score), *X-Spam-Status* (set to a one-line description of the spam score and matching tests), and *X-Spam-Flag* (set to Yes).

  When `report_safe` is set to 0, the message body is kept intact, and the header *X-Spam-Report* is added with a detailed description of the rules that matched. When `report_safe` is set to 1, a new MIME message is created with the spam report as an attachment and the original spam message as an attachment with content-type *message/rfc822*. When `report_safe` is set to 2, SpamAssassin behaves similarly, but the original spam message is attached with content-type *text/plain*.

`rewrite_subject`*(SpamAssassin 2.x only)*
: If this directive is set to 1, SpamAssassin will prepend "*****SPAM*****" to the message subject in the *Subject* header if the message is considered spam. This is useful when users have mail clients that can filter only on standard headers.

`rewrite_header`*(SpamAssassin 3.0 only)*
: This directive can be used to rewrite the *Subject*, *From*, or *To* headers of messages that SpamAssassin considers spam. Rewriting the *Subject* header prepends a given string to the message subject. For example, to prepend "*****SPAM*****" to a spam message's subject, use the following:

```
rewrite_header subject *****SPAM*****
```

Rewriting *From* or *To* headers adds the given string to the email address as a parenthetical comment.

`skip_rbl_checks`
: SpamAssassin typically looks up a sender's IP address in a set of Domain Name System (DNS)-based real-time blacklists (DNSBLs or RBLs) to determine whether they have been listed as known spam source, open proxy or relay, dialup host, etc. Many ISPs perform these checks in the MTA itself in order to reject connections from such hosts at the earliest possible point. If you do that, you can prevent SpamAssassin from doing its own lookups by setting this directive to 1; the default is 0. It is also possible to perform lookups against one set of DNSBLs at the MTA and a different set in SpamAssassin.

Testing SpamAssassin

Once the basic systemwide configuration is in place, it's a good idea to test SpamAssassin to ensure that it can correctly distinguish a known non-spam message from a known spam message. To facilitate this, the SpamAssassin source code includes two files, *sample-nonspam.txt* and *sample-spam.txt*. The former contains an email message that has very few hallmarks of spam; the latter contains an email message that includes the GTUBE (Generic Test for Unsolicited Bulk Email) string, a special test string that is used to validate spam-checkers.

> If you installed SpamAssassin using CPAN, you'll find the *sample-nonspam.txt* and *sample-spam.txt* files in whichever directory CPAN performs its builds. Often that will be a subdirectory of *root*'s home directory named *.cpan/build/Mail-Spamassassin-2.63*.

To test the `spamassassin` script, run it in test mode by using the `--test-mode` command-line argument and provide one of the sample files on its standard input. In test mode, spamassassin will produce a spam score at the bottom of the message whether or not the message meets the required score for spam. Example 2-3 shows a test of spamassassin on the *sample-nonspam.txt* file, which produces a final score of 0.0.

Example 2-3. Testing spamassassin with sample-nonspam.txt

```
$ cd Mail-SpamAssassin-2.63
$ spamassassin --test-mode < sample-nonspam.txt
Return-Path: <tbtf-approval@world.std.com>
Delivered-To: foo@foo.com
Received: from europe.std.com (europe.std.com [199.172.62.20])
        by mail.netnoteinc.com (Postfix) with ESMTP id 392E1114061
        for <foo@foo.com>; Fri, 20 Apr 2001 21:34:46 +0000 (Eire)
...
Content preview:  -----BEGIN PGP SIGNED MESSAGE----- TBTF ping for
  2001-04-20: Reviving T a s t y B i t s f r o m t h e T e c h n o l o g
  y F r o n t [...]

Content analysis details:   (0.0 points, 5.0 required)

 pts rule name              description
---- ---------------------- --------------------------------------------------
 0.0 LINES_OF_YELLING       BODY: A WHOLE LINE OF YELLING DETECTED
```

Example 2-4 shows the same test using *sample-spam.txt*, which produces a final score of 1000.

Example 2-4. Testing spamassassin with sample-spam.txt

```
$ spamassassin --test-mode < sample-spam.txt
Received: from localhost [127.0.0.1] by tala.mede.uic.edu
        with SpamAssassin (2.60 1.212-2003-09-23-exp);
        Sun, 16 Nov 2003 21:38:03 -0600
...
Content preview:  This is the GTUBE, the Generic Test for Unsolicited
  Bulk Email. If your spam filter supports it, the GTUBE provides a test
  by which you can verify that the filter is installed correctly and is
  detecting incoming spam. You can send yourself a test mail containing
  the following string of characters (in uppercase and with no white
  spaces and line breaks): [...]

Content analysis details:   (1000.0 points, 5.0 required)

 pts rule name              description
---- ---------------------- --------------------------------------------------
1000 GTUBE                  BODY: Generic Test for Unsolicited Bulk Email
```

If these tests succeed, you might try testing with a few real spam and non-spam messages from your mailbox to get a feel for how the scoring works.

SpamAssassin Options

The `spamassassin` script has a large number of command-line options that control its behavior. Some of the most commonly used for spam-checking are detailed here; others are featured in Chapter 3 and Chapter 4. A complete list of options can be found in the man page for `spamassassin`.

Locating configuration files

SpamAssassin expects to find its rulesets in */usr/share/spamassassin*, its systemwide configuration file at */etc/mail/spamassassin*, and per-user preferences in *~/.spamassassin/user_prefs*. If you've installed SpamAssassin in different locations, you may need to use these command-line options to help the `spamassassin` script locate these files.

`--configpath` */path/to/ruleset/directory*
: Specifies the path to the directory containing the SpamAssassin ruleset configuration files. This option also can be called as `--config-file` or `--config-dir`.

`--siteconfigpath` */path/to/sitewide/directory*
: Specifies the path to the directory containing the sitewide configuration file *local.cf*.

`--prefspath` */path/to/user_prefs*
: Specifies the path to the file containing user preferences for the user running `spamassassin`. `--prefs-file` can also be used.

Scripting and testing options

Two `spamassassin` options are useful in scripting.

`--exit-code` *[integer]*
: When this option is used, the `spamassassin` script will exit with a nonzero exit code if the message it checked was determined to be spam, and a zero exit code if it was not. The default spam exit code is 5, but you can specify one as an argument to this option. If `spamassassin` exits due to a program error, it returns exit code 64 (if bad arguments were given to `spamassassin`) or 70 (for other errors).

  This option provides a useful way for a calling script to determine if a message is considered spam.

`--log-to-mbox` */path/to/mbox/file* *(SpamAssassin 2.x only)*
: This option causes copies of all of the messages processed by `spamassassin` to be logged to the given file in *mbox* format. The messages are logged in the form in which `spamassassin` receives them, with no spam-tagging. This option can be used to preserve pristine copies of email, but such a function is probably better performed by the MTA itself, rather than by SpamAssassin.

Untagging

No spam-checking system is perfect. If SpamAssassin mistakenly tags a non-spam message as spam, it will add several message headers and reformat the message to include its report as the first MIME attachment and the original message as a second attachment. To remove these headers and restore the message to a near-original state, pipe the message to `spamassassin` with the `--remove-markup` option, as shown in Example 2-5.

Example 2-5. Removing SpamAssassin markup

```
$ spamassassin < sample-spam.txt > marked-message
$ spamassassin --remove-markup < marked-message > unmarked-message
$ diff -s sample-spam.txt unmarked-message
Files sample-spam.txt and unmarked-message are identical
```

Messages that have been tagged and then untagged via `--remove-markup` may differ in minor ways from the original message. For example, headers that may have included line breaks in the original message may be concatenated into one long line.

Reporting

If you've installed clients for spam checksum clearinghouses, you can report spam to those clearinghouses by piping a message to `spamassassin --report`. The message will be untagged before being reported. In SpamAssassin 2.63, if you also provide the `--warning-from=`*`emailaddress`* option, the sender of the spam will receive an email (apparently from the provided *`emailaddress`*) warning her that her message has been reported as spam. This is rarely useful (because most spam forges or obfuscates the sender's address), and this option has been removed in SpamAssassin 3.0.

You can also use SpamAssassin's reporting capability to set up *spam traps*. A spam trap is an email address that has never been used by a real recipient and never requests email from anyone. People who set up spam trap addresses often include the addresses on web pages or in Usenet postings with instructions that people should not send mail to the addresses—instructions that spammers' address-harvesting programs will ignore. Because any email that's sent to the spam trap address can be safely assumed to be spam, you can report it as such to spam clearinghouses. To set up a spam trap with SpamAssassin, create an email alias that pipes messages to `spamassassin --report`. For most clearinghouse systems, you will need to determine which user your mail system will invoke `spamassassin --report` as and set up some files in that user's home directory to control how it will interact with the clearinghouse client. See your clearinghouse documentation for details.

Never report spam sent to a legitimate address that you have not verified with your own eyes. The clearinghouse systems rely on these spam reports, and their effectiveness is diminished when non-spam messages are reported as spam. If you do accidentally report a non-spam message, you can revoke your report by piping the message to `spamassassin --revoke`. Not all clearinghouses support message revocation. As of SpamAssassin 3.0, only Vipul's Razor does.

Invoking SpamAssassin with procmail

Running `spamassassin` from a shell is a handy way to test the system, but for daily use you'd like to have it automatically run on every incoming email message that's being delivered to your system's mailboxes. One easy way to do this is to have your system's MDA program filter all messages through SpamAssassin as part of the delivery process.

procmail is a mail-processing program that accepts messages on standard input and applies a set of rules or actions (a "recipe") for the disposition of the message. Because the default message disposition is "append to the user's mailbox," and because procmail is written to be very safe in its handling of messages, it makes an excellent MDA. Indeed, many Unix systems use the procmail program as their default local MDA. If procmail is available and isn't the system MDA, it's usually easy for users to configure the message-forwarding feature of the system's MTA to filter messages through procmail. In either environment, procmail can be a good place to pass messages through SpamAssassin. Figure 2-1 illustrates this configuration.

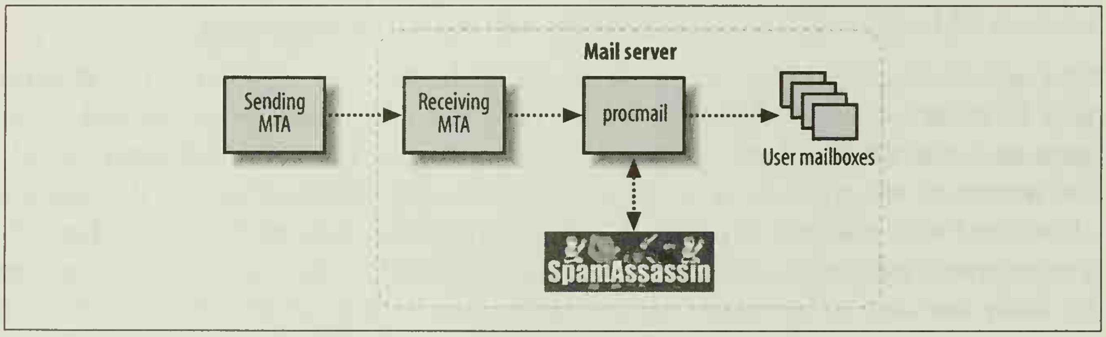


Figure 2-1. Invoking SpamAssassin with procmail

The easiest way to use SpamAssassin with procmail is to call it in the systemwide procmail recipe file, which is usually */etc/procmailrc*. Example 2-6 shows a complete */etc/procmailrc*.

Example 2-6. A complete /etc/procmailrc

```
DROPPRIVS=yes
PATH=/bin:/usr/bin:/usr/local/bin
SHELL=/bin/sh

# Spamassassin
:0fw
* <300000
|/usr/bin/spamassassin
```

In this example, the SpamAssassin recipe comprises the three lines beneath the comment `# Spamassassin`. The first line tells procmail that the message should be filtered (`f`) and that procmail should wait (`w`) for the filter's successful exit before considering the message filtered. The second line indicates that this recipe should be applied to messages less than 300,000 bytes in length and serves to prevent a lengthy SpamAssassin invocation on a long message that is unlikely to be spam. The third line directs procmail to pipe the message to `spamassassin`. (For more information about procmail recipes, see the man pages for procmail, procmailrc, and procmailex.)

By placing this recipe in the systemwide procmail configuration file, it will be activated every time procmail is invoked, either as the default MDA or by a user. If you don't have access to the systemwide procmail configuration file, you can still invoke SpamAssassin for your own messages in your account's per-user procmail recipe file, which is usually *~/.procmailrc*. This might also be useful if you wish to run SpamAssassin a second time with a different set of command-line arguments.

If your system doesn't provide procmail, it may provide another mail-filtering system. Any mail filter that can pass a message to a program on standard input and read back the (modified) message from the program's standard output can use SpamAssassin in this way.

Using spamc/spamd

If you are filtering a lot of incoming mail, the processing time required to invoke a new `spamassassin` script (and starting the Perl interpreter) for each message can become prohibitive. An alternative approach is to run the SpamAssassin daemon, `spamd`. `spamd` is started once at system boot and loads the SpamAssassin Perl modules to perform spam-checking. Instead of running the `spamassassin` script on each message, messages are piped to the `spamc` program. `spamc` is a lightweight client, written in C and compiled to an executable that simply takes messages, relays them to `spamd`, and returns the results.

`spamd` has several important command-line arguments that control its operation. Once it's properly set up, however, using `spamc` is simple.

Setting up spamd

By default, spamd is installed in */usr/bin*. It is typically started by *root* from a system boot script but can also be started *by root* from the shell for testing. The simplest invocation of spamd is:

```
/usr/bin/spamd --daemonize --pidfile /var/run/spamd.pid
```

The `--daemonize` command-line option directs spamd to operate as a daemon in the background. The `--pidfile` command-line option specifies the file to which spamd will write its process ID number. This option is important because spamd must be signaled with a HUP signal to its process ID whenever the systemwide SpamAssassin configuration is changed (you'll find an example later in this chapter).

When spamd receives a connection, it forks a child process to handle the connection. Typically, the child process reads a request to perform spam-checking from the client (including the account name of the user making the request, the message to check, and other data), performs the requested check, returns the (possibly tagged) message back to the client, and exits.

Several options are used with spamd in many environments. The most common are detailed in the following sections.

Connection type

spamd can accept connections from spamc clients either by listening on a TCP port or a Unix domain socket. By default, spamd binds TCP port 783 on the local 127.0.0.1 IP address, which should prevent remote users from connecting to it. You can change how it listens with these command-line options:

`--socketpath` *`/path/to/socket`*
: Listen on a Unix domain socket at the specified path instead of a TCP port. Using a Unix domain socket is more efficient than a TCP port and ensures that only local users can access the daemon.

`--listen-ip` *`ip-address`*
: Listen on a TCP port on the specified IP address. This can be used to override the default 127.0.0.1 IP address and allow spamd to receive connections from remote machines. This might be useful if you wanted to dedicate a single machine in a LAN to spam-checking in order to manage the processing load or to let many client machines share a well-tuned daemon.

`--port` *`port-number`*
: Listen on a TCP port other than the default port (783).

`--allowed-ips` *`ip-address,ip-address,...`*
: Specify a comma-separated list of IP addresses from which connections will be accepted. Although this provides a measure of access control for a daemon that

accepts remote connections, it should be supplemented with host-based firewall rules for greater security.

`--ssl`
: Require connections from clients to use the SSL/TLS (Secure Sockets Layer/Transport Layer Security) protocol. This provides for encryption of the data between client and server and potentially for authentication of the server to the client, although SpamAssassin's `spamc` does not attempt to verify the server certificate.

`--server-key` *`keyfile`*
: Specifies the file containing the SSL private key for `spamd`, if SSL connections are to be required.

`--server-cert` *`certfile`*
: Specifies the file containing the SSL certificate for `spamd`, if SSL connections are to be required.

If you want to provide secure remote access to `spamd`, the SSL support in `spamd/spamc` is not sufficient, as it provides no mechanism for `spamd` to authenticate `spamc` clients. An alternative approach would be to wrap the server and client connections in an SSL tunnel with a program like *stunnel* that does provide two-way authentication.

Running as a non-root user

You must start `spamd` as *root* so that it can bind its TCP port or open its socket for connections. By default, `spamd` continues to run as *root*. When it receives a connection from `spamc`, it drops privileges and runs as the user that `spamc` claims to be running as. This enables it to access private, per-user configuration files.

Many system administrators are uncomfortable running `spamd` as *root*. A bug in `spamd` could provide an attacker with *root* privileges; a local attacker could also spoof `spamc` and claim to be a different user (which can be ameliorated with the `--auth-ident` option discussed later).

To provide additional security, `spamd` can be instructed to run as a non-*root* user. After binding its TCP port or Unix socket, `spamd` gives up *root* privileges and runs as the specified user. Ideally, you should create a new user (e.g., *spamd*) with its own group (*spamd*) and a private home directory (*/home/spamd*). All systemwide configuration files should be made readable by the new user, and the pid file given to the `--pidfile` command-line option should be in a directory writable by the new user (perhaps its home directory). If `spamd` is using a Unix domain socket, the socket will automatically have its owner set to the new user, so no changes to this path are necessary, but the directory in which the socket will be created must be writable by the user.

After creating your new user, start `spamd` like this, as *root*:

```
/usr/bin/spamd --daemonize --username spamd --pidfile /home/spamd/spamd.pid
```

The `--username` command-line option specifies the name of the user that `spamd` will run as.

If you want to allow per-user configuration, users' home directories and *.spamassassin* subdirectories will have to be searchable by the new user (which typically means they must be world-searchable), and files in their *.spamassassin* directories will have to be readable by the new user. Alternatively, you can turn off per-user configuration with the `--nouser-config` command-line option (or store per-user configuration in an SQL database, as discussed in Chapter 3).

> You can also run `spamd` as a non-*root* user simply by starting it as a non-*root* user. In this case, the user running `spamd` must be able to read all of the relevant system configuration files, and you must specify a port number higher than 1024 (or a Unix domain socket in a directory the `spamd` user can write in).

Other security features

Three command-line options provide additional assurances that `spamd` will operate only when the user running `spamc` is actually the user that `spamc` claims to be running for.

`--auth-ident`
: This option causes `spamd` to perform an ident (RFC 1413) lookup on the connection. If the client's system is running a (trustworthy) ident server, the lookup will return the username of the user running `spamc`. `spamd` will confirm that this username matches the username provided by `spamc` and will refuse to respond if it does not.

`--ident-timeout` *`number-of-seconds`*
: Specify the number of seconds to wait for the ident server to respond. If the response doesn't come after this number of seconds, `spamd` will refuse to perform spam-checking for the connection.

`--paranoid`
: Specify that `spamd` should report an error and exit if it finds itself still running as *root* after it should have changed to a non-*root* user ID (either the one given by `--username` or the user running `spamc`), or if it cannot look up a given user's name. Without this option, `spamd` continues running as the *nobody* user.

One command-line option can protect `spamd` from being used to commit a denial-of-service attack against its server.

`--max-children` *`number`*
: Specifies the maximum number of child processes that `spamd` will fork. When this maximum is reached, connections will be queued until the number of children drops below the maximum again (or until the operating system can no

longer queue connections). If max-children is used, spamd must open pipes to communicate with each child.

In SpamAssassin 3.0, the --max-children option defaults to 5, but in SpamAssassin 2.x, the default number of children is unlimited. I highly recommend explicitly setting --max-children to a reasonable value for your system.

Here's what a typical invocation of spamd might look like for a system that is only performing spam-checking for local users and that runs an ident server:

```
/usr/bin/spamd --daemonize --username spamd --pidfile /home/spamd/spamd.pid --auth-
ident --paranoid --max-children=25
```

Locating configuration files

Like SpamAssassin, spamd looks for rulesets in */usr/share/spamassassin* and system-wide configuration files in */etc/mail/spamassassin*. If you've installed SpamAssassin in different locations, you can use the --configpath and --siteconfigpath command-line options to help spamd locate these files. These options work just as they do for the spamassassin script and were described earlier.

Testing spamc

Once spamd is running, use spamc instead of the spamassasin script to check a mail message. You can test spamc/spamd much as you would test spamassassin:

```
$ cd Mail-SpamAssassin-2.63
$ spamc -c < sample-nonspam.txt
0.0/5.0
$ spamc -c < sample-spam.txt
1000.0/5.0
```

The -c command-line option instructs spamc to produce only the score (and the spam threshold score) that spamd computes for each message. It also causes the spamc process to return an exit code of 1 for messages judged to be spam and 0 for messages judged not to be spam, which can be useful in scripting.

spamc Options

Like the spamassassin script, spamc takes several command-line options that modify its behavior. Here are some of the most useful (see the manpage for spamc for a complete list).

Connection type

By default, spamc attempts to connect to spamd at TCP port 783 on localhost. If you run spamd on a different IP address (perhaps on a different machine altogether) or listening on a Unix domain socket, spamc must be told where to connect.

spamc can take advantage of multiple spamd servers at different hosts to increase reliability or balance the processing load. In addition to specifying the proper command-line options to spamc (descriptions follow), you must designate a hostname in DNS with multiple A records, each listing the IP address of a spamd server host.

These command-line options control the spamc connection to spamd.

`-d` *`host`*
: Connect to the spamd server on *`host`*, instead of *localhost*. If *`host`* is a hostname that resolves to multiple IP addresses, each one will be tried in turn until a successful connection can be made.

`-p` *`port`*
: Specify the TCP port number to connect to spamd on. If multiple servers are used, all servers must use the same port number.

`-H`
: When multiple spamd servers are used, try servers in random order instead of the order in which they are returned by the DNS server. This promotes load-balancing across the servers.

`-S`
: Make connections to spamd with SSL. If multiple spamd servers are used, all servers must support SSL connections.

`-U` *`/path/to/socket`*
: Specify a Unix domain socket to connect to spamd on, instead of using TCP.

Handling problems

By default, if spamc is unable to contact a spamd server, it returns the message unprocessed. This ensures that mail will not be lost due to problems with spamd but means that spam may be accepted without tagging. Two command-line options modify this behavior.

`-t` *`number-of-seconds`*
: Specifies the number of seconds that spamc should wait for a reply from spamd before considering the spamd server unreachable. It defaults to 600 seconds (10 minutes), which may be too long to wait on a busy mail server. Setting the *`number-of-seconds`* to 0 disables the timeout altogether—spamc will wait as long as it takes (and potentially forever).

`-x`

This option prevents spamc from returning messages unprocessed when it can't contact a spamd server. Instead, spamc will exit with an error code. Ideally, whatever process is calling spamc will interpret this error code properly, and the message will be queued for later retry. This option requires great care.

spamc's options are different than those accepted by spamassassin, so it is not generally possible to simply substitute spamc for spamassasin in scripts without reviewing each option. Some of the options to spamassassin are instead given as options to spamd when it is started.

Invoking spamc with procmail

Just as spamc is run manually in place of the spamassassin script, it can also be run in a procmail recipe. Example 2-7 shows a typical */etc/procmailrc* recipe for a system using spamd:

Example 2-7. A complete /etc/procmailrc for spamd

```
DROPPRIVS=yes
PATH=/bin:/usr/bin:/usr/local/bin
SHELL=/bin/sh

# Spamassassin
:0fw
* <300000
|/usr/bin/spamc
```

Changing SpamAssassin Configuration Files

To increase efficiency, spamd caches the spam-checking rules in memory when it starts up. Therefore, when spamd is in use, the daemon must be signaled whenever you make changes to the SpamAssassin rulesets or systemwide configuration file. Changes in user preferences do not require a signal because user preference files, if they are used, are reread each time they are needed.

spamd reloads configuration files when it receives a HUP signal. To send a process a HUP signal, read the process ID from the pidfile and use the kill command to send the signal:

```
# kill -HUP `cat /home/spamd/spamd.pid`
```

If you can't find the pidfile, use the ps command to locate the process ID:

```
# ps auxw | grep spamd              (On SysV systems, ps elf)
spamd    30124  0.0  0.6 22200 1596 ?        S    Nov22   0:02 usr/bin/spamd --
daemonize --username spamd --pidfile /home/spamd/spamd.pid
alansz   30521  0.0  0.1  1520  508 pts/1    S    15:44   0:00 grep -E spamd
# kill -HUP 30124
```

After reloading, spamd will have a new process ID.

Invoking SpamAssassin in a Perl Script

Because the heart of the SpamAssassin system is a set of Perl modules, it's fairly straightforward to call SpamAssassin from a Perl script to perform spam-checking of an email message. The *Mail::SpamAssassin* module (and its submodules) provide an object-oriented interface to the spam-checking and message-tagging logic. Many MTA-based filtering systems are written in Perl, and use the SpamAssassin modules to perform spam-checking on messages without invoking a separate program.

Examples 2-8 and 2-9 show Perl scripts that work like simple versions of the spamassassin script, accepting a message on standard input, checking it, and producing the (possibly rewritten) message on standard output. Example 2-8 illustrates the process for SpamAssassin 2.63.

Example 2-8. Using Mail::SpamAssassin 2.63 in Perl

```
#!/usr/bin/perl

use Mail::SpamAssassin;

my @lines = <STDIN>;
my $mail = Mail::SpamAssassin::NoMailAudit->new(data => \@lines);
my $spamtest = Mail::SpamAssassin->new( );
my $status = $spamtest->check($mail);
$status->rewrite_mail( ) if $status->is_spam( );
print $status->get_full_message_as_text( );
```

Before any SpamAssassin objects can be created, the script must use the *Mail::SpamAssassin* module. The message is read from standard input and saved to the array @lines. Then, the new( ) method of *Mail::SpamAssassin::NoMailAudit* is called, with a reference to the array provided as the value of the data parameter.* This method returns a *Mail::SpamAssassin::Message* object encapsulating the email message, which I call $mail in the example.

A new *Mail::SpamAssassin* object called $spamtest is then created, and its check( ) method is called, passing in the message as an argument. check( ) returns a *Mail::SpamAssassin::PerMsgStatus* object, called $status in the script, that contains a copy of the message as well as the results of the spam check. In particular, the is_spam( ) method of $status returns 1 if the message was judged to be spam, and 0 otherwise.

* On systems with the *Mail::Audit* module, *Mail::SpamAssassin* 2.x can be used as a plug-in for *Mail::Audit*. See the documentation for both modules for details. SpamAssassin 3.0 no longer supports *Mail::Audit*, however; so this approach should be avoided for new installations.

If the message was spam, the `rewrite_mail()` method of the `$status` object is called and performs the complete SpamAssassin tagging process on the message, including adding relevant headers and MIME-encapsulating a spam report and the original message. Finally, the script prints the message to standard output by calling the `get_full_message_as_text()` method of `$status` and printing the result.

Example 2-9 illustrates the process for SpamAssassin 3.0.

Example 2-9. Using Mail::SpamAssassin 3.0 in Perl

```
#!/usr/bin/perl

use Mail::SpamAssassin;

my @lines = <STDIN>;
my $spamtest = Mail::SpamAssassin->new( );
my $mail = $spamtest->parse(\@lines);
my $status = $spamtest->check($mail);
print $status->rewrite_mail( );
```

Before any SpamAssassin objects can be created, the script must use the *Mail::SpamAssassin* module. The message is read from standard input and saved to the array `@lines`. Then, the `new( )` method of *Mail::SpamAssassin* is called to create a new `Mail::SpamAssassin` object named `$spamtest`.

The `parse( )` method on `$spamtest` is invoked and passed a reference to the array of message lines. This method returns a *Mail::SpamAssassin::Message* object encapsulating the email message, which I call `$mail` in the example.

Next, `$spamtest`'s `check( )` method is called, passing in the message as an argument. `check( )` returns a *Mail::SpamAssassin::PerMsgStatus* object, called `$status` in the script that contains a copy of the message as well as the results of the spam check.

Finally, the `rewrite_mail( )` method of the `$status` object is called, which performs the complete SpamAssassin tagging process on the message, including adding relevant headers and, if the message is spam, MIME-encapsulating a spam report and the original message. The return value of `rewrite_mail( )` is the rewritten message, so the script prints it to standard output.

As these scripts illustrate, simple spam-checking is easily added to Perl scripts that process email messages. The options to the `spamassassin` script are all available through Perl either as arguments that can be passed to the *Mail::SpamAssassin* constructor (e.g., to specify the location of the sitewide configuration file) or as methods of the *Mail::SpamAssassin::PerMsgStatus* object (e.g., to get the spam score or the specific tests that were triggered). The manual (or *perldoc*) pages for *Mail::SpamAssassin*, and *Mail::SpamAssassin::PerMsgStatus* provide complete details. Other SpamAssassin modules support SpamAssassin's advanced features, such as learning, and are also documented with *perldoc*.

SpamAssassin and the End User

The discussion so far in this chapter has focused on getting SpamAssassin to analyze incoming mail and mark spam by modifying the message before delivery. For end users who read their email on the server or download it with a POP or IMAP client, the final step is to take action on messages. Messages processed through SpamAssassin fall into one of the categories described in the next four sections.

True Negatives (ham)

True negatives are messages that both you and SpamAssassin agree are non-spam, or *ham*, messages. SpamAssassin does not modify these messages much. It adds an *X-Spam-Status* header beginning with the word "No," and an *X-Spam-Checker-Version* header giving the version of SpamAssassin in use. These messages look just as they should to a user's mail reader.

True Positives (spam)

True positives are messages that both you and SpamAssassin agree are spam. These messages are tagged by SpamAssassin. At minimum, SpamAssassin adds *X-Spam-Level*, *X-Spam-Status*, and *X-Spam-Flag* headers. If `rewrite_subject` is on, SpamAssassin also changes the subject of the message to begin with *****SPAM*****. Example 2-10 shows these headers.

Example 2-10. Headers added to spam by SpamAssassin

```
Subject: *****SPAM***** Live your dream life!!              MPNWSTU
X-Spam-Status: Yes, hits=12.9 required=5.0 tests=CLICK_BELOW,
        FORGED_MUA_EUDORA,FROM_ENDS_IN_NUMS,MISSING_OUTLOOK_NAME,
        MSGID_OUTLOOK_INVALID,MSGID_SPAM_ZEROES,NORMAL_HTTP_TO_IP,
        SUBJ_HAS_SPACES,SUBJ_HAS_UNIQ_ID autolearn=no version=2.60
X-Spam-Flag: YES
X-Spam-Checker-Version: SpamAssassin 2.60 (1.212-2003-09-23-exp)
X-Spam-Level: ************
```

Most people will want either to complain about spam to the spammer's ISP or to discard it. In the former case, simply being able to quickly identify spam messages on sight is usually sufficient, and the modified *Subject* header makes that simple. If the user is reading his mail on a system with the `spamassassin` script and applications for distributed spam clearinghouses, he can pipe the message to `spamassassin --report` to report the message to the clearinghouses.

In the latter case, that of wanting to discard spam, users can set up their personal mail filters to delete spam or save it to a "spam" mailbox that they can check now and then. Users on shell accounts with procmail might use the following recipes in their *~/.procmailrc* file:

```
:0
* ^X-Spam-Level: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
/dev/null

:0
* ^X-Spam-Flag: YES
spambox
```

The first recipe checks to see if the message has at least 15 asterisks in the *X-Spam-Level* header. These messages are very likely to be true positives and are discarded by delivering them to */dev/null*. The second recipe catches all other messages that SpamAssassin considers spam (e.g., with scores between 5.0 and 14.99) and saves them to a separate mailbox file called *spambox*.

Users of POP mail clients can use their client's filtering capabilities. Nearly all modern POP mail clients provide the ability to filter messages based on strings contained in the *Subject* header, so spam can be redirected by checking the *Subject* for *****SPAM*****. Some POP clients provide greater control over filtering and allow checking arbitrary headers; these clients can do the equivalent of the preceding procmail recipes.

False Positives

False positives are the bane of all spam-checkers. A false positive occurs when SpamAssassin incorrectly marks a message as spam that you actually wanted to receive. Because of the potential for false positives, it's a good idea to encourage users to think of SpamAssassin's tags as advisory and to avoid discarding messages unseen on the basis of a spam classification by SpamAssassin. Instead, as illustrated in the earlier section on true positives, spam can be filtered to a special spam mailbox that the user can check periodically to ensure that it does not contain any false positives.

If you're reading email on a system that has the `spamassassin` script and you find a false positive, you can pipe the message through `spamassassin --remove-markup` to remove the SpamAssassin report and restore the message to its untagged state.

Identifying false positives and reporting them to SpamAssassin is key to improving SpamAssassin's Bayesian classifier. The Bayesian classifier is discussed in detail in Chapter 4.

False Negatives

A false negative is a missed spam. It occurs when SpamAssassin fails to tag a message as spam that you actually consider spam. The more false negatives you get, the less effective the spam-checking is in saving you time. You can reduce false negatives by lowering SpamAssassin's threshold score, but you will increase false positives at the same time. Keeping track of false negatives can help you find patterns that may let you tweak SpamAssassin's rules to match your environment more closely.

As with true positives, if the user is reading her mail on a system with the `spamassassin` script and applications for distributed spam clearinghouses, she can pipe the message to `spamassassin --report` to report the message to the clearinghouses."

Identifying false negatives and reporting them to SpamAssassin is key to improving SpamAssassin's Bayesian classifier. The Bayesian classifier is discussed in detail in Chapter 4.

Measuring SpamAssassin's Performance

One of the ways that SpamAssassin's developers measure SpamAssassin's performance is by running SpamAssassin on large corpora of messages that are known to be spam or non-spam and measuring the rate of true and false positives and negatives at different thresholds (from –4 to 20) and with different features enabled. The results of these tests are distributed in the *rules* directory in files *STATISTICS.txt* (statistics without network or Bayesian tests), *STATISTICS-set1.txt* (statistics with network tests but no Bayesian tests), *STATISTICS-set2.txt* (statistics with Bayesian tests but no network tests), and *STATISTICS-set3.txt* (statistics with both network and Bayesian tests).

Here's an example of the contents of *STATISTICS-set3.txt* showing performance with a spam threshold of 5.0:

```
# SUMMARY for threshold 5.0:
# Correctly non-spam:  15550  46.59%  (99.90% of non-spam corpus)
# Correctly spam:      17648  52.87%  (99.08% of spam corpus)
# False positives:        15  0.04%  (0.10% of nonspam,   1133 weighted)
# False negatives:       164  0.49%  (0.92% of spam,    437 weighted)
# TCR: 74.527197  SpamRecall: 99.079%  SpamPrec: 99.915%  FP: 0.04%  FN: 0.49%
```

With those features and that threshold, SpamAssassin had a true positive rate of 99. 08%, a true negative rate of 99.9%, a false positive rate of 0.1%, and a false negative rate of 0.92%.

CHAPTER 3

SpamAssassin Rules

SpamAssassin performs its spam-checking by applying a series of tests to an email message. Most tests examine the message headers or body for patterns that are suggestive of spam; others perform Internet lookups against network-based blacklists of IP addresses or checksums of spam messages. Each positive test yields a score, and the sum of the scores is the total spam score of the message.

This chapter describes the SpamAssassin pattern-based and network-based tests: how they are written and scored, and how you can modify the score of a built-in test or write your own custom tests. This chapter also covers whitelist and blacklist rules, which can override SpamAssassin's usual determination of whether or not a message is spam.

The tests described in this chapter are all *static* tests—they don't change over time as SpamAssassin analyzes messages. Chapter 4 explains learning tests, which use information from messages seen in the past to improve decisions in the future.

The Anatomy of a Test

Most SpamAssassin tests consist of the same basic components:

- A test name, consisting of up to 22 uppercase letters, numbers, or underscores. Names that begin `T_` refer to rules in testing.
- A more verbose description of the test, which is used in the reports generated by SpamAssassin. Typically, descriptions are up to 50 characters long.
- An indication of where to look. Tests can be applied to the message headers only, the message body only, uniform resource identifiers (URIs) in the message body, or the complete message. When testing the message body, the body can be analyzed in its raw state, after MIME-decoding the text, or after MIME-decoding, stripping of HTML, and removal of all line breaks.

- A description of what to look for. Tests can specify a header to check for existence, a Perl regular expression pattern to match, a DNS-based blacklist to query, or a SpamAssassin function to evaluate.
- Optional test flags that control the conditions under which the test is applied or other exceptional features.
- A score or scores for the test. Tests can have a single score that is always used, or they can have separate scores for messages that test positive under each of four conditions:
    - When the Bayesian classifier and network tests are not in use
    - When the Bayesian classifier is not in use, but network tests are
    - When the Bayesian classifier is in use, but network tests are not
    - When the Bayesian classifier and network tests are both in use

Example 3-1 shows the complete definition of a test that matches when a message's *From* address begins with at least two numbers. This test is defined in the file */usr/share/spamassassin/20_head_tests.cf* (although its score appears in the *50_scores.cf* file).

Example 3-1. A test definition and score

```
header FROM_STARTS_WITH_NUMS    From =~ /^\d\d/
describe FROM_STARTS_WITH_NUMS  From: starts with nums

score FROM_STARTS_WITH_NUMS     0.390 1.574 1.044 0.579
```

How does this test work? The `header` directive defines it as a test that will be applied to the message headers and gives the test name (`FROM_STARTS_WITH_NUMS`) and the test itself, a match of the *From* header against the regular expression `/^\d\d/`. That regular expression denotes a string that begins with two digits.

For information about how to read and write regular expressions, see the Perl manual page *perlre*, or Jeffrey Friedl's book *Mastering Regular Expressions* (O'Reilly).

The `describe` directive provides a human-readable description of the test that SpamAssassin will insert in reports when the test matches. The `score` directive determines how many points SpamAssassin will add to the spam score of a message if the test matches. Higher scores mean that a message that matches the test is more likely to be spam. In this example, SpamAssassin will add 0.39 points to the spam score of a matching message if network and Bayesian tests are not in use, 1.574 points if network tests are in use but Bayesian tests are not, 1.044 points if Bayesian tests are in use but network tests are not, and 0.579 points if both network and Bayesian tests are in use.

The tests distributed with SpamAssassin are typically stored in files in */usr/share/spamassassin*. Tests are stored in a set of ruleset files based on the type of test being performed, and scores for all tests are stored together in one file. These tests are discussed in detail later in this chapter. Following are some other examples of test definitions from the distributed tests, along with their scores.

Testing for a *To*, *From*, or *Cc* header that mentions *friend@public.com* (this test is distributed disabled):

```
header FRIEND_PUBLIC       ALL =~ /^(?:to|cc|from):.*friend\@public\.com/im
describe FRIEND_PUBLIC     sent from or to friend@public.com
score FRIEND_PUBLIC        0
```

Testing for the existence of the *X-PMFLAGS* header:

```
header X_PMFLAGS_PRESENT        exists:X-PMFLAGS
describe X_PMFLAGS_PRESENT      Message has X-PMFLAGS header
score X_PMFLAGS_PRESENT         2.900 2.800 2.800 2.700
```

Testing for long lines of hexadecimal code in the message body:

```
body LARGE_HEX                  /[0-9a-fA-F]{70,}/
describe LARGE_HEX              Contains a large block of hexadecimal code
score LARGE_HEX                 0.633 1.595 1.193 1.160
```

Testing for a *Subject* header in all capital letters, by evaluating a SpamAssassin function:

```
header SUBJ_ALL_CAPS            eval:subject_is_all_caps()
describe SUBJ_ALL_CAPS          Subject is all capitals
score SUBJ_ALL_CAPS             0.550 0.567 0 0
```

Testing for a message that includes HTML to open a new window with JavaScript (disabled by default):

```
body HTML_WIN_OPEN              eval:html_test('window_open')
describe HTML_WIN_OPEN          Javascript to open a new window
score HTML_WIN_OPEN             0
```

Testing for an HTTP (Hypertext Transfer Protocol) URI anywhere in the message that uses a numeric IP address (e.g., *http://3502894884*):

```
uri NUMERIC_HTTP_ADDR           /^https?\:\/\/\d{7,}/is
describe NUMERIC_HTTP_ADDR      Uses a numeric IP address in URL
score NUMERIC_HTTP_ADDR         2.899 2.800 2.696 0.989
```

Modifying the Score of a Test

You may find some tests more indicative of spam than SpamAssassin does by default. If SpamAssassin already provides a test that you value but doesn't assign it a high enough score (higher scores are more indicative of spam), you can easily modify the score of the test. Similarly, if one of SpamAssassin's tests is giving you too many false positives, you can reduce its score or disable the test entirely by setting its score to 0. SpamAssassin will not attempt to run a test with a score of 0.

Modifying Scores Systemwide

Make systemwide score adjustments in the systemwide configuration file, typically */etc/mail/spamassassin/local.cf*. To modify the score of a test, you must first determine its test name, either by reading the ruleset files or by examining the spam report from a message. To get a spam report on a message that doesn't score high enough for SpamAssassin to generate a report, you can use `spamassassin --test-mode`, as described in Chapter 2.

To change the score of a test, simply add a new `score` directive to the configuration file, like this:

```
score HTML_WIN_OPEN 2
```

This will enable the HTML_WIN_OPEN test and add two points to the score of messages that test positive on this test.

You can use the same approach to modify the descriptions of tests by adding new `describe` directives. For example, the default description for the HOT_NASTY test is "Possible porn - Hot, Nasty, Wild, Young". To shorten that to "Possible porn", add this directive to the configuration file:

```
describe HOT_NASTY Possible porn
```

Modifying Scores on a Per-User Basis

Users can use the `score` directive in per-user preference files to change the scoring of a test for an individual user. To do so, a user edits the *.spamassassin/user_prefs* file in her home directory and adds `score` directives. This approach to customizing scores is the simplest, but it requires users to have accounts on the system and access to files in their accounts.

Storing Scores in an SQL Database

When users do not have accounts or shell access (e.g., on a system that is an IMAP or webmail server), per-user scores can be stored in an SQL database and `spamd` can be configured to look up scores in the database. To store scores in SQL, you must install the *DBI* Perl module and an appropriate driver module for your SQL database server. Common choices are *DBD-mysql* (for the MySQL server), *DBD-Pg* (for the PostgreSQL server), and *DBD-ODBC* (for connection to an ODBC-compliant server).*

* "ODBC" stands for Open Database Connectivity.

You should create a database and a user with privileges to access it. You must then create a table in the database to store the user scores. The SpamAssassin source code includes a schema for a MySQL table in the *sql* subdirectory, which is shown in Example 3-2. SpamAssassin 3.0 also includes a schema for a PostgreSQL table.

Example 3-2. A MySQL table for user scores

```
CREATE TABLE userpref (
  username varchar(100) NOT NULL,
  preference varchar(30) NOT NULL,
  value varchar(100) NOT NULL,
  prefid int(11) NOT NULL auto_increment,
  PRIMARY KEY (prefid),
  INDEX (username)
) TYPE=MyISAM;
```

You can use a different name for the table. The name given in Example 3-2 is the default, however, and using it will require the least amount of SpamAssassin configuration effort.

Each row in this table specifies the score for a single test for an individual user. SpamAssassin expects the columns to contain the following information:

`username`
: Gives the username or email address of the user (the latter is more useful in virtual hosting environments). The special username @GLOBAL can be used to define global values in SQL that will be applied to all users.

`preference`
: Gives the name of the test to modify the score of. The column can also be used with other directives (e.g., `required_hits`, `auto_report_threshold`, and the whitelisting and blacklisting directives described later in this chapter) but cannot define new rules or modify administrative settings.

`value`
: Gives the new score for the test or a new value for one of the other directives (e.g., number of hits required to call a message spam or an email address to add to the whitelist). SpamAssassin does not provide any tools for adding data to these tables.

The `prefid` column and the PRIMARY KEY and INDEX clauses are useful but not necessary. `prefid` defines a primary key for the table, and an index is built on the `username` column to speed up queries.

To configure SQL support for user scores, set the following configuration parameters in your systemwide configuration file (*local.cf*):

`user_scores_dsn` *`DSN`*

This directive defines the data source name (DSN) for the SQL database. It tells `spamd` how it will connect to the database server. A typical DSN, for the Perl *DBI* module, is written like this:

```
DBI:databasetype:databasename:hostname:port
```

For example, to use a MySQL database named *sascores* running on a database server on the SpamAssassin host, the DSN would read:

```
DBI:mysql:sascores:localhost:3306
```

If the server were running PostgreSQL, the DSN would read:

```
dbi:Pg:dbname=sascores;host=localhost;port=5432;
```

`user_scores_sql_username` *`username`*

This directive defines the username that will be used to connect to the database server. This user must have permission to issue SELECT queries against the table but need not be permitted to modify the data or database structure.

`user_scores_sql_password` *`password`*

This directive defines the password associated with the username that will be used to connect to the server.

`user_scores_sql_table` *`tablename`*

This directive defines the name of the table that contains user preferences. The default *`tablename`* is "userpref".

`user_scores_sql_custom_query` *`query`* *(SpamAssassin 3.0)*

This directive specifies the SQL query that SpamAssassin will use to look up user preferences. The query must be specified on a single (long) line in the configuration file. The default query is:

```
SELECT preference, value FROM _TABLE_
WHERE username = _USERNAME_ OR username = '@GLOBAL'
ORDER BY username ASC
```

This is read as "return the preference and value fields from the configured table (_TABLE_) for those rows with the specified username (_USERNAME_) or with the @GLOBAL username, in ascending lexicographic order." Because SpamAssassin will use the value of each matching preference it encounters in order, and because @GLOBAL sorts before all usernames, user-specific preferences will effectively override global preferences.

You can use this directive to construct your own custom queries. Custom queries must also return the preference and value columns (in that order). Queries may use the special symbols `_TABLE_` (replaced by the name of the table where user preferences are stored), `_USERNAME_` (replaced by the user's username), `_MAILBOX_` (replaced by the portion of the username before an at sign [`@`] or the whole username if there is no at sign), and `_DOMAIN_` (replaced by the portion of the username after an at sign or a null value if there is none). The manpage for *Mail::SpamAssassin::Conf* provides a few interesting examples of default queries. To support individual, domain, and global settings, add rows to the table with usernames of @~*domain* (which will sort after @GLOBAL but before real usernames) and use this query:

```
SELECT preference, value FROM _TABLE_
WHERE username = _USERNAME_ OR username = '@GLOBAL'
OR username = '@~'||_DOMAIN_
ORDER BY username ASC
```

If you prefer to have some global preferences that cannot be overridden by users and others that can, you can add rows to the table for the unchangeable preferences with username ~GLOBAL (which will sort after all usernames) and rows for the changeable preferences with username @GLOBAL and use this query:

```
SELECT preference, value FROM _TABLE_
WHERE username = _USERNAME_ OR username = '@GLOBAL'
OR username = '~GLOBAL'
ORDER BY username ASC
```

Finally, you'll need to start spamd with the --nouser-config command-line option and either the --sql-config or --setuid-with-sql option to enable SQL-based configuration (and disable the use of *~/.spamassassin/user_prefs* files, which cannot be used by spamd together with SQL). If spamd runs as a non-*root* user, or if your users don't have home directories, use --sql-config; if spamd runs as *root* and users have home directories, using --setuid-with-sql will enable spamd's usual practice of changing uid to the user running spamc so that it can access the user's autowhitelist files.

Storing Scores in an LDAP Database

Another way to store per-user preferences in SpamAssassin 3.0 is in an LDAP (Lightweight Directory Access Protocol) database. This approach may appeal particularly to sites that already store their user account configuration in LDAP. To store scores in LDAP, you must install the *Net::LDAP* and *URI* Perl modules.

LDAP objects (like those that represent users) and their attributes (such as username, password, email address, etc.) are defined by one or more LDAP *schemas*. To add SpamAssassin preferences to your users, extend the objectClass that represents a user to allow an additional, optional spamassassin attribute, which you should define like this:

```
# spamassassin
# see http://SpamAssassin.org/ .
attributetype ( 2.16.840.1.113730.3.1.217
        NAME 'spamassassin'
        DESC 'SpamAssassin user preferences settings'
        EQUALITY caseExactMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )
```

The attribute SYNTAX must be multivalued (as in the example, which specifies the DirectoryString syntax with object identifier (OID) 1.3.6.1.4.1.1466.115.121.1.15), because a user object will have multiple spamassassin attributes, one for each preference setting.

The attributes themselves should be stored in the database. A spamassassin LDAP attribute should be set to the name of a SpamAssassin configuration directive followed by the value for the directive, separated by a space. SpamAssassin 3.0 includes an example of what such user definitions might look like in LDIF (LDAP Interchange Format) format. The spamassassin attribute added to this user's LDAP entry is emphasized:

```
dn: cn=Curley Anderson,ou=MemberGroupB,o=stooges
ou: MemberGroupB
o: stooges
cn: Curley Anderson
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
mail: CAnderson@isp.com
givenname: Curley
sn: Anderson
uid: curley
initials: Joe
homePostalAddress: 14 Cherry Ln.$Plano TX 78888
postalAddress: 15 Fitzhugh Ave.
spamassassin: add_header all Foo LDAP read
l: Dallas
st: TX
postalcode: 76888
pager: 800-555-1319
homePhone: 800-555-1313
telephoneNumber: (800)555-1214
mobile: 800-555-1318
title: Developemnt Engineer
facsimileTelephoneNumber: 800-555-3318
userPassword: curleysecret
```

To configure LDAP support for user scores, set the following configuration parameters in your systemwide configuration file (*local.cf*):

user_scores_dsn *DSN*

Defines the data source name for the LDAP database. It tells spamd how it will connect to the LDAP server. LDAP DSNs are specified as URLs according to RFC 2255, like this:

```
ldap://host:port/basedn?attr?scope?filter
```

For example, to use the LDAP server on the SpamAssassin host to search for objects under the base DN of dc=example,dc=com and to return the spamassassin attributes for those in which the uid attribute matches the username that SpamAssassin is running for, the DSN would be:

```
ldap://localhost:389/dc=example,dc=com?spamassassin?sub?uid=__USERNAME__
```

`user_scores_ldap_username` *`bind_dn`*
: Provides the DN that SpamAssassin should use to bind to the LDAP server. This DN must have sufficient privileges to perform the query defined in the DSN.

`user_scores_ldap_password` *`password`*
: Provides the password that SpamAssassin should use to authenticate itself when binding to the LDAP server with the specified *`bind_dn`*.

Finally, you'll need to start `spamd` with the `--nouser-config` command-line option and either the `--ldap-config` or `--setuid-with-ldap` option to enable LDAP-based configuration (and disable the use of *~/.spamassassin/user_prefs* files, which cannot be used by `spamd` together with LDAP). If `spamd` runs as a non-*root* user, or if your users don't have home directories, use `--ldap-config`; if `spamd` runs as *root* and users have home directories, using `--setuid-with-ldap` will enable `spamd`'s usual practice of changing uid to the user running `spamc` so that it can access the user's autowhitelist files.

Writing Your Own Tests

When none of the existing tests does what you'd like, you can write a custom test of your own. Custom tests are just like the distributed tests, except that you install them in the systemwide configuration file or in a per-user preference file.

> Users can write their own tests in their per-user preference files, but for security reasons these tests will not be used when `spamd` is performing spam-checking, unless the `allow_user_rules` option is set to 1 in the systemwide configuration. However, setting this option is dangerous because `spamd` runs as *root* and a malicious or inexperienced user can construct a custom test that causes the system to hang or to invoke an arbitrary command as *nobody* or as `spamd`'s uid. Users who want their own tests on a system that uses `spamd` should reinvoke the `spamassassin` script on their incoming mail (probably in their *.procmailrc*). Chapter 2 illustrates this approach.

The first step in writing a custom test is to choose a symbolic test name and write a meaningful test description with the `describe` directive. For now, do not begin any of your names with a double underscore (__). Test names that begin with two underscores are not listed in test hit reports, nor are they added to the spam score on their own; such names are used for creating sets of subtests that should be applied in combination. SpamAssassin calls these combinations *meta tests*, and they are discussed later in this section.

Second, determine what part of the message you wish to test. Table 3-1 summarizes the directives used to test different portions of a message. Each is covered in greater detail in the following sections.

Table 3-1. Message portions and associated test directives

Message part	Directive	Possible tests
Headers	header TESTNAME	Match a regexp
		Don't match a regexp
		Exists
		Evaluate Perl code
		Check *Received* headers against DNSBL
Message subject and text of message body, decoding all textual MIME parts, with HTML tags and line breaks removed	body TESTNAME	Match a regexp
		Evaluate Perl code
Text of message body, decoding all textual MIME parts, with HTML tags and line breaks retained	rawbody TESTNAME	Match a regexp
		Evaluate Perl code
Undecoded message body including all MIME parts	full TESTNAME	Match a regexp
		Evaluate Perl code
URIs in the message body	uri TESTNAME	Match a regexp
URIs in the message body	uridnsbl TESTNAME	(SpamAssassin 3.0) Check for address in a DNS-based blacklist

Third, decide if your test requires any special test flags. Test flags are used to inform SpamAssassin that your test may apply only under certain conditions or may do something unusual. Use the `tflags` ***`TESTNAME flaglist`*** directive to indicate test flags. The ***`flaglist`*** is a space-separated list of flags. Table 3-2 lists the available flags in SpamAssassin and their effects.

Table 3-2. Test flags

Flag	Meaning
net	A network-based test that will not be run when SpamAssassin is directed to run local tests only
learn	A test that requires training before use (e.g., the Bayesian tests)
userconf	A test that requires user configuration before use (e.g., a test that expects the user to provide a list of addresses)
nice	A test that will be given a negative score
noautolearn	(Spamassassin 3.0) A test that will not be applied in the spam score when determining whether the message should be automatically learned as spam or non-spam

For example, the RCVD_IN_BL_SPAMCOP_NET test, which checks the message's *Received* headers against the DNS-based blacklist at *bl.spamcop.net* is defined in *20_dnsbl_tests.cf* like this:*

```
header   RCVD_IN_BL_SPAMCOP_NET eval:check_rbl_txt('spamcop', 'bl.spamcop.net.')
```

* The upcoming "Header Tests" section explains the details of how DNS-based blacklist-checking is performed.

```
describe RCVD_IN_BL_SPAMCOP_NET Received via a relay in bl.spamcop.net
tflags   RCVD_IN_BL_SPAMCOP_NET net
```

Finally, after adding or modifying a test, you should run `spamassassin --lint` to check your new rules for correct syntax. This command will attempt to parse all of the rules and configuration files in the ruleset directory and systemwide configuration directory. It exits quietly if no errors are found.

Versioning Your Rules

If you plan to create an extensive set of new rules, and especially if you plan to distribute them to other SpamAssassin users, you should use the `version_tag` configuration option to set a string that will denote your version of the rules. This string will appear in the *X-Spam-Status* header, after SpamAssassin's version number.

For example, set `version_tag` like this:

```
version_tag example.com
```

to produce the following in the header:

```
X-Spam-Status: No, hits=0.9 required=5.0 tests= FROM_NO_LOWER autolearn=no
version=3.0.0-example.com
```

If your rules rely on a particular version of SpamAssassin, include the `require_version` directive, followed by the required version number. When SpamAssassin sees this directive when parsing a file, it skips the rest of the file unless the version number is an exact match for the running version. For example, to ensure that custom rules you wrote for SpamAssassin 2.63 won't be used in SpamAssassin 3.0, add this line to the top of the file containing your rules:

```
require_version 2.63
```

Header Tests

Use the *header* directive to define a header test. Header tests can test for the existence of a header or check to see if a header matches (or fails to match) a regular expression.

To check for the existence of a header, use the following syntax:

```
header TESTNAME exists:headername
```

Regular expression tests can be applied to any single header in a message, both the *To* and *Cc* headers, all *Message-Id* headers, or all headers. Use the following form to match a header to a Perl regular expression:

```
header TESTNAME headername =~ /regexp/modifiers
```

Use this next syntax to test whether a header does not match a regular expression:

```
header TESTNAME headername !~ /regexp/modifiers
```

In these tests, the *headername* can be the name of a single header, or can be *ToCc* (to match in the *To* or *Cc* header), *MESSAGEID* (to match in any *Message-Id* header), or *ALL* (to match in any header). SpamAssassin 3.0 also supports *headername* `EnvelopeFrom` to match against the address supplied in the `SMTP MAIL FROM` command if the MTA provides this information to SpamAssassin.

A header that does not exist will not match any regular expression. To handle the possibility of a nonexistent header, you can add an optional `[if-unset: STRING]` after the regular expression and modifiers, and *STRING* will be tested against the regular expression if the header does not exist. For example, to look for a *Reply-To* header that either contains `@localhost` or is missing, you could use this rule:

```
header LOCAL_OR_NO_REPLY reply-to =~ /@localhost/ [if-unset: @localhost]
```

Many of the methods available in the *Mail::SpamAssassin::EvalTests* module test headers. This module is not documented, but you can learn about its methods by reading the rules distributed with SpamAssassin. For example, the `subject_is_all_caps( )` method matches when the *Subject* header contains all capital letters. This test is the basis of the SUBJ_ALL_CAPS rule distributed with SpamAssassin:

```
header SUBJ_ALL_CAPS      eval:subject_is_all_caps( )
```

Configurable header tests (SpamAssassin 3.0)

Some of the header tests in SpamAssassin 3.0 that use *Mail::SpamAssassin::EvalTests* methods have configurable parameters that control their operation. These parameters should be defined in sitewide or user configuration files.

The `check_for_from_dns( )` method performs a DNS lookup on the address in the message's *Reply-To* or *From* header to ensure that an MX record listing a host willing to receive mail for the message sender's host exists. Because DNS lookups can be slow, two configuration file options, `check_mx_attempts` and `check_mx_delay` are provided so you can adjust these lookups. Set `check_mx_attempts` to the number of lookup attempts you are willing to have SpamAssassin make (the default is 2). Set `check_mx_delay` to the number of seconds to wait between attempts in case the domain name server is temporarily down (the default is 5).

The `check_hashcash_value( )` and `check_hashcash_double_spend( )` methods implement Hashcash verification (*http://www.hashcash.org*). If a message includes an *X-Hashcash* header, SpamAssassin can quickly verify that the sender spent the required processing time to produce a valid header and reduces the message's spam score in proportion to how difficult it was for the sender to produce the header. To control SpamAssassin's use of Hashcash, define the following configuration variables:

`use_hashcash`
: If this variable is set to `1` (the default), Hashcash headers in messages will be checked. To disable Hashcash-checking, set this variable to `0`.

`hashcash_accept` *`address(es)`*

In order for SpamAssassin to perform a Hashcash check, it must know all of the valid addresses that could receive mail with Hashcash headers. Set this variable to provide those addresses.

You can use multiple `hashcash_accept` directives or multiple addresses in a single directive to list several addresses. You can also use an asterisk (*) as a wildcard for zero or more characters and the question mark (?) as a wildcard for zero or one character, much as you would to specify filename patterns in a shell. Finally, you can use `%u` to represent the current user's username in a sitewide configuration file. For example, a sitewide configuration file for users at *example.com* might include:

```
hashcash_accept %u@example.com %u@*.example.com
```

`hashcash_doublespend_path` *`/path/to/file`*

Set this variable to the path at which SpamAssassin will create and maintain a (Berkeley DB format) database of previously seen Hashcash headers to prevent a sender from reusing a header. The default file is *~/.spamassassin/hashcash_seen*. For a shared sitewide database, the user SpamAssassin runs as must have permission to write to this file and its directory.

`hashcash_doublespend_file_mode` *`mode`*

The file mode, in octal, for the Hashcash double-spend database. The default file mode is 0700. The file mode should include execute bits so that SpamAssassin can create directories, if necessary; i.e., use 0700 rather than 0600.

check_rbl( )

A set of methods that can be the basis for new tests are the `check_rbl( )`, `check_rbl_txt( )`, and `check_rbl_sub( )` methods. These methods extract IP addresses from a message's *Received* headers, discard those that are known to be reserved addresses or on trusted networks, and query a DNS-based blacklist for each address. If any of the addresses are listed in the blacklist, the test matches. Rules using these methods are written like other eval rules:

```
header A_NEW_BLACKLIST    eval:check_rbl('nasties','new.blacklist.zone')
```

Call `check_rbl( )` with two arguments. The first argument is the *zone ID*, a string that's used to identify the blacklist. It's primarily useful when you're querying a blacklist that's composed of many different lists, and you later want to evaluate the query result by which sublists the addresses were on (this topic is discussed later in this chapter).

If you append *-notfirsthop* to the name of the zone ID, the originating IP address will be excluded from RBL lookups unless it is the only IP address. This is useful when querying blacklists of dialup or DSL (Digital Subscriber Line) hosts that are expected to relay all their email through an ISP's mail server. If *new.blacklist.zone* was this kind of blacklist, you might have written the test like this:

```
header A_NEW_BLACKLIST    eval:check_rbl('nasties-not-firsthop','new.blacklist.zone')
```

Similarly, you can append *-firsttrusted* to check the IP address that appears in the *Received* header that was added by the most remote trusted server (IP addresses in *Received* headers added by more remote relays cannot be trusted). This is useful for querying a DNS-based whitelist to determine whether the server that first relayed the email to a trusted server appears on the whitelist. By appending *-untrusted*, you will check only the untrusted IP addresses (those more remote than the most remote trusted server). Here's a definition for a test of a DNS-based whitelist:

```
header A_NEW_WHITELIST    eval:check_rbl('friends-firsttrusted','new.whitelist.zone')
tflags A_NEW_WHITELIST    nice
```

(Remember, as Table 3-2 points out, when defining a test that will lower the spam score, you must set the `nice` test flag.)

The second argument is the DNS zone for the blacklist. SpamAssassin checks the blacklist by performing a DNS query for a hostname in this zone. SpamAssassin determines the hostname by reversing the IP address that it's trying to check (e.g., 128.0.10.0 becomes 0.10.0.128) and prepending it to the zone name (e.g., creating 0.10.0.128.new.blacklist.zone). It then issues a query for a DNS A record associated with that hostname. Typically, if an address is blacklisted, the DNS query will be successful—it will return an IP address (usually 127.0.0.1). If the address is not on the blacklist, the DNS query will fail (returning an NXDOMAIN response).

check_rbl_txt( )

Some blacklists are based on DNS TXT records instead of DNS A records. (Blacklist operators should indicate which kind of lookup is appropriate for their blacklist.) Use the `check_rbl_txt( )` method to perform lookups using a blacklist based on TXT records. `check_rbl_txt( )` accepts the same arguments as `check_rbl( )` and works analogously. SpamAssassin reverses the IP address that it's trying to check (e.g., 128.0.10.0 becomes 0.10.0.128) and prepends it to the zone name (e.g., creating 0.10.0.128.new.blacklist.zone). It then issues a query for a DNS TXT record associated with that hostname. If the address is blacklisted, the TXT query will return a string explaining why the address is blacklisted. If the address is not on the blacklist, the DNS query will fail (returning an NXDOMAIN response).

check_rbl_sub( )

Some DNSBLs are aggregations of many different blacklists. These DNSBLs typically return different IP addresses in response to a successful A lookup to indicate on which sublist(s) the blacklisted address appears (e.g., the query returns 127.0.0.1 for addresses on sublist 1, 127.0.0.2 for addresses on sublist 2, etc.).

Use the `check_rbl_sub( )` method to query a combined DNSBL and determine if the IP address is on a specific sublist. This method also takes two arguments: the first is a zone ID, and the second indicates which response is associated with the desired

Trusted and Untrusted Servers

Some mail servers are more trustworthy than others. In many organizations, email is received at an SMTP (Simple Mail Transfer Protocol) gateway on the Internet, checked for viruses, and then relayed through a firewall to an internal SMTP gateway that is responsible for delivering mail to individual machines on the internal network. In such a configuration, messages received by internal machines will have *Received* headers added by the internal SMTP gateway and the external SMTP gateway. The organization may also maintain (or contract with) off-site machines that serve as backup mail exchangers if the main SMTP gateway is unreachable. All of these machines are under the organization's control (or the control of a trusted provider), and the information in their headers can be trusted. *Received* headers added by other machines may be forged.

SpamAssassin doesn't check the IP addresses of trusted relays against DNS-based blacklists. By default, SpamAssassin works backward through the *Received* headers, beginning with the one added by the MTA on its own system (which is always trusted), and decides whether or not the addresses in each header are trusted. SpamAssassin treats *Received* lines that show messages being received from the local host, from a host on the same /16 subnet, from a host with a private IP address, or by a host with a private IP address as accurate and uses them to infer trusted relays.

When these simple inferences are not sufficient, you can manually define a set of trusted relays or networks using the `trusted_networks` configuration option, like this:

```
trusted_networks 10/8 127/8 209.58.173.10
```

This specifies that all hosts in the 10.*.*.* range, all hosts in the 127.*.*.* range, and the single host 209.58.173.10 are to be trusted. Multiple `trusted_networks` directives can be used.

SpamAssassin 3.0 adds the `internal_networks` configuration option. Set `internal_networks` to the list of relays or networks that you trust because you manage them (or they are within your organization or are mail exchangers for your organization). `trusted_networks` may include other hosts that you trust but that are not part of your mail organization. Separating these concepts allows SpamAssassin 3.0 to do a better job of detecting spam from dialup hosts being routed around their ISP's designated outgoing mail server, while still allowing messages from trusted sites to skip blacklist-testing.

sublist. For example, if the *new.blacklist.zone* blacklist is composed of sublists that return 127.0.0.1 and 127.0.0.2, you could check IP addresses against only the second sublist:

```
header A_NEW_BLACKLIST    eval:check_rbl('nasties','new.blacklist.zone')
header NEW_BLACKLIST_2    eval:check_rbl_sub('nasties','127.0.0.2')
```

Less commonly, composite lists may return a single A record whose IP address is to be interpreted as a bitmask of matching sublists. To check a sublist in this case, provide a bitmask (as a positive decimal number) as the second argument to `check_rbl_sub( )`.

Note that you must have a rule that uses `check_rbl( )` or `check_rbl_txt( )` to associate a zone ID string with the blacklist in order to check the result against a sublist.

Body Tests

The `body`, `rawbody`, and `full` directives define tests on the body of an email message. Two basic kinds of tests are provided. Message bodies can be tested against a regular expression pattern, and message bodies can be submitted to an eval test defined in *Mail::SpamAssassin::Evaltests*.

The `body` directive defines a test to be applied to the text of a message, as it would be likely to appear to a person reading the message in a text-based mail client. The *Subject* header is considered to be the first paragraph of the message body. All textual MIME components of the message are decoded, and HTML tags are removed. The message is reformatted into paragraphs (text separated by multiple newlines), and newlines within paragraphs are removed. The test is then applied to each message paragraph. Here's an example of a body test distributed with SpamAssassin that matches if the word "remove" appears in quotes in the body:

```
body REMOVE_IN_QUOTES           /\"remove\"/i
```

The `rawbody` directive defines a test to be applied to the text of a message, as it would be likely to appear to a person reading the message in an HTML-based mail client. The *Subject* header is *not* included. All textual MIME components of the message are decoded, and the message is split into lines based on the line breaks in the message. The test is then applied to each message line. Here's an example of a rawbody test distributed with SpamAssassin that's designed to find a JavaScript statement that's common in spam:

```
rawbody HIDE_WIN_STATUS         /<[^>]+onMouseOver=[^>]+window\.status=/I
```

Note that this test could not be written as a body test because this JavaScript appears inside an HTML tag.

The `full` directive defines a test to be applied to the full text of a message. All headers are included, along with all textual MIME components of the message body, but no decoding is performed. The message is split into lines based on the line breaks in the message, and the test is then applied to each header and message line. SpamAssassin does not distribute any full tests that match regular expressions; it reserves full for eval tests that must submit the raw message to external spam clearinghouses (which are discussed later in this chapter).

Body tests are powerful but slow. Be especially careful when defining regular expressions to test message bodies, as these expressions will be applied to large amounts of text. Consult Jeffrey Friedl's book *Mastering Regular Expressions* (O'Reilly) for important tips on optimizing regular expression processing.

URI Tests

The uri directive defines a test on all URIs that appear in an email message. SpamAssassin creates a list of *http*, *https*, *ftp*, *mailto*, *javascript*, and *file* URIs and transforms bare hostnames starting with *www* or *ftp* into appropriate URIs. The test is applied to each URI in the message.

URIs can be matched against a regular expression pattern. Here's an example of a distributed URI test that checks for a *mailto* URI with the string "remove" in the address portion:

```
uri MAILTO_TO_REMOVE           /^mailto:.*?remove/is
```

SpamAssassin 3.0 includes a plug-in called *Mail::SpamAssassin::Plugin::URIDNSBL*. When loaded, this plug-in enables the uridnsbl directive, which takes each URI in the message, extracts the name of the host in the URI, looks up its IP address in DNS, and then checks the IP address against a specified DNSBL. These tests catch spam that is relayed through innocent (or temporary) mail servers but that advertise web sites on spammer servers. Here's a portion of SpamAssassin 3.0's *25_rules.cf* file that defines a uridnsbl test called URIBL_SBLXBL:

```
loadplugin Mail::SpamAssassin::Plugin::URIDNSBL
...
uridnsbl  URIBL_SBLXBL    sbl-xbl.spamhaus.org.   TXT
header    URIBL_SBLXBL    eval:check_uridnsbl('URIBL_SBLXBL')
describe  URIBL_SBLXBL    Contains a URL listed in the SBL/XBL blocklist
```

Meta Tests

A meta test is a test that combines the results of several other tests using Boolean logic. For example, a meta test might be positive if either of two subtests are positive, or might specify that both subtests must be positive. A meta test can combine several tests using Boolean operators for and (&&), or (||), and not (!), along with parentheses to modify the precedence in the expression.

When using meta tests, you will often want some or all of the subtests to contribute only to the meta test and not to be separately scored. To achieve this effect, give the subtests names that begin with two underscores. This prevents SpamAssassin from scoring them separately. You can then assign a single score to the meta test. Because non-scoring subtests will never be listed in a SpamAssassin report, you need not include a describe directive for these tests.

Example 3-3 shows the CLICK_BELOW meta test in SpamAssassin.

Example 3-3. A meta test and its subtests

```
body CLICK_BELOW_CAPS      /CLICK\s.{0,30}(?:HERE|BELOW)/s
describe CLICK_BELOW_CAPS  Asks you to click below (in capital letters)

body __CLICK_BELOW         /click\s.{0,30}(?:here|below)/is
```

Example 3-3. A meta test and its subtests (continued)

```
meta CLICK_BELOW            (__CLICK_BELOW && !CLICK_BELOW_CAPS)
describe CLICK_BELOW        Asks you to click below
```

The CLICK_BELOW_CAPS test is standard body test that is positive if the words "CLICK BELOW" or "CLICK HERE" appear in the message in uppercase. Although it is a standard test that is used and scored on its own, SpamAssassin also uses it as a subtest in a meta test. The __CLICK_BELOW test is a nonscoring subtest that is positive if the same phrases appear in any combination of upper- and lowercase letters. The CLICK_BELOW meta test is positive when __CLICK_BELOW is positive and CLICK_BELOW_CAPS is *not* positive—that is, when the phrase appears in anything except all uppercase. Typically, a mixed or lowercase occurrence is assigned a lower score than the uppercase version.

In addition to using Boolean logic operators, it's also possible to use arithmetic operators (+, -, *, /) and comparisons (>, >=, <, <=, !=, =). When you combine tests with arithmetic operators, the values of subtests are 1 if they are positive and 0 if they are negative. One such meta test in SpamAssassin is MULTI_FORGED, which counts the number of positive tests for different kinds of *Received* header forgery and is positive when two or more forgeries appear in the same message. This test is shown in Example 3-4.

Example 3-4. The MULTI_FORGED meta test

```
meta MULTI_FORGED     ((FORGED_AOL_RCVD + FORGED_HOTMAIL_RCVD + FORGED_EUDORAMAIL_RCVD +
FORGED_YAHOO_RCVD + FORGED_JUNO_RCVD + FORGED_GW05_RCVD) > 1)
```

The Built-in Tests

SpamAssassin is distributed with over 700 test rules defined for English-language spam. SpamAssassin 2.63 includes another 2,900 rules for spam in other languages. (Language support in SpamAssassin 3.0 is currently available only for French and German, but language support is likely to increase as SpamAssassin gets into wider release.) Reading the rules distributed with SpamAssassin is an excellent way to learn to write your own rules.

SpamAssassin's rules are defined in a set of files typically installed in */usr/share/spamassassin*:

10_misc.cf
: The *10_misc.cf* file defines templates for the spam report that SpamAssassin attaches to spam messages, definitions of headers that SpamAssassin adds to messages, and default settings for the most common configuration options. This file is described in more detail later in this chapter.

10_plugins.cf (SpamAssassin 3.0)
: This file provides a convenient place to load SpamAssassin plug-in modules with the `loadplugin` directive. Plug-ins extend SpamAssassin's features.

20_fake_helo_tests.cf
: This file defines a set of rules used to test for forged HELO hostnames. This file is also described in more detail later in this chapter.

20_body_tests.cf
: This file defines most tests against message bodies, spam clearinghouses, message languages, and message locales. It's described in more detail later.

20_dnsbl_tests.cf
: This file defines tests against many different DNS blacklists, using the `check_rbl( )`, `check_rbl_sub( )`, and `check_rbl_txt( )` eval tests described earlier in this chapter. These blacklists include NJABL (*http://www.dnsbl.njabl.org/*), SORBS (*http://www.dnsbl.sorbs.net/*), OPM (*http://opm.blitzed.org/*), Spamhaus (*http://sbl.spamhaus.org*), DSBL (*http://dsbl.org*), Spamcop (*http://bl.spamcop.net*), MAPS (*http://www.mail-abuse.org*), and several others.

20_ratware.cf and 20_anti_ratware.cf
: The *20_ratware.cf* file contains tests that look for tell-tale signs of specialized mail programs known to be used by spammers (*ratware* or *spamware*). Most of them are tests of message headers. The *20_anti_ratware.cf* file is designed to contain tests that look for signs of non-spam mail programs that might be mistaken for spamware, but it doesn't contain any active tests as of SpamAssassin 3.0.

20_head_tests.cf
: This file contains most of the tests that SpamAssassin performs against message headers. This includes tests for blacklisted and whitelisted addresses in the *From* and *To* headers (discussed in greater detail in Chapter 4).

20_porn.cf (all SpamAssassin versions) and 20_drugs.cf (SpamAssassin 3.0)
: These files contain body tests that look for common indicators of pornographic spam and online pharmacy spam, respectively.

20_phrases.cf
: This file contains body tests that look for common phrases that appear in spam. Most of them are either instructions for how you can be removed from the mailing list or claims that the message conforms to a bill that putatively regulates unsolicited email.

20_uri_tests.cf
: This file contains most of the tests that SpamAssassin performs against URIs that appear in messages.

20_compensate.cf
: Tests in this file are intended to compensate for common false positives in header tests and are "nice" tests (with negative spam scores).

20_html_tests.cf
: This file contains body tests that target messages that contain HTML markup. Certain types of markup are very commonly seen in spam, and several of these tests make for interesting reading.

20_meta_tests.cf
: This file contains meta tests. Meta tests are tests that combine other tests, and are described earlier in this chapter.

23_bayes.cf
: This file contains tests that act on the results of the Bayesian classifier. The Bayesian system and these tests are described in greater detail in Chapter 5.

25_head_tests_es.cf, 25_body_tests_es.cf, 25_head_tests_pl.cf, 25_body_tests_pl.cf (SpamAssassin 2.6x)
: These files contain header and body tests for Spanish (es) and Polish (pl) messages.

25_uribl.cf (SpamAssassin 3.0)
: This file loads the URIDNSBL plug-in and defines URI tests against DNS blacklists.

30_text_*.cf (de,es,fr,it,pl,sk)
: These files don't define any new tests but provide translations of test descriptions and report templates into different languages, such as German (de), Spanish (es), French (fr), Italian (it), Polish (pl), and Slovak (sk). SpamAssassin 3.0 includes only German and French tests at the time of this writing.

50_scores.cf
: This file defines the scores associated with all of the tests defined in the other files. The scores are separated into a single file because they are generated by an algorithm that applies each test to a large corpus of spam and non-spam messages and adjusts the scores to minimize false positives and false negatives.

60_whitelist.cf
: The rules in this file set up default whitelists for several large well-known addresses and companies, such as *Amazon.com*.

Because these files are overwritten whenever SpamAssassin is upgraded, they should not be changed. All local rules or changes to the scoring of distributed rules should be performed in the systemwide configuration file (or in per-user preference files) rather than in these files. Reading these files, however, provides the most information about how SpamAssassin rules are designed.

The following sections describe some of the more important rule files in greater detail.

10_misc.cf

The *10_misc.cf* file defines special rules that are not spam tests. These include templates for the spam report that SpamAssassin attaches to spam messages, definitions of headers that SpamAssassin adds to messages, and default settings for the most common configuration options (such as those described in Chapter 2).

Templates are defined with the `report`, `unsafe_report`, and `spamtrap` directives, and the corresponding utility directives `clear_report_template`, `clear_unsafe_report_template`, and `clear_spamtrap_template`. Use the `report` template to design the report that SpamAssassin attaches to spam messages. Use the `unsafe_report` template to design the report that SpamAssassin attaches to messages that contain potentially executable code. Use the `spamtrap` template to design the message that SpamAssassin sends back to senders who email a spam trap address that calls the `spamassassin` script with the `--report` and `--warning-from` options (spam-reporting is discussed in Chapter 2).

Each time it encounters a template directive, SpamAssassin appends new text to the template. Accordingly, to ensure that you're starting with a clean slate when you define a new template, you must first clear the template and then add your desired text. Here's how the spam report might be defined in SpamAssassin:

```
clear_report_template
report Spam detection software, running on the system "_HOSTNAME_", has
report identified this email as possible spam. The original message
report is attached to this so you can view it (if it isn't spam) or block
report similar future email.  If you have any questions, see
report _CONTACTADDRESS_ for details.
report
report Content preview:  _PREVIEW_
report
report Content analysis details:   (_HITS_ points, _REQD_ required)
report
report " pts rule name              description"
report ---- ---------------------- --------------------------------------
report _SUMMARY_
```

`_HOSTNAME_`, `_CONTACTADDRESS_`, `_PREVIEW_`, `_HITS_`, `_REQD_`, and `_SUMMARY_` are variables that are replaced by their values when the template is generated for each message. The complete list of variables, which appears in the *Mail::SpamAssassin::Conf* manpage, is given in Table 3-3.

Table 3-3. Variables for use in report and header templates

Variable	Value
Variables that depend on the message	
YESNOCAPS	"YES" if message is spam; "NO" if message is not spam.
YESNO	"YES" if message is spam; "NO" if message is not spam.

Table 3-3. Variables for use in report and header templates (continued)

Variable	Value
HITS	Spam score for message.
BAYES	Bayesian classifier score.
AUTOLEARN	"spam" if message was auto-learned as spam by the Bayesian classifier; "ham" if auto-learned as non-spam; "NO" if the message was not auto-learned.
AWL	Autowhitelist score modifier.
DATE	Date and time of SpamAssassin scan in RFC 2822 format.
STARS	A string containing one asterisk for each point of spam score (up to 50).
STARS(*character*)	A string containing one of *character* for each point of spam score (up to 50).
RELAYSTRUSTED	List of relays found in the message and deemed to be trusted. The list includes the IP address, reverse DNS lookup, and HELO address for each relay.
RELAYSUNTRUSTED	List of relay IP addresses found in the message and deemed to be untrusted.
TESTS, _TESTSSCORES_	Comma-separated list of tests matched, or tests matched and their associated scores.
TESTS(*character*), _TESTS–SCORES(*character*)_	As in _TESTS_, _TESTSSCORES_ but separated by *character* instead of comma.
LANGUAGES	List of languages that SpamAssassin thinks a message is written in.
PREVIEW	Preview of message content.
SUMMARY	Multiline list of tests matched and their scores and descriptions.
REPORT	One line list of tests matched.
RBL	Results of positive DNSBL queries.
DCCB, _DCCR_	Checking host and results of DCC check of message.
PYZOR	Results of Pyzor check of message.
Variables that don't depend on the message	
REQD	SpamAssassin's threshold score for calling a message spam.
VERSION, _SUBVERSION_	Version and subversion of SpamAssassin.
HOSTNAME	Hostname of SpamAssassin host.
CONTACTADDRESS	The value of the `report_contact` directive (typically, the email address of the postmaster).

The variables in Table 3-3 can also be added to customized message headers for messages processed by SpamAssassin by using the `add_header` directive, which takes the following form:

```
add_header messagetype headername string
```

The *messagetype* can be `spam`, `ham` (non-spam), or `all` and determines which kind of messages will have the header added. The new header will be named `X-Spam-`*`headername`*, and *`string`*, which should be enclosed in double quotes, will be the value of the header. For example, the following directive, which appears in the distributed

10_misc.cf file, adds an *X-Spam-Status* header to all messages—spam or not—that shows whether or not each message is spam, the spam score, the spam threshold score, the tests that were matched, whether the message is being automatically learned (see Chapter 5), and the version of SpamAssassin:

```
add_header all Status "_YESNO_, hits=_HITS_ required=_REQD_ tests=_TESTS_ autolearn=_
AUTOLEARN_ version=_VERSION_"
```

If you want to change or remove a default header, you can use the `remove_header` directive:

```
remove_header messagetype headername
```

You can remove all headers with the `clear_headers` directive.

20_fake_helo_tests.cf

This file defines a set of rules that use the eval test `check_for_rdns_helo_mismatch( )`. This test takes two arguments: a regular expression pattern to match against the reverse DNS lookup of the connecting client's IP address, and a regular expression pattern to match against the hostname provided by the client during in the SMTP HELO command. Spammers often use mail programs that forge the HELO hostname, and these tests look for such forgeries when the clients have hostnames that match those of major commercial ISPs. Here's an example of a test from this file:

```
header FAKE_HELO_AOL  eval:check_for_rdns_helo_mismatch("aol\.com","aol\.com")
describe FAKE_HELO_AOL  Host HELO did not match rDNS: aol.com
```

This test matches if the client connects from an IP address that reverse-resolves to an *aol.com* hostname but claims in the HELO to have a hostname that does not match "aol.com". These tests are applied to all of the *Received* headers from untrusted relays.

You can use this eval test to reject messages that claim, in their HELO, to be from your own host. If your hostname is *myhost.example.com*, and you know that your IP address reverse-resolves to the same hostname, you could add a rule like this (to the systemwide configuration file):

```
header FAKE_MY_HELO eval:check_for_rdns_helo_mismatch("(?!myhost\.example\.com).
{18}$","myhost\.example\.com")
describe FAKE_MY_HELO Host HELO faked my hostname
score FAKE_MY_HELO 5.0
```

The regular expression `(?!myhost\.example\.com).{18}$` matches any hostname containing at least 18 characters that does not end in *myhost.example.com*, which should match the reverse DNS lookup of any untrusted relay host other than your own. If any such host claims in their HELO to be *myhost.example.com*, it is forging your hostname.

20_body_tests.cf

This file contains most of the tests that SpamAssassin performs against message bodies. In addition to tests for regular expressions in the body, this file defines tests against spam clearinghouses and tests of message language and locale.

A spam clearinghouse is a server that maintains a database of checksums of messages reported as spam and allows clients to test a message against the checksum database. SpamAssassin supports three spam clearinghouses: Vipul's Razor (*http://razor.sf.net/*), Pyzor (*http://pyzor.sf.net*), and the Distributed Checksum Clearinghouse, or DCC (*http://rhyolite.com/anti-spam/dcc/*). Special client software must be installed on the system in order for SpamAssassin to use these tests. The `spamassassin --report` command can be used to report confirmed spam to these clearinghouses as well.

In SpamAssassin 3.0, the `pyzor_options` configuration directive can be set to a string of additional options to be passed to the Pyzor client on the command line when SpamAssassin invokes it. Similarly, the `dcc_options` directive can be set to provide additional options to the DCC client.

Whitelists and Blacklists

Although SpamAssassin generally does a good job of avoiding false positives, you may find that some mail that you want to receive contains enough spamlike characteristics that SpamAssassin regularly tags them as spam. You may want to be sure that SpamAssassin will never mistake email from an important user, client, vendor, or other sender for spam. You may even have users who don't like spam-filtering. SpamAssassin allows you to set up systemwide or user-specific lists of senders whose mail should not be considered spam, and (systemwide) lists of users who don't want their email filtered. Such lists are called *whitelists*.

On the other hand, you may regularly receive unwanted mail from a particular sender that doesn't get tagged reliably by SpamAssassin. You may know ahead of time that you don't want to receive mail from certain organizations or senders. SpamAssassin also allows you to set up system-wide or user-specific lists of senders whose mail should be tagged as spam. Such lists are called *blacklists*.

This chapter discusses how to set up whitelists and blacklists. It begins by examining the SpamAssassin directives for systemwide whitelisting and blacklisting, and then explores two different ways to manage user-specific lists. A related feature, autowhitelists, is covered in Chapter 4.

Systemwide Whitelists

SpamAssassin whitelists reduce the spam scores of messages when the sender or recipient appears on the whitelist. Whitelists are most commonly used to ensure that messages from important senders are not marked as spam, but they can also be used to change the spam threshold for recipients or enable recipients to effectively opt out of spam-tagging.

Whitelisting senders

Use the `whitelist_from` directive to whitelist a sender's address. The sender's address is the address that appears in the *Resent-From* header, if that header exists, or in any of the headers: *From*, *Envelope-Sender*, *Resent-Sender*, or *X-Envelope-From*. If a sender's address matches a `whitelist_from` address, the spam score of the message is reduced by 100 points, which makes it nearly impossible for the message to be tagged as spam.

For example, if you receive important messages from *boss@mybigclient.com*, you can ensure that they won't be tagged as spam by using this line in the systemwide configuration file:

```
whitelist_from boss@mybigclient.com
```

You can use multiple `whitelist_from` directives or multiple addresses in a single directive to whitelist several addresses. You can also use an asterisk (*) as a wildcard for zero or more characters and a question mark (?) as a wildcard for zero or one character, much as you would to specify filename patterns in a shell. For example, you could whitelist all mail from *mybigclient.com* and from all hosts in the *example.com* domain with these lines:

```
whitelist_from *@mybigclient.com
whitelist_from example.com *.example.com
```

A whitelist entry can be removed with the `unwhitelist_from` directive. Because SpamAssassin is distributed with several default whitelist entries (in the *60_whitelist.cf* file), you may find that you want to remove some of them. The `unwhitelist_from` directive is also useful in per-user configuration files, to remove one of the systemwide whitelist entries. To remove a whitelist entry, the address in the `unwhitelist_from` directive must exactly match the one given to `whitelist_from`.

Whitelisting senders by relay

Sometimes whitelisting by the sender's address alone isn't sufficient. For example, the sender's address might be one that's easily guessed or likely to be spoofed by spammers. For example, a spammer might try to ensure that you read his message by forging the sender's address to *hostmaster@internic.net* or *billing@amazon.com*.

SpamAssassin offers more control over whitelisted senders with the `whitelist_from_rcvd` directive. This directive associates a sender's email address with the hostname or domain name of the last trusted relay. SpamAssassin uses DNS to do a reverse-lookup of the IP address of the last trusted relay; the reverse-lookup yields one or more hostnames associated with the IP address. Here's how you would whitelist *boss@mybigclient.com* only if the last trusted relay reverse-resolves to a hostname in the *mybigclient.com* domain:

```
whitelist_from_rcvd boss@mybigclient.com mybigclient.com
```

Messages that match a `whitelist_from_rcvd` directive have their spam scores lowered by 100.

> In order for SpamAssassin to distinguish trusted and untrusted relays, you may need to set the `trusted_networks` option, which was described earlier. If your mail topology is relatively simple—you or your ISP control all of the IP addresses in the class B network that includes your mail server's public IP address—SpamAssassin can usually make a reasonable guess.

SpamAssassin is distributed with several, default, relay-based whitelist entries in the *60_whitelist.cf* file. These entries are defined with the `def_whitelist_from_rcvd` directive, which works just like `whitelist_from_rcvd` but lowers the spam score by only 15 when a message matches.

As you might expect, whitelist entries based on relays can be removed with the `unwhitelist_from_rcvd` *`address`* directive. The *`address`* must exactly match the address defined in a `whitelist_from_rcvd` or `def_whitelist_from_rcvd` directive. If the `whitelist_from_rcvd` directive uses wildcards, the `unwhitelist_from_rcvd` directive must specify those same wildcards.

Whitelisting recipients

SpamAssassin provides three levels of whitelisting for message recipients. Whitelisting a recipient lowers the spam score on all messages addressed to the recipient. Use recipient-whitelisting to prevent any spam-checking from being performed on behalf of a recipient. You can also use recipient-whitelisting as a crude mechanism for increasing the spam threshold—lowering the false positive rate at the cost of more false negatives—for a recipient.

A recipient's address may appear in several headers. If *Resent-To* and/or *Resent-Cc* headers are present, the address is checked against only those headers. Otherwise, the address may be matched in the last three *Received* headers or the headers *To*, *Apparently-To*, *Delivered-To*, *Envelope-Recipients*, *Apparently-Resent-To*, *X-Envelope-To*, *Envelope-To*, *X-Delivered-To*, *X-Original-To*, *X-Rcpt-To*, *X-Real-To*, or *Cc*.

The three levels of recipient-whitelisting are configured with the directives `whitelist_to` (lower spam score by 6), `more_spam_to` (lower spam score by 20), and `all_spam_to` (lower spam score by 100). For example, to ensure that no messages to *root* or *postmaster* are tagged as spam, you could use the following lines:

```
all_spam_to root@*
all_spam_to postmaster@*
```

No `unwhitelist_to` directive is provided because whitelisting by recipient is really useful only in systemwide configuration. Individual users can just change their `required_hits` setting in their *.spamassassin/user_prefs* file instead.

Systemwide Blacklists

SpamAssassin has only two blacklist directives (and two directives to unblacklist addresses). You can blacklist sender addresses or recipient addresses.

The `blacklist_from` directive is used to specify a sender's address to blacklist. The sender's address is the address that appears in the *Resent-From* header, if that header exists, or in any of the headers *From*, *Envelope-Sender*, *Resent-Sender*, or *X-Envelope-From*. If the sender's address matches a `blacklist_from` address, the spam score of the message is increased by 100 points, which makes it almost certain that the message will be tagged as spam.

For example, a spammer might send messages from *support@microsofts.com* in the hope that you'll think it's an important message from an operating system vendor. If you never expect to receive legitimate messages from *support@microsofts.com*, you can ensure that any message from that address will be tagged as spam by using this line in the systemwide configuration file:

```
blacklist_from support@microsofts.com
```

You can use multiple `blacklist_from` directives or multiple addresses in a single directive to blacklist several addresses. You can also use an asterisk (*) as a wildcard for zero or more characters and a question mark (?) as a wildcard for zero or one character, much as you would to specify filename patterns in a shell. For example, you could blacklist all mail from *public.com* and from all hosts in the *example.com* domain with these lines:

```
blacklist_from *@public.com
blacklist_from example.com *.example.com
```

You can remove a blacklist entry with the `unblacklist_from` directive. To remove a blacklist entry, the address in the `unblacklist_from` directive must exactly match the one given to `blacklist_from`.

The `blacklist_to` directive performs blacklisting based on recipient address. As with whitelisting, a recipient's address may appear in several headers. If *Resent-To* and/or *Resent-Cc* headers are present, the address is checked only against those headers. Otherwise, the address may be matched in the last three *Received* headers or the headers *To*, *Apparently-To*, *Delivered-To*, *Envelope-Recipients*, *Apparently-Resent-To*, *X-Envelope-To*, *Envelope-To*, *X-Delivered-To*, *X-Original-To*, *X-Rcpt-To*, *X-Real-To*, or *Cc*. If a recipient address matches a `blacklist_to` entry, the spam score of the message is increased by 10 points.

Blacklisting by recipient is most useful when spammers use software that sends mail with recognizably forged *To* headers (specifying the real recipient in the SMTP transaction, of course). For example, it used to be popular to send spam with a *To* header of *friend@public.com*. Although SpamAssassin already includes a special test for this address in headers, you could also use the `blacklist_to` configuration directive to increase the spam score for such messages by 10 points:

```
blacklist_to friend@public.com
```

No `unblacklist_to` directive is provided. Simply don't blacklist a recipient who should continue to receive mail.

> It's possible, but silly, for the same address to be both blacklisted and whitelisted. In this case, both lists are applied and, if the blacklist adds 100 to the spam score and the whitelist subtracts 100, cancel one another out.

Per-User Whitelists and Blacklists

Email from a given address may be welcomed by one user and shunned by another. Although systemwide whitelists and blacklists are useful antispam tools, in many cases, each user will want her own individual whitelist and blacklist entries.

SpamAssassin provides two mechanisms for per-user whitelists and blacklists. The first mechanism is the simplest: add the appropriate configuration directives to the per-user configuration file for the user's account (typically *~/.spamassassin/user_prefs*). The disadvantage of this approach is that it requires users to have accounts and access to their home directories.The second mechanism is to configure `spamd` to look up per-user test scores and whitelists and blacklist entries in an SQL or LDAP database, as described earlier in this chapter.

If users want to remove systemwide whitelist or blacklist entries, they can use the `unwhitelist_from` or `unblacklist_from` directives described earlier in this chapter.

Whitelists and Blacklists Without SpamAssassin

If SpamAssassin is not run on a systemwide basis on all messages, users can also implement whitelists and blacklists by carefully organizing the filters they use to run SpamAssassin on their messages.

For example, on a Unix system that uses procmail for message delivery, a user could whitelist *boss@mybigclient.com* and blacklist *support@microsofts.com* with *procmail* recipes before the recipe that runs SpamAssassin. The user's *.procmailrc* might contain:

```
:0
* ^From:.*boss@mybigclient.com
$DEFAULT

:0
* ^From:.*support@microsofts.com
/dev/null

:0fw
* <300 000
|/usr/bin/spamassassin
```

CHAPTER 4

SpamAssassin as a Learning System

SpamAssassin provides many rules that have proven useful in distinguishing spam from non-spam messages, and these rules are updated at each new release. But SpamAssassin provides more than just generic rules; it has the capability of learning about *your* email environment and adapting its detection behavior to maximize its accuracy in that environment.

SpamAssassin includes two adaptive systems that can be used in concert: autowhitelisting and Bayesian filtering. This chapter discusses the principles, configuration, and operation of both systems.

Autowhitelisting

SpamAssassin's autowhitelisting algorithm learns each sender's history of sending spam or non-spam messages and modifies the spam score of their subsequent mailings on the basis of this history. The primary goal of autowhitelisting is to reduce false positives—to make it less likely that a non-spam message will be tagged as spam—by assuming that people who send you non-spam messages will not begin to spam you. It can also reduce false negatives if a spammer consistently sends email from the same email address, but this happens infrequently enough that autowhitelisting rarely has a significant effect on false negatives.

Principles

When autowhitelisting is enabled, SpamAssassin maintains a database keyed on message senders' email addresses and the IP addresses of their nearest untrusted relay (if any). Each time a message from a given sender is received, the message's spam score is added to the sender's total score in the database, and a count of the number of messages received from that sender is updated.

The average sender score—the total score divided by the number of messages received—is used to modify the spam score of new messages from that sender.

Specifically, the difference between the average score and the new message's score is multiplied by a configurable factor, and the result is added to the new message's spam score. The effect is that when the new message has a higher spam score than average, its spam score is adjusted downward; when the new message has a lower spam score than average, its spam score is adjusted upward.

As you might expect from this explanation, the autowhitelist tests are the last ones performed by SpamAssassin. All other tests must be run first in order to have the most accurate spam score for a message before comparing it to the sender's historical average. In addition, the sender's historical average is updated with the spam score of a new message *before* the autowhitelist modifier is applied.

Configuration

The most important decisions to make in autowhitelisting are how much weight SpamAssassin should put on a sender's history of sending spam or non-spam messages and how much weight it should put on the spam score of the message it is checking.

Use the `auto_whitelist_factor` directive to set the multiplier that is applied to the difference between a message's spam score and the sender's historical average score. It can range from 0 to 1. The default factor is 0.5, which causes the final spam score to be halfway between the message's spam score and the sender's average score.

To put more weight on the historical average, increase the `auto_whitelist_factor`. When the `auto_whitelist_factor` is set to 1, the historical average alone will be the new message's spam score (recall, however, that the score before autowhitelisting is performed is fed back into the system and becomes part of the new historical average).

To put less weight on the historical average, decrease the `auto_whitelist_factor`. When the `auto_whitelist_factor` is set to 0, the historical average is ignored, and the current message's spam score will not be modified based on the sender's past messages.

Table 4-1 illustrates the impact of several different settings for `auto_whitelist_factor`. Each row of the table represents a new message from the same sender. Table columns show the spam score of each message before applying an autowhitelist modifier, the sender's historical average score, and the spam score after applying an autowhistelist modifier. In this example, the sender sends several non-spam messages and then sends a message that looks like spam to SpamAssassin (a false positive). As you can see, with autowhitelisting using factors of 0.5, 0.75, or 1, the message will not reach the usual spam threshold of 5 because of the sender's history of non-spam messages. Without autowhitelisting (i.e., with an factor of 0), the message receives a score of 6.

Table 4-1. The impact of auto_whitelist_factor (AWF)

Message number	Message score (before autowhitelist)	Sender average score	Score after autowhitelist with given AWF			
			0	.5	.75	1
1	2	(none)	2	2	2	2
2	1	2	1	1.5	1.75	2
3	1	1.5	1	1.25	1.375	1.5
4	0	1.33	0	0.67	1.00	1.33
5	2	1.0	2	1.5	1.25	1.0
6	6	1.2	6	3.6	2.4	1.2

SpamAssassin stores its autowhitelist data in database files. SpamAssassin lets Perl's *AnyDBM* module choose which database format will be used, based on which system libraries are available. In SpamAssassin 3.0, you can control this choice by setting the `auto_whitelist_db_modules` option to a space-separated list of Perl database modules to be tried in order; the first module that loads successfully will be used. For example, the default module order is specified like this:

```
auto_whitelist_db_modules DB_File GDBM_File NDBM_File SDBM_File
```

How you configure autowhitelisting also depends on whether you want each user to have his own whitelist database, or whether you want to use one database in common across all users.

Configuring per-user autowhitelists

By default, SpamAssassin maintains a separate autowhitelist for each user on the system. SpamAssassin stores the autowhitelist database for a user in the *auto-whitelist* file in the *.spamassassin* subdirectory of each user's home directory. SpamAssassin uses one of several database formats for this file, depending on what database libraries are available on the system; the Berkeley DB format is chosen when it's available.

SpamAssassin 3.0 can also store autowhitelists in an SQL database, which is useful when users don't have accounts on the mail server. To store addresses in SQL, you must install the *DBI* Perl module and an appropriate driver module for your SQL server. Common choices are *DBD-mysql* (for the MySQL server), *DBD-Pg* (for the PostgreSQL server), and *DBD-ODBC* (for connection to an ODBC-compliant server).

You should create a database and a user with privileges to access it. You must then create a table in the database to store the user autowhitelist. The SpamAssassin source code includes schemas for MySQL and PostgreSQL tables in the *sql* subdirectory. Here is the MySQL schema:

```
CREATE TABLE awl (
  username varchar(100) NOT NULL default '',
  email varchar(200) NOT NULL default '',
  ip varchar(10) NOT NULL default '',
  count int(11) default '0',
  totscore float default '0',
  PRIMARY KEY  (username,email,ip)
) TYPE=MyISAM;
```

Each row in this table specifies an autowhitelist entry for a single sender for an individual SpamAssassin user. SpamAssassin uses the columns to store the following information:

`username`
: Stores the username or email address of the user (the latter is more useful in virtual hosting environments).

`email`
: Stores the email address of a sender whose messages' spam scores are being tracked.

`ip`
: Stores the IP address of the sender.

`count`
: Stores the total number of messages received from the sender.

`totscore`
: Stores the total spam score of messages received from the sender.

To configure SQL support for autowhitelists, set the following configuration parameters in your systemwide configuration file (*local.cf*):

`auto_whitelist_factory Mail::SpamAssassin::SQLBasedAddrList`
: Configures SpamAssassin to use SQL-based autowhitelists instead of file-based autowhitelists.

`user_awl_dsn` *`DSN`*
: Defines the data source name for the SQL database, telling `spamd` how it will connect to the database server. A typical DSN for the Perl *DBI* module is written like this:

```
DBI:databasetype:databasename:hostname:port
```

For example, to use a MySQL database named *saawl* running on a database server on the SpamAssassin host, the DSN would read:

```
DBI:mysql:saawl:localhost:3306
```

If the server were running PostgreSQL, the DSN would read:

```
dbi:Pg:dbname=saawl;host=localhost;port=5432;
```

`user_awl_sql_username` *`username`*
: Defines the username that will be used to connect to the database server. This user must have permission to modify the data in the table (including inserting and deleting rows).

`user_awl_sql_password` *`password`*
: Defines the password associated with the username that will be used to connect to the server.

`user_awl_sql_table` *`tablename`*
: Defines the name of the table that contains autowhitelist data. The default *`tablename`* is `awl`.

Configuring a system-wide autowhitelist

It is often desirable to maintain a single autowhitelist for all users of a system. When users don't have home directories, such an approach is not just desirable but may be necessary if autowhitelisting is to be used. You can configure a systemwide autowhitelist by setting the `auto_whitelist_path` directive to the full path of the autowhitelist database file. Set `auto_whitelist_path` in the systemwide configuration file. For example, to set up a systemwide autowhitelist in the file */etc/mail/spamassassin/auto-whitelist*, use the following directive:

```
auto_whitelist_path /etc/mail/spamassassin/auto-whitelist
```

If SpamAssassin encounters this directive, it checks to be sure the database file exists. If the file does not exist, SpamAssassin attempts to create it. You may not want to give SpamAssassin write access to the directory you specify. One way around that is to create the file as *root*, change its ownership to the SpamAssassin user, and set the mode to allow read/write access, all before you add the `auto_whitelist_path` to your configuration file.

However you create it, the systemwide autowhitelist database file should be readable and writable by the user running SpamAssassin. Depending on your configuration, SpamAssassin may be running as *root*, as one of several users on the system, or as a default unprivileged user such as `nobody`. If you let SpamAssassin create the systemwide autowhitelist database file, you can use the `auto_whitelist_file_mode` directive to specify the file's mode. It defaults to 0700 but may need to be set to 0770 or 0777 depending on your configuration, when multiple users must access the file.

Using a systemwide autowhitelist with mode 0777 (or 0770 and an inappropriate group) will enable a curious local user to learn the email addresses of message senders and their average spam scores *or to modify those scores*. A malicious user could modify the database to give legitimate senders a false history of spamming. In general, file modes other than 0700 should be avoided.

Using an Autowhitelist

Once the autowhitelisting system is configured, you must instruct SpamAssassin to use it. In SpamAssassin 2.63, if you invoke SpamAssassin with the `spamassassin` script, add the `--auto-whitelist` option to direct the script to consult your autowhitelist. If you invoke SpamAssassin with the `spamc` client, you should start `spamd` (the daemon) with the `--auto-whitelist` option to direct it to consult user autowhitelists.

SpamAssassin 3.0 contains no `--auto-whitelist` command-line options. Instead, autowhitelists are always used when the `use_auto_whitelist` configuration option is set in a user's (or a systemwide) configuration file.

Using Autowhitelists in Perl

If you've written a Perl application that uses *Mail::SpamAssassin* to checks messages, you can take advantage of autowhitelists, but it requires a little additional setup. You must create an *address list factory*, an object that generates objects to store autowhitelisted addresses, and you must associate the address list factory with your *Mail::SpamAssassin* object. Here is sample code that does this:

```
#!/usr/bin/perl

use Mail::SpamAssassin;

my $spamtest = Mail::SpamAssassin->new( );
my $awl = Mail::SpamAssassin::DBBasedAddrList->new;
$spamtest->set_persistent_address_list_factory($awl);
# Now go on to use $spamtest as usual.
```

Mail::SpamAssassin also provides methods for adding and removing addresses from the autowhitelist. See the manpage for more information.

You can use the `spamassassin` script to manipulate the contents of your autowhitelist. The following command-line options to `spamassassin` operate on your autowhitelist:

`--add-addr-to-whitelist=`*`emailaddress`*
: Adds *`emailaddress`* to the autowhitelist with an initial score of –100. SpamAssassin will forget any past history associated with the address.

`--add-addr-to-blacklist=`*`emailaddress`*
: Adds *`emailaddress`* to the autowhitelist with an initial score of 100. SpamAssassin will forget any past history associated with the address.

`--remove-addr-from-whitelist=`*`emailaddress`*
: Removes *`emailaddress`* from the autowhitelist. SpamAssassin will forget any past history associated with the address.

`--add-to-whitelist`
: When you pipe an email message to `spamassassin --add-to-whitelist`, SpamAssassin adds all email addresses found in the *To*, *From*, *Cc*, *Reply-To*, *Sender*, *Errors-To*, and *Mail-Followup-To* headers or in the body of the message to the autowhitelist with initial scores of –100. SpamAssassin will forget any past history associated with these addresses.

`--add-to-blacklist`
: When you pipe an email message to `spamassassin --add-to-blacklist`, SpamAssassin adds all email addresses found in the *To*, *From*, *Cc*, *Reply-To*, *Sender*, *Errors-To*, and *Mail-Followup-To* headers or in the body of the message to the autowhitelist with initial scores of 100. SpamAssassin will forget any past history associated with these addresses. Because this behavior will probably result in the blacklisting of your own email address, this option is usually useless.

`--remove-from-whitelist`
: When you pipe an email message to `spamassassin --remove-from-whitelist`, SpamAssassin removes all email addresses found in the *To*, *From*, *Cc*, *Reply-To*, *Sender*, *Errors-To*, and *Mail-Followup-To* headers or in the body of the message from the autowhitelist and forgets any past history associated with these addresses.

> Be careful with `--add-to-blacklist`. A malicious spammer could send you HTML email with friendly addresses (including your own) embedded in invisible `mailto:` tags. Piping this message to `spamassassin --add-to-blacklist` causes SpamAssassin to add all of *those* addresses to the autowhitelist as likely spammers! Using `--add-addr-to-blacklist` with individual email addresses is safer.

Bayesian Filtering

SpamAssassin's Bayesian classifier learns to distinguish the features that characterize spam from those that characterize non-spam in the messages that you receive. Properly trained, the Bayesian classifier can reduce both false positives and false negatives.

Principles

Bayesian filtering is based on Bayes' Theorem, a statement of probability theory propounded by the Reverend Thomas Bayes in 1763. Bayes' Theorem is important in many fields where classifying data is essential, including computer vision, psychophysics, and diagnostic decision-making in health care. SpamAssassin's implementation is mostly based on the work of Paul Graham (archived at *http://www.paulgraham.com*) and Gary Robinson (*http://www.garyrobinson.net*).

Conceptually, Bayes' Theorem states that the probability of some event (such as a message being spam) given a test result (such as matching a spam-checking rule) depends on the baseline probability of the event before the test result is known and on the discriminating power of the test. A corollary is that the discriminating power of a test can be measured by comparing the probability of the event given a known test result to the baseline probability before the result is known. The more the test result can increase (or decrease) the probability from baseline, the stronger the test.

Actually, SpamAssassin's "Bayesian" system doesn't really compute the baseline probability or frequency of spam versus non-spam messages—which some have argued means it's not strictly Bayesian at all. Instead it assumes values that seem reasonable and useful.

In the context of spam-checking, a Bayesian approach amounts to developing potential rules and asking how much each rule, if matched, should change the system's perception of the likelihood that a message is spam. Very strong rules come in two forms. Some are patterns that only occur in spam (and never in non-spam), thus yielding a high probability that a message that matches one of the patterns is spam. Others are patterns that only occur in non-spam (and never in spam), thus yielding a low probability that a message that matches the pattern is spam. Weaker rules—patterns found in both spam and non-spam messages but with different frequencies—result in less extreme probabilities.

To use Bayesian filtering successfully, you must have a corpus of messages that you have decided are definitely spam, a corpus of messages that you have decided are definitely non-spam, and an algorithm for analyzing the two sets of messages to develop rules and test their strength. SpamAssassin provides the algorithm and a script that you can use to identify messages as spam or non-spam in order to train the filter. It also provides a mechanism for training itself with messages that are very likely to be spam or non-spam.

The results of the SpamAssassin learning process are a set of databases. One database contains *tokens* (strings of 3–15 characters) that have been seen, how often each has been seen in spam and non-spam messages, and the date and time that each token last proved useful in classifying a message. During learning, tokens are derived from both the message headers (with several commonly misleading headers ignored) and message body. Tokens that haven't been useful in a long time may be removed from the database to increase efficiency. Another database keeps track of which messages have been learned, so SpamAssassin doesn't waste time relearning old messages.

During spam-checking, a message to be checked is split into tokens. SpamAssassin then looks up each token in the token database. Up to 150 of the most diagnostic tokens in the message are identified, and their associated predictive values are combined using one of two mathematical functions to yield a final prediction of the probability that the message is spam. This predicted probability is matched by special SpamAssassin rules that associate probability ranges with spam score modifiers.

Configuration

SpamAssassin's Bayesian classifier is controlled by more than a dozen configuration directives, though only a few are regularly modified by system administrators. These are the most useful:

`use_bayes`
: This directive controls whether the Bayesian classifier is used at all. It defaults to 1 (use Bayesian filtering). By setting it to 0, Bayesian filtering is disabled completely.

`bayes_auto_learn`, `bayes_auto_learn_threshold_nonspam`, `bayes_auto_learn_threshold_spam`
: These directives configure the automatic learning system, which automatically feeds messages with very high or very low spam scores to the Bayesian classifier. The `bayes_auto_learn` directive enables (1) or disables (0) this feature; it is enabled by default. The `threshold` directives determine which messages will be automatically learned as spam or non-spam. Messages with spam scores lower than `bayes_auto_learn_threshold_nonspam` are learned as non-spam; this value defaults to 0.1. Messages with spam scores higher than `bayes_auto_learn_threshold_spam` are learned as spam; this value defaults to 12 and cannot be set lower than 6. The spam score used for making this determination does not include modifiers for the Bayesian system itself, for the autowhitelist, or for user-configured whitelists or blacklists.

`bayes_ignore_header` *headername*
: This directive tells the Bayesian classifier to ignore the given header when learning or classifying messages. It is most often used when another spam-tagging system adds headers before SpamAssassin receives the message, in order to prevent the classifier from learning the other spam tag instead of the features of the actual message.

`bayes_ignore_from` *address (SpamAssassin 3.0)*
: This directive prevents Bayesian classification and learning from being performed on messages sent from *address* and is a form of whitelisting. It's most useful when you want to receive messages from a few senders and the messages may include tokens that would otherwise suggest spam.

  You can use multiple `bayes_ignore_from` directives or multiple addresses in a single directive to whitelist several addresses. You can also use as asterisk (*) as a wildcard for zero or more characters and a question mark (?) as a wildcard for zero or one character, much as you would to specify filename patterns in a shell.

`bayes_ignore_to` *address (SpamAssassin 3.0)*
: This directive prevents Bayesian classification and learning from being performed on messages sent to *address*, and is a form of whitelisting recipients. It's useful in sitewide Bayesian filtering to prevent any learning from being performed from messages sent to *postmaster*, for example, who is likely to receive forwarded spam, non-spam messages discussing spam, etc. Specify addresses as you would to the `bayes_ignore_from` directive discussed previously.

`bayes_learn_during_report`
: When this directive is enabled (1), messages that are reported to clearinghouses as spam with the `spamassassin --report` command are also learned as spam by the Bayesian classifier. This saves you an extra learning step. Set the directive to 0 to disable this feature. It is enabled by default.

`bayes_path` *and* `bayes_file_mode`
: By default, SpamAssassin maintains separate Bayesian databases for each user on the system. The databases for a user are stored in the *.spamassassin* subdirectory of the user's home directory and their names begin *bayes_*, such as *bayes_seen* and *bayes_toks*. These files are kept in one of several possible database formats (Berkeley DB format is generally preferred when it's available to SpamAssassin).

  Separate databases for each user are ideal for Bayesian learning because different users may receive different kinds of spam and non-spam messages. However, it is often necessary to maintain a single Bayesian database for all users of a system, either to save on disk space or because users don't have home directories. You can configure a systemwide Bayesian database set by setting the `bayes_path` directive to the full path of the Bayesian database file prefix. For example, to set up systemwide Bayesian databases in the files */etc/mail/spamassassin/bayes_**, use the following directive:

```
bayes_path /etc/mail/spamassassin/bayes
```

  By default, the Bayesian databases are created with mode 0700. The `bayes_file_mode` directive can be used to set a different file mode (e.g., 0770) if you need to share the databases among a group. This might be necessary if SpamAssassin can be invoked with the privileges of different users. Care should be taken with this directive, as a malicious user with access to the Bayesian databases can cause legitimate email to be mistagged as spam.

The following directives influence the internal workings of the Bayesian classifier. For the most part, they can be left to the default settings.

`bayes_min_ham_num` *and* `bayes_min_spam_num`
: These directives set the minimum number of ham (non-spam) and spam messages that must be learned by SpamAssassin before it will use the predictions of the Bayesian classifier to score new messages. They default to 200 each; until

200 ham and 200 spam messages have been learned, the SpamAssassin rules that rely on the Bayesian classifier will not be applied to email.

`bayes_use_hapaxes`
: *Hapaxes* are tokens that have been seen only once during learning so far. Accordingly, SpamAssassin's concept of whether a hapax is associated with spam or ham is based on limited data and may not be reliable. On the other hand, SpamAssassin can learn hundreds or thousands of hapaxes, and using hapaxes seems to provide better accuracy, so this setting defaults to 1 (enabled).

`bayes_use_chi2_combining`
: This directive controls which of the two mathematical functions are used to combine token probabilities into an overall message probability. When enabled (1), the approach is based on the distribution of the chi-squared statistic; when disabled (0), a so-called "naïve Bayesian" function combines the probabilities using the assumption that errors in classification from each token are independent of one another. SpamAssassin's maintainers have found the chi-squared method more useful, and it is the default.

`bayes_auto_expire` *and* `bayes_expiry_max_db_size`
: When `bayes_auto_expire` is enabled (1), SpamAssassin will automatically attempt to remove old tokens during learning when the token database exceeds `bayes_expiry_max_db_size` tokens. This is the default. When disabled (0), token expiration must be performed manually. Automatic expiration occurs no more than once every 12 hours.

`bayes_learn_to_journal` *and* `bayes_journal_max_size`
: When `bayes_learn_to_journal` is enabled (1), SpamAssassin will store newly learned data in a journal file, rather than directly into the Bayesian databases. The journal file will be synchronized into the databases at least daily, or when the journal exceeds `bayes_journal_max_size` bytes (102,400 by default). Using journaling reduces disk contention for the databases, which must be exclusively locked while being updated, but results in a delay between the time a message is learned and the time the learned tokens can be used to classify further messages. Journaling might be particularly useful if the journal could be kept in a different location than the databases (e.g., on a RAM disk), but this directive is not supported as of SpamAssassin 3.0. `bayes_learn_to_journal` is disabled by default.

Training

There are two main strategies for training a Bayesian classifier: train everything and train-on-error. In the train everything strategy, you train the classifier with every message that you receive. This strategy is highly responsive to changes in spam patterns but may change too quickly in response to unrelated variability in messages. In addition, it is resource intensive to scan every message. In the train-on-error strategy, you train the classifier only with messages that it has previously classified incorrectly (i.e., false positives and false negatives). This strategy is resource efficient but may not train the classifier as quickly when spam patterns change.

Based on experiments conducted by Greg Louis (and described at *http://www.bgl.nu/bogofilter/*), the train everything strategy appears to be more efficient for initial training. Once a suitable number of messages have been learned, however, switching to a train-on-error approach saves resources, because many fewer messages must be trained. Louis suggests that switching to train-on-error after 10,000 spam and 10,000 non-spam message have been learned may be reasonable. You can train SpamAssassin's Bayesian classifier with either strategy.

The sa-learn script is your primary interface for training the Bayesian classifier. The first step in using Bayesian filtering is collecting a corpus of messages you've received that you have verified are spam and a corpus that you've verified are non-spam. The easiest and best way to do so is to simply start saving spam you receive to one folder and any non-spam messages that you would ordinarily delete to another. The two collections of messages can either be in *maildir* format (in which each file contains a single message) or *mbox* format (in which a single file contains multiple messages).

It's important that the messages be from the same time period; if you train SpamAssassin with a set of spam messages from 2003 and a set of non-spam messages from 2004, it will quickly learn that an effective way to detect spam is to look for messages in 2003! Similarly, forwarded spam, or messages discussing spam in your corpus ("Hey, look at this spam I just got; it's really strange. Here it is...") can result in the classifier learning artificial rules that will degrade its accuracy with normal messages.

Next, run sa-learn on each corpus, using either the --spam or --ham command-line options to specify what each corpus represents. Example 4-1 shows the process for a set of *mbox* files—a file of saved spam, a file of saved (non-spam) messages related to a project, and the user's mail spool. The project files and mail spool files together form a corpus of known good messages. This example assumes that each user maintains her own Bayesian databases, so sa-learn is run by each user on her own messages.

Example 4-1. Learning from a set of mbox files

```
$ ls -F Mail
spam    myproject
$ sa-learn --mbox --spam Mail/spam
$ sa-learn --mbox --ham mail/myproject
$ sa-learn --mbox --ham /var/spool/mail/$LOGNAME
```

Example 4-2 shows the process for a set of *maildirs*, again assuming that each user has his own Bayesian databases. The commands in the example are those that would be executed by each individual user. Providing a directory as an argument to sa-learn causes it to learn from every file in that directory. The example also illustrates the use of the --no-rebuild option to defer rebuilding of the databases until the --rebuild option is used. When performing learning on a large set of small files (the very essence of a *maildir*), deferring the expensive database-rebuilding step is more efficient than rebuilding after each file.

Example 4-2. Learning from a set of maildirs

```
$ ls -F mail
INBOX/    spam/    myproject/
$ sa-learn --no-rebuild --spam mail/spam
$ sa-learn --no-rebuild --ham mail/INBOX
$ sa-learn --no-rebuild --ham mail/myproject
$ sa-learn --rebuild
```

If you're the sort who likes to see the progress of the training (or who worries when you run a command that takes longer than a few seconds to finish), you can add the --showdots option to cause sa-learn to print a period for each message it processes.

You can also call sa-learn on an individual file containing a mail message, or you can pipe a mail message to sa-learn's standard input. Finally, you can put the names of mailboxes, files, or directories into a file and run sa-learn with the --folders=*filename* option, and it will read the file and directory names from the *filename* file and learn from each.

The Bayesian classifier is most effective when trained on large collections of both spam and non-spam messages. In particular, training using many spam messages and fewer non-spam messages is likely to produce an ineffective filter. Aim for a couple thousand messages of each type, collected prospectively from your personally received mail.

If you mistakenly train the Bayesian classifier that a message is spam, simply direct sa-learn to relearn it as ham; if you mistakenly learn a message as ham, you can direct sa-learn to relearn it as spam. This process is also how you later train the classifier on errors. You can also cause SpamAssassin to forget a message entirely by running sa-learn --forget on the message.

sa-learn also accepts the same --configpath */path/to/ruleset/directory*, --prefs-path */path/to/user_prefs*, and --siteconfigpath */path/to/sitewide/directory* directives that the spamassassin script does. They are described in Chapter 2.

Daily Use

When you first enable the Bayesian classifier in SpamAssassin, you will initially notice little change in the way messages are checked for spam. Once you've trained the classifier with enough messages, however, your spam scores for messages will begin to change substantially in two ways:

What's Being Learned?

Once your Bayesian classifier has been trained and is contributing to spam-checking, you might be curious to find out which tokens are actually being used. The `sa-learn --dump` *`type`* command displays that information. *`type`* can be one of these choices:

- `data` will cause `sa-learn` to display all of the tokens it has learned, with their associated spam probabilities, number of occurrences in spam and ham messages, and last time used.
- `magic` will cause `sa-learn` to display "magic" tokens. Although they're stored in the database, these tokens don't represent parts of email messages. They include such information as the number of spam and ham messages in the databases, the last time a token was used, etc.
- `all` will cause `sa-learn` to display tokens of both types.

Here are the first and last five lines of `sa-learn --dump data | sort -n` as executed on one system:

```
0.000    0    110 1072880922  discussion
0.000    0    112 1071162080  HMBOX-Line:2002
0.000    0    112 1072907632  modify
0.000    0    113 1072915324  H*u:Windows
0.000    0    115 1072900545  Sender
...
1.000  310      0 1071162080  N:HEADER_NBITS
1.000  316      0 1072026198  8-bit
1.000  323      0 1071162080  HEADER_8BITS
1.000  328      0 1072026198  N:N-bit
1.000  394      0 1072910571  Forged
```

The first five lines show tokens that have only exclusively appeared in non-spam messages. The last five show tokens that have exclusively appeared in spam messages. Tokens starting with H were found in headers; some headers are abbreviated with special codes starting with an asterisk (*)—so `H*u:` means the *User-Agent* header. Tokens starting with N: indicate that Ns that appear in the token should match any sequence of digits.

You can restrict which tokens are shown by `sa-learn --dump` by adding the `--regexp` *`regexp`* command-line option and providing a regular expression pattern *`regexp`*. Only tokens that match *`regexp`* will be displayed. This option is useful when you want to see the spam probability associated with specific tokens.

- Messages will show that they are hitting SpamAssassin rules with names like BAYES_44 or BAYES_80. These rules, which can be found in the *23_bayes.cf* file, are triggered when the Bayesian classifier assigns a given probability of spam to a message. For example, the BAYES_44 rule is matched when a message has a probability of spam between 0.44 and 0.4999; the BAYES_80 rule is triggered when a message has a probability of spam between 0.80 and 0.90. Rules that match on probabilities less than 0.5 lower spam scores, and those that match on probabilities greater than 0.5 raise spam scores.

- Most of the non-Bayesian rules assign different scores when the classifier is trained and in use than when it is not. In many cases, non-Bayesian rules produce less extreme scores, which reflects the supposition that the Bayesian classifier should be better than static rules at distinguishing spam from non-spam.

Ongoing training

Ongoing training is essential to maintaining the performance of a Bayesian filter. As in initial training, you must continue to provide examples of both spam and non-spam messages.

As you receive messages, check each message classified as spam to be sure that it is really spam and not a false positive. If the message's spam score is higher than the threshold for automatic learning, the message should have already been fed back into the classifier to train it. You can determine if this has happened by looking at the `autolearn=` section of the *X-Spam-Status* header added by SpamAssassin. If the message's spam score wasn't high enough for automatic learning, submit it to `sa-learn --spam` yourself. If you come across a false positive, submit it to `sa-learn --ham` instead.

Similarly, you can submit your non-spam messages to `sa-learn --ham` if their spam scores are too high for the automatic learning threshold for ham. Any spam SpamAssassin misses should definitely be submitted to `sa-learn --spam`.

You can make the ongoing training process more convenient using one of two common ways. If you read your email with an email client that allows you to bind commands to keys, you could define keystrokes to invoke `sa-learn --ham` or `sa-learn --spam` on the current message. Another approach is to save all spam messages into a single mail folder and all non-spam messages that you plan to delete into a second folder, and then run `sa-learn` on each folder (and possibly on your inbox if you keep many undeleted messages there) at the end of your mail-reading session. Users or system administrators can set up `cron` jobs to automate this process.

Expiration and importing

Expiration and importing are two other functions of `sa-learn` that you will use infrequently. Expiration removes old tokens from the database, and importing updates the database if a new SpamAssassin release changes database formats.

As discussed earlier in this chapter, when `bayes_auto_expire` is enabled (the default), SpamAssassin's Bayesian classifier regularly reviews its database of tokens to determine if any should be expired. Expiration is always skipped when fewer than 100,000 tokens are in the database. The automatic expiration process runs no more than once every 12 hours and only when the number of tokens exceeds `bayes_expiry_max_db_size`.

If you do not use `bayes_auto_expire`, or if you want to expire tokens manually, you can force an expiration attempt by running `sa-learn --force-expire`. Doing so may not actually expire any tokens; for example, when fewer than 100,000 tokens or all tokens have been recently used, no tokens will be expired.

The `sa-learn --import` command is used to update the Bayesian databases from their format in an older version of SpamAssassin to the current format. The release notes for new versions of SpamAssassin should tell you when running `sa-learn --import` is necessary. In many cases, SpamAssassin will perform importation when it automatically learns a new message, so this command may not be necessary.

The import process can be both CPU and disk intensive, especially with a large database of tokens. It is best run during off-hours or times of low system load.

Storing Bayesian Data in SQL

SpamAssassin 3.0 can optionally store per-user Bayesian data in an SQL database, which is useful when users don't have accounts on the mail server. To store Bayesian data in SQL, you must install the *DBI* Perl module and an appropriate driver module for your SQL server. Common choices are *DBD-mysql* (for the MySQL server), *DBD-Pg* (for the PostgreSQL server), and *DBD-ODBC* (for connection to an ODBC-compliant server).

You should create a database and a user with privileges to access it. You must then create a set of tables in the database to store the Bayesian data. The SpamAssassin source code includes schemas for MySQL, PostgreSQL, and SQLite tables in the *sql* subdirectory. Here is the MySQL schema:

```
CREATE TABLE bayes_expire (
  username varchar(200) NOT NULL default '',
  runtime int(11) NOT NULL default '0',
  KEY bayes_expire_idx1 (username)
) TYPE=MyISAM;

CREATE TABLE bayes_global_vars (
  variable varchar(30) NOT NULL default '',
  value varchar(200) NOT NULL default '',
  PRIMARY KEY  (variable)
) TYPE=MyISAM;

INSERT INTO bayes_global_vars VALUES ('VERSION','2');

CREATE TABLE bayes_seen (
  username varchar(200) NOT NULL default '',
  msgid varchar(200) binary NOT NULL default '',
    flag char(1) NOT NULL default '',
    PRIMARY KEY  (username,msgid),
    KEY bayes_seen_idx1 (username,flag)
  ) TYPE=MyISAM;

  CREATE TABLE bayes_token (
    username varchar(200) NOT NULL default '',
    token varchar(200) binary NOT NULL default '',
    spam_count int(11) NOT NULL default '0',
    ham_count int(11) NOT NULL default '0',
    atime int(11) NOT NULL default '0',
    PRIMARY KEY  (username,token)
  ) TYPE=MyISAM;

  CREATE TABLE bayes_vars (
    username varchar(200) NOT NULL default '',
    spam_count int(11) NOT NULL default '0',
    ham_count int(11) NOT NULL default '0',
    last_expire int(11) NOT NULL default '0',
    last_atime_delta int(11) NOT NULL default '0',
    last_expire_reduce int(11) NOT NULL default '0',
    PRIMARY KEY  (username)
  ) TYPE=MyISAM;
```

For each user, these tables maintain information about token expiration (bayes_expire), messages seen (bayes_seen), tokens seen (bayes_token), and per-user configuration variables (bayes_vars). A table for global configuration variables (bayes_global_vars) is also available. The names of rows in these tables are similar to the corresponding SpamAssassin configuration variables and indicate the data they store.

To configure SQL support for Bayesian data, set the following configuration parameters in your systemwide configuration file (*local.cf*):

bayes_store_module Mail::SpamAssassin::BayesStore::SQL
: Configures SpamAssassin to use SQL-based storage for Bayesian data instead of file-based (DBM) storage.

bayes_sql_dsn *DSN*
: Defines the data source name for the SQL database. See the earlier definition of bayes_awl_dsn for examples of how to define a DSN.

bayes_dsn_sql_username *username*
: Defines the username that will be used to connect to the database server. This user must have permission to modify the data in the table (including inserting and deleting rows).

bayes_dsn_sql_password *password*
: Defines the password associated with the username that will be used to connect to the server.

SpamAssassin will now store Bayesian data learned from messages (either automatically or via `sa-learn`) in the SQL database and will look up tokens in this database when checking messages for a user.

SpamAssassin provides one additional configuration variable for SQL storage of Bayesian data:

`bayes_sql_override_username` *`someusername`*
: When this directive is set, the SQL query for Bayesian data will use *`someusername`* in place of the current user's name when adding new message data or retrieving data for message-checking. Generally, this directive should only be used in per-user configuration files so that most users have their own personal Bayesian data. In principle, you could also use it in the site-wide configuration file to create a sitewide Bayesian database, and then use it in per-user configuration files to exclude certain users from the sitewide data.

A Sitewide Bayesian Classifier

Bayesian filtering is most effective when each user maintains his own set of token databases trained from his own email. By learning about the peculiar characteristics of spam and non-spam messages received by an individual user, the Bayesian classifier becomes an effective test for future messages to that user. A pharmacist might receive a lot of legitimate email about sildenafil citrate, and having all of these messages tagged as spam (or worse) could be a serious problem.

Many sites, however, prefer to have a single set of databases for all users at the site, either to save disk space or because users do not have home directories and setting up SpamAssassin 3.0's SQL storage is infeasible. Setting up a sitewide Bayesian classifier is possible with SpamAssassin. Perform the following steps:

1. Set `bayes_path` and `bayes_file_mode` in the systemwide configuration file. Be sure the directory specified in `bayes_path` is readable, writable, and searchable by the user that SpamAssassin will be running as, so that it can create the proper files. The `bayes_file_mode` should be as strict as possible, typically 0700, which is the default setting. It's a good idea to set it explicitly, rather than rely on the default.
2. Provide a mechanism for users or administrators to submit messages for training. This step is the most difficult part of a sitewide Bayesian classifier. Because the database files will be owned by the user that SpamAssassin runs as, even local users typically will not be able to run `sa-learn` with the proper permissions to update the databases.

One solution for enabling users to submit spam messages for training is to ask users to bounce any spam they receive to a central mailbox that can be processed by a privileged script. For example, set up an email alias of *spamtrap* on the SpamAssassin system that pipes incoming messages to a script like that shown in Example 4-3. As an extra benefit, you can publicize the *spamtrap* address on public web pages or in

Usenet postings and actually use it as a spam trap—spammers who harvest the address and send spam to it will find their spam fed into your learning and reporting systems.

Example 4-3. A sitewide script for learning spam

```
#!/bin/sh
#
# This script accepts an email message on its standard input
# and feeds it to SpamAssassin's learning and/or reporting systems
# It is meant to be run as root or as the user who owns the
# SpamAssassin Bayesian databases


PATH=/bin:/usr/bin:/sbin:/usr/sbin

# Three choices:
# 1. Uncomment the following line to use --report if
# you have bayes_learn_during_report enabled.
spamassassin --report

# 2. Uncomment the following line to use sa-learn and
# spamassassin --report when you don't have
# bayes_learn_during_report enabled
# sa-learn --spam | spamassassin --report

# 3. Uncomment the following line to use sa-learn
# alone.
#sa-learn --spam
```

If you ask users to use a centralized *spamtrap* address, it is crucial that they *bounce* or *redirect* their messages, rather than *forward* their messages. A forwarded message's headers will show the message as being sent by the forwarding user, which is not what you want the Bayesian classifier to learn! Most mail clients provide a function for redirecting a message to a new address so that it still appears to be coming from the original sender. If your mail clients add extra headers when they do this, these headers are good candidates for `bayes_ignore_header`. You have to test to determine which, if any, headers your mail clients add and to be sure SpamAssassin is ignoring them.

A similar solution for non-spam messages is much more difficult—for social, rather than technical, reasons. Users may well be reluctant to forward their legitimate email to any central address. Unfortunately, without a good corpus of non-spam messages, the Bayesian filter will not perform well. One possible approach is to raise the `bayes_auto_learn_threshold_nonspam` slightly (e.g., to 0.5 or 1.0) so that much legitimate email will be auto-learned.

CHAPTER 5

Integrating SpamAssassin with sendmail

sendmail has long been the most widely used mail transport agent in the world. It was routing mail before the Internet existed as such and continues to form the backbone of many of the largest mail servers on the Net today. This chapter explains how to integrate SpamAssassin into a sendmail-based mail server to perform spam-checking for local recipients or to create a spam-checking mail gateway.

sendmail is a complex piece of software and can have several security implications for systems on which it runs. You should always run the most up-to-date version of sendmail and keep track of new bug reports and security advisories. This chapter assumes that you are running the latest release of sendmail—Version 8.12—and does not cover how to securely install, configure, or operate sendmail itself. For that information, see the sendmail documentation and the book *sendmail* by Bryan Costales and Eric Allman (O'Reilly).

Spam-Checking at Delivery

The easiest way to add SpamAssassin to a sendmail system is to configure sendmail to use procmail as its local delivery agent, and to add a procmail recipe for spam-tagging to */etc/procmailrc*. The advantages of this approach are

- It's very easy to set up.
- You can run `spamd`, and the procmail recipe can use `spamc` for faster spam-checking.
- User preference files, autowhitelists, and Bayesian databases can be used.

There are also some disadvantages:

- sendmail must complete the SMTP transaction and accept an email message for local delivery before spam-checking takes place. Accordingly, you can't save bandwidth or mailbox space by rejecting spam during the SMTP transaction.

- sendmail only runs the local delivery agent for email destined for a local recipient. You cannot create a spam-checking gateway with this approach.

To configure sendmail to use procmail as its local delivery agent, add the following line to your *sendmail.mc* file (before the `MAILER(`local')` line) and regenerate *sendmail.cf* from it:

```
FEATURE(`local_procmail',`/path/to/procmail')dnl
```

When you restart sendmail, it will use procmail instead of the system's default local MDA (e.g., */bin/mail*) for mail delivery.

Next, configure procmail to invoke SpamAssassin. If you want to invoke SpamAssassin on behalf of every user, do so by editing the */etc/procmailrc* file. Example 5-1 shows an */etc/procmailrc* that invokes SpamAssassin.

Example 5-1. A complete /etc/procmailrc

```
DROPPRIVS=yes
PATH=/bin:/usr/bin:/usr/local/bin
SHELL=/bin/sh

# Spamassassin
:0fw
* <300 000
|/usr/bin/spamassassin
```

If you run `spamd`, replace the call to `spamassassin` in *procmailrc* with a call to `spamc` instead. Using `spamc`/`spamd` will significantly improve performance on most systems, but makes it more difficult to allow users to write their own rules.

Spam-Checking During SMTP

If you want to refuse spam before it reaches your recipients, or set up a spam-checking gateway to an internal email server, you need a way to perform spam-checking during the SMTP transaction. If a message is found to be spam, you may want to refuse it and end the SMTP session, or accept it and add headers that users can use in their mail client filters. sendmail provides a general-purpose filtering interface, called *milter*, for use during the SMTP transaction.

The Milter Interface

In sendmail's parlance, milter refers to several things. Milter is an application programming interface (API) for writing filters for sendmail, and a protocol for communication between sendmail and a filter. A milter is also a filter program written using this API that listens for connections from a `sendmail` process and defines functions to call at different points of the SMTP transaction to accept, reject, discard, temporarily refuse, or modify a message. The milter library, *libmilter*, provides most of the code required to set up a milter and manage the work of calling your filtering functions during an SMTP transaction.

A milter can provide functions that sendmail will call at the following points in an SMTP transaction:

- When a mail client connects to sendmail
- After the SMTP `HELO` or `EHLO` commands
- After the SMTP `MAIL FROM` command
- After the SMTP `RCPT TO` command
- After each message header is transmitted during the DATA step
- After all message headers are transmitted
- After each piece of the message body is transmitted
- At the end of the DATA step, after the entire message has been transmitted
- When the SMTP transaction is aborted
- When the client connection is closed

Milter functions can perform the following operations on a message:

- Add, change, or delete a header
- Add or remove a recipient
- Replace the message body
- Reject a connection, message, or recipient
- Temporarily fail a connection, message, or recipient
- Accept and discard a message
- Accept a message

Milters operate as daemons. They are typically started before `sendmail` during system startup and listen for connections from a `sendmail` process on a TCP or Unix domain socket. Milters do not have to be run as *root*. For more information about writing milters, visit *http://www.milter.org*.

You configure sendmail to use a milter by adding an `INPUT_MAIL_FILTER( )` macro to the *sendmail.mc* configuration file and generating a new *sendmail.cf* file. Example 5-2 shows parts of a *sendmail.mc* file that includes a milter.

Example 5-2. A sendmail.mc file with a milter

```
divert(0)dnl
VERSIONID(`example mc')dnl
OSTYPE(linux)dnl
DOMAIN(generic)dnl
...
INPUT_MAIL_FILTER(`mymilter', `S=unix:/var/run/mymilter.sock, F=T, T=S:60s;R:60s;E:
5m')dnl
```

Example 5-2. A sendmail.mc file with a milter (continued)

```
...
MAILER(smtp)dnl
MAILER(local)dnl
MAILER(procmail)dnl
```

The `INPUT_MAIL_FILTER` macro takes two arguments. The first provides the name of the milter (`mymilter` in Example 5-2), and the second tells sendmail how to interact with the milter. The second argument in turn consists of several instructions, separated by commas:

`S=`*`socket description`*
: This argument describes how sendmail should connect to the milter. The socket description consists of a protocol (`unix` for a Unix domain socket, `inet` for a TCP/IP socket, `inet6` for a TCP/IPv6 socket), a colon, and a protocol-specific address. For Unix domain sockets, the address is the path to the socket file. For TCP sockets, the address is in the form *`port@host`*.

`F=`*`failure mode`*
: This argument determines how sendmail will behave if it fails to connect to the milter process. Use `F=T` to cause sendmail to temporarily refuse email when it can't contact the milter. Use `F=R` to cause sendmail to reject connections when it can't contact the milter. Omit an `F=` argument to cause sendmail to accept messages without filtering when it can't contact the milter.

`T=`*`timeout list`*
: This argument determines how long sendmail should wait for the milter to respond before treating the connection attempt as a failure. It consists of a set of states and the amount of time to allow for each, separated by semicolons. In Example 5-1, sendmail uses a 60-second timeout for sending data to the milter (`S:60s`), a 60-second timeout for reading replies from the milter (`R:60s`), and a 5-minute timeout for waiting for the milter's final acknowledgment after sending the message (`E:5m`). There is also a `C` timeout for connecting to the milter. If you leave any timeouts unspecified, sendmail uses its default timeouts: 10 seconds for sending and reading, and 5 minutes for connecting and final acknowledgment.

The `INPUT_MAIL_FILTER` macro results in the following lines being added to the *sendmail.cf* file when you generate it:

```
O InputMailFilters=mymilter
...
Xmymilter, S=unix:/var/run/mymilter.sock, F=T, T=S:60s;R:60s;E:5m
```

SpamAssassin itself is not a milter. However, several milters have been written that invoke SpamAssassin on messages and then take action during the SMTP transaction.

Milter in sendmail 8.11

The milter interface was formally announced in sendmail 8.12 but is available as an experimental feature in sendmail 8.11. To use milter in sendmail 8.11, add the following line to your *sendmail.mc* file:

```
    define(`_FFR_MILTER')dnl
```

Milter support in sendmail 8.11 is not as complete as in sendmail 8.12, however, and I strongly encourage you to upgrade to sendmail 8.12 or later rather than use sendmail 8.11's milter subsystem.

Older versions of sendmail do not provide milter. If you must use one of these versions, you are limited to integrating SpamAssassin through procmail.

MIMEDefang

MIMEDefang is one of the most popular sendmail milters. It provides a general framework for performing milter functions in Perl and comes with a default configuration that performs several functions:

- Messages can be checked with a virus scanner, and messages carrying viruses can be refused, discarded, or quarantined.
- MIME attachments can be examined, and messages can be refused, discarded, or quarantined if they contain attached files with given filename extensions (e.g., extensions that denote executable Windows files).
- The HTML attachment in a message of type *multipart/alternative* (containing both text and HTML versions of the same message) can be dropped.
- SpamAssassin can be invoked on the message, and spam can be refused, discarded, quarantined, or tagged.

MIMEDefang is developed by Roaring Penguin Software and is available as free software at *http://www.mimedefang.org*. Roaring Penguin also produces commercial products, CanIt and CanIt-PRO, which are based on MIMEDefang and SpamAssassin and add several other features including web-based interfaces for administrators and users.

The rest of this section details the installation, operation, and customization of MIMEDefang 2.42 as an example of a full-scale, milter-based approach to using SpamAssassin. MIMEDefang's other functions, such as virus-checking, are mentioned but not covered in detail; read the MIMEDefang documentation for more information.

Use the latest available version of MIMEDefang. In particular, only versions 2.42 and later support SpamAssassin 3.0.

Installing MIMEDefang

MIMEDefang is written in Perl and invokes SpamAssassin through the *Mail::SpamAssassin* Perl modules. Because MIMEDefang itself is a daemon, you do not need to run `spamd`. It's easiest to install SpamAssassin (and your antivirus software) first and then install MIMEDefang.

A good way to begin a MIMEDefang installation is to verify that you have the prerequisite Perl modules on hand. MIMEDefang requires sendmail 8.12 (or later). MIMEDefang also requires several Perl modules, including: *MIME::Tools*, *IO::Stringy*, *MIME::Base64*, *MailTools*, *Digest::SHA1*, and *HTML::Parser*. Most of them can be installed using CPAN.

MIMEDefang will not work correctly with the standard version of *MIME::Tools* 5.411a. Either install *MIME::Tools* 6 or later, or install the special version of *MIME::Tools* 5.411a available from Roaring Penguin's web site.

You should create a new user account and group for running MIMEDefang; the usual name for both the user and group is *defang*. This user will own MIMEDefang's files, and the user (or group) must have access to SpamAssassin's configuration and database files as well.

MIMEDefang uses two important directories. It uses */var/spool/MIMEDefang* as a working directory for unpacking email messages and scanning them. For optimal performance, place this directory on a fast disk—even a RAM disk if your operating system supports it and you have enough memory to spare. MIMEDefang stores quarantined email messages in */var/spool/MD-Quarantine*. Speed is not so critical with this directory, and it should never be located on a RAM disk because you will want to be sure that you can access quarantined files. Create these directories before you install MIMEDefang. The directories should be owned by user and group *defang* and should not be world-readable or world-searchable.

Next, download the MIMEDefang source code from *http://www.roaringpenguin.com*, unpack it, run the `configure` script, `make`, and perform a `make install` as *root*. Example 5-3 shows this process from the point of running the `configure` script:

Example 5-3. Compiling MIMEDefang

```
$ ./configure
creating cache ./config.cache
...
```

Example 5-3. Compiling MIMEDefang (continued)

```
creating config.h

*** Virus scanner detection results:
H+BEDV   'antivir'   NO (not found)
Vexira   'vexira'    NO (not found)
NAI      'uvscan'    NO (not found)
BDC      'bdc'       NO (not found)
Sophos   'sweep'     NO (not found)
TREND    'vscan'     NO (not found)
CLAMSCAN 'clamav'    YES - /usr/bin/clamscan
AVP      'AvpLinux'  NO (not found)
FSAV     'fsav'      NO (not found)
FPROT    'f-prot'    NO (not found)
SOPHIE   'sophie'    NO (not found)
NVCC     'nvcc'      NO (not found)
CLAMD    'clamd'     YES - /usr/sbin/clamd
File::Scan           NO (not found)
TROPHIE  'trophie'   NO (not found)

Found Mail::SpamAssassin.  You may use spam_assassin_* functions
Did not find Anomy::HTMLCleaner.  Do not use anomy_clean_html( )
Found HTML::Parser.  You may use append_html_boilerplate( )

Note: SpamAssassin, File::Scan, HTML::Parser and Anomy::HTMLCleaner are
detected at run-time, so if you install or remove any of those modules, you
do not need to re-run ./configure and make a new mimedefang.pl.
$ make
gcc -g -O2 -Wall -Wstrict-prototypes -pthread -D_POSIX_PTHREAD_SEMANTICS -DPERL_PATH=\"/
usr/local/bin/perl\" -DMIMEDEFANG_PL=\"/usr/local/bin/mimedefang.pl\" -DRM=\"/bin/rm\" -
DVERSION=\"2.42\" -DSPOOLDIR=\"/var/spool/MIMEDefang\" -DQDIR=\"/var/spool/MD-Quarantine\
" -DCONFDIR=\"/etc/mail\" -I../sendmail-8.12.11/include -c -o mimedefang.o mimedefang.c
...
$ su
Password: XXXXXX
# make install
mkdir -p /etc/mail && chmod 755 /etc/mail
...
Please create the spool directory, '/var/spool/MIMEDefang',
if it does not exist.  Give it mode 700, and make
it owned by the user you intend to run MIMEDefang as.
Please do the same with the quarantine directory, '/var/spool/MD-Quarantine'.
#
```

The following programs and files are installed:

`mimedefang`
: The milter itself. This program receives requests from sendmail to filter messages and pass them on to `mimedefang-multiplexor` to perform the checks. It then communicates the results back to sendmail.

`mimedefang-multiplexor`
: A program to receive requests from `mimedefang` and farm them out to a pool of *mimedefang.pl* Perl processes for scanning. It is responsible for maintaining the process pool, creating and destroying processes as necessary. This approach minimizes the time and CPU overhead required in starting new processes for each scan.

`mimedefang.pl`
: A Perl script to perform all of the message-checking functions of MIMEDefang. During the several stages of checking a message, this script calls functions defined in */etc/mail/mimedefang-filter*.

`md-mx-ctrl`
: A command-line tool for viewing the status of the multiplexor or for ordering it to reload its slave processes.

`watch-mimedefang`
: A graphical interface based on Tcl/Tk.

/etc/mail/spamassassin/sa-mimedefang.cf
: A sitewide configuration file used by MIMEDefang. By default, MIMEDefang's install process generates a simple file, with few options.

/etc/mail/mimedefang-filter
: A file containing Perl subroutines called by *mimedefang.pl* at different stages of message-processing. These subroutines check messages or message parts, and direct MIMEDefang to accept, quarantine, discard, or bounce a message. MIMEDefang installs a default *mimedefang-filter* that invokes SpamAssassin to add an *X-Spam-Score* header and a SpamAssassin report to all messages. To implement more complex spam-checking behavior, you'll edit *mimedefang-filter*. This file is discussed in greater detail in the "MIMEDefang Configuration" section, later in this chapter.

Starting the MIMEDefang multiplexor

To run MIMEDefang, you must start two processes: the multiplexor (`mimedefang-multiplexor`) and the milter (`mimedefang`). You should start the multiplexor first because the milter process will connect to it. Start each process as *root*; each changes its uid to the *defang* user after startup.

`mimedefang-multiplexor` has over a dozen command-line options, but you will typically need to use only a few of them. The most common are described here; for complete information, see the manpage.

`-U` *`user`*
: Instructs `mimedefang-multiplexor` to run as the given user (e.g., *defang*). Running as a non-*root* user is an important security measure.

`-s` *`/path/to/socket`*
: Specifies the path to the Unix domain socket that the multiplexor will use to listen for requests from the milter process. It defaults to */var/spool/MIMEDefang/mimedefang-multiplexor.sock*.

`-p` *`filename`*
: Causes the multiplexor to write its process ID to the specified file. You can use this ID to signal the multiplexor to reread the filter when you change it or to stop the multiplexor (these operations are discussed later in this section).

`-m` *`number-of-slaves`*
: Specifies the minimum number of slave, *mimedefang.pl* processes that should be running at any given time. This value defaults to 0, but on most systems, you want to have at least two slave processes running at all times to minimize startup overhead.

`-x` *`number-of-slaves`*
: Specifies the maximum number of slave, *mimedefang.pl* processes that should be running at any given time. This value defaults to 2, but busy mail servers will require more than two processes to be available at any given time. You should plan to increase this value to 5, 10, or higher, depending on your needs.

`-q` *`number-of-requests`*
: Causes the multiplexor to queue an incoming request when a multiplexor is not immediately available to service that request. By default, the multiplexor causes sendmail to temporarily fail a message when all slave processes are busy (returning a 4xx SMTP status code to the sending MTA, which should retain the message in its queue and try to deliver it again later).

-D
: Causes the multiplexor to run in the foreground, for debugging purposes. Without this option, the multiplexor detaches from the terminal and runs in the background.

A typical invocation of `mimedefang-multiplexor` might be:

```
/usr/local/bin/mimedefang-multiplexor -U defang -p /var/run/mimedefang-multiplexor.
pid -m 2 -x 10
```

Checking multiplexor status

Once the multiplexor is running, use the `md-mx-ctrl` command to examine its status. `md-mx-ctrl status` provides a human-readable status report on the multiplexor's slave processes; `md-mx-ctrl msgs` shows the total number of messages processed by the multiplexor. If you're using a nondefault socket for the multiplexor, you can specify that socket to `md-mx-ctrl` using the `–s` *`/path/to/socket`* command-line option. Example 5-4 shows these `md-mx-ctrl` invocations and their output. On the system in the example, the multiplexor has been configured with a minimum of two slaves (both of which are idle) and a maximum of ten, and has processed 17,366 messages.

Example 5-4. Invoking md-mx-ctrl

```
# md-mx-ctrl status
Max slaves: 10
Slave 0: stopped
Slave 1: stopped
Slave 2: idle
Slave 3: stopped
Slave 4: stopped
Slave 5: stopped
Slave 6: idle
Slave 7: stopped
Slave 8: stopped
Slave 9: stopped
# md-mx-ctrl msgs
17366
```

Starting the MIMEDefang milter

`mimedefang` performs a simpler task than the multiplexor. Its job is to receive filtering requests from sendmail and pass them on to the multiplexor to handle. Accordingly, it has fewer command-line options. Here are the most commonly used options.

`-p` *`/path/to/socket`*
: Specifies the path to the Unix domain socket that the milter process will listen on for requests from sendmail. This path must match the path you specify in sendmail's `INPUT_MAIL_FILTER( )` macro. A typical choice is */var/spool/MIMEDefang/mimedefang.sock*, which is a required option.

`-m` *`/path/to/multiplexor/socket`*
: Specifies the Unix domain socket on which the multiplexor is listening for requests. `mimedefang` sends requests to the multiplexor on this socket. This option is required, and the value should match that of the multiplexor's `–s` option (typically */var/spool/MIMEDefang/mimedefang-multiplexor.sock*).

`-U` *`user`*
: Instructs `mimedefang` to run as the given user (e.g., *defang*). You must provide the same user to `mimedefang-multiplexor` and `mimedefang`.

`-P` *`filename`*
: Directs `mimedefang` to write its process ID to the specified file. Note that this option uses a capital P.

A typical invocation of `mimedefang` might be:

```
/usr/local/bin/mimedefang -U defang -P /var/run/mimedefang.pid \
-p /var/spool/MIMEDefang/mimedefang.sock \
-m /var/spool/MIMEDefang/mimedefang-multiplexor.sock
```

A sample boot script for automatically starting and stopping MIMEDefang can be found in the *examples* directory of MIMEDefang's source code. Editing this script and installing it with your other system boot scripts is an easy way to properly configure MIMEDefang, as it lists all of the multiplexor and milter process options as shell variables. Ideally, the script should run before sendmail's startup script so that the milter socket is in place before sendmail starts. Likewise, you should stop sendmail before you stop MIMEDefang's process.

Verifying the MIMEDefang processes

You can use the ps command to verify that all your MIMEDefang processes are running. Example 5-5 shows the process listing and the contents of */var/spool/MIMEDefang* and */var/spool/MD-Quarantine* on a typical system running sendmail and MIMEDefang. MIMEDefang's processes include one `mimedefang-multiplexor` process, three slave *mimedefang.pl* processes started by the multiplexor for scanning messages, and four `mimedefang` milter processes started by sendmail. All processes are running as user *defang*. The */var/spool/MIMEDefang* directory contains working directories used temporarily by MIMEDefang (names starting with "mdefang"), as well as Unix domain sockets and pid files. The */var/spool/MD-Quarantine* directory includes subdirectories holding quarantined messages.

Example 5-5. Processes and layout of a typical MIMEDefang system

```
# ps auxw | egrep 'mime'
defang   27145  0.0  0.0  1312  688 ?        S    Jan15   0:42 /usr/local/bin/mimedefang-
multiplexor -p /var/spool/MIMEDefang/mimedefang-multiplexor.pid -m 2 -x 10 -U defang -b
300 -l -T -s /var/spool/MIMEDefang/mimedefang-multiplexor.sock
defang   27162  0.0  0.1  2552  856 ?        S    Jan15   0:00 /usr/local/bin/mimedefang
-P /var/spool/MIMEDefang/mimedefang.pid -U defang -m /var/spool/MIMEDefang/mimedefang-
multiplexor.sock -p /var/spool/MIMEDefang/mimedefang.sock
defang   20548  1.0  2.8 23464 22416 ?       S    12:05   1:43 perl -w /usr/local/bin/
mimedefang.pl -server
defang   25089  0.0  0.1  2552  856 ?        S    13:57   0:00 /usr/local/bin/mimedefang
-P /var/spool/MIMEDefang/mimedefang.pid -U defang -m /var/spool/MIMEDefang/mimedefang-
multiplexor.sock -p /var/spool/MIMEDefang/mimedefang.sock
defang   25142  0.0  0.1  2552  856 ?        S    13:59   0:00 /usr/local/bin/mimedefang
-P /var/spool/MIMEDefang/mimedefang.pid -U defang -m /var/spool/MIMEDefang/mimedefang-
multiplexor.sock -p /var/spool/MIMEDefang/mimedefang.sock
defang   25589  0.0  0.1  2552  856 ?        S    14:11   0:00 /usr/local/bin/mimedefang
-P /var/spool/MIMEDefang/mimedefang.pid -U defang -m /var/spool/MIMEDefang/mimedefang-
multiplexor.sock -p /var/spool/MIMEDefang/mimedefang.sock
defang   26616  0.3  2.6 21588 20572 ?       S    14:35   0:04 perl -w /usr/local/bin/
mimedefang.pl -server
defang   26617  0.2  2.6 21492 20492 ?       S    14:35   0:03 perl -w /usr/local/bin/
mimedefang.pl -server

# ls -l /var/spool/MIMEDefang
drwx------    3 defang   defang        149 Jan 28 14:47 mdefang-iOSKkoMD027104
drwx------    3 defang   defang        149 Jan 28 14:48 mdefang-iOSKlwMB027198
-rw-------    1 defang   defang          6 Jan 15 10:40 mimedefang-multiplexor.pid
srw-------    1 defang   defang          0 Jan 15 10:40 mimedefang-multiplexor.sock
-rw-------    1 defang   defang          6 Jan 15 10:40 mimedefang.pid
srwx------    1 defang   defang          0 Jan 15 10:40 mimedefang.sock
```

Example 5-5. Processes and layout of a typical MIMEDefang system (continued)

```
# ls -l /var/spool/MD-Quarantine
drwx------   2 defang   defang       212 Dec 27 10:37 qdir-2004-01-28-10.37.35-001
drwx------   2 defang   defang       212 Dec 27 16:25 qdir-2004-01-28-16.25.03-001
```

Customizing MIMEDefang

Use the *mimedefang-filter* file to configure the actions that MIMEDefang takes when filtering messages. The file is written in Perl. MIMEDefang distributes and installs a working sample file, typically in */etc/mail*, but you will need to modify several settings in the file for your local environment. Example 5-6 shows the configuration settings near the beginning of this file. You should always change `$AdminAddress`, `$AdminName`, and `$DaemonAddress`. Generally, `$AddWarningsInline` and `md_graphdefang_log_enable( )` can be left unchanged, and `$MaxMIMEParts` should be uncommented to prevent denial-of-service attacks.

Example 5-6. Configuration section of mimedefang-filter

```
#***********************************************************************
# Set administrator's e-mail address here.  The administrator receives
# quarantine messages and is listed as the contact for site-wide
# MIMEDefang policy.  A good example would be 'defang-admin@mydomain.com'
#***********************************************************************
$AdminAddress = 'postmaster@localhost';
$AdminName = "MIMEDefang Administrator's Full Name";

#***********************************************************************
# Set the e-mail address from which MIMEDefang quarantine warnings and
# user notifications appear to come.  A good example would be
# 'mimedefang@mydomain.com'.  Make sure to have an alias for this
# address if you want replies to it to work.
#***********************************************************************
$DaemonAddress = 'mimedefang@localhost';

#***********************************************************************
# If you set $AddWarningsInline to 1, then MIMEDefang tries *very* hard
# to add warnings directly in the message body (text or html) rather
# than adding a separate "WARNING.TXT" MIME part.  If the message
# has no text or html part, then a separate MIME part is still used.
#***********************************************************************
$AddWarningsInline = 0;

#***********************************************************************
# To enable syslogging of virus and spam activity, add the following
# to the filter:
# md_graphdefang_log_enable( );
# You may optionally provide a syslogging facility by passing an
# argument such as:  md_graphdefang_log_enable('local4');  If you do this, be
# sure to setup the new syslog facility (probably in /etc/syslog.conf).
# An optional second argument causes a line of output to be produced
```

Example 5-6. Configuration section of mimedefang-filter (continued)

```
# for each recipient (if it is 1), or only a single summary line
# for all recipients (if it is 0.)  The default is 1.
# Comment this line out to disable logging.
#***********************************************************************
md_graphdefang_log_enable('mail', 1);

#***********************************************************************
# Uncomment this to block messages with more than 50 parts.  This will
# *NOT* work unless you're using Roaring Penguin's patched version
# of MIME tools, version MIME-tools-5.411a-RP-Patched-02 or later.
#
# WARNING: DO NOT SET THIS VARIABLE unless you're using at least
# MIME-tools-5.411a-RP-Patched-02; otherwise, your filter will fail.
#***********************************************************************
# $MaxMIMEParts = 50;
```

The remainder of the *mimedefang-filter* file is a set of Perl functions that *mimedefang.pl* will call when checking a message. You can modify these functions to customize MIMEDefang's behavior. The functions include:

`filter_begin( )`
: Called with no arguments at the start of filtering. Suitable for setting variables that you expect to use throughout the filter, or for performing whole-message checks like virus-scanning immediately.

`filter_multipart(`*`entity,name,extension,type`*`)`
: Called for each MIME part of the message that contains other MIME parts within it. The *`entity`* is a *MIME::Entity* object, *`name`* is the suggested filename of the part, *`extension`* is the file extension, and *`type`* is the MIME type. Suitable for validating MIME parts or refusing specific multipart types (e.g., message/partial).

`filter(`*`entity,name,extension,type`*`)`
: Called for each MIME part of the message that does *not* contain other MIME parts within it. Arguments are the same as for `filter_multipart( )`. Suitable for validating filenames, virus-scanning individual MIME parts, or refusing specific MIME types.

`filter_end(`*`entity`*`)`
: Called at the end of filtering with the *MIME::Entity* object representing the entire message to be returned to sendmail. Suitable for checking variables that you set elsewhere in the filter and performing computationally expensive whole-message checks like spam-tagging if necessary.

These functions can make decisions about the disposition or modification of individual message parts by calling one of the MIMEDefang action functions. In most cases, actions should be taken only by the `filter( )` or `filter_multipart( )` functions. The most commonly used action functions are:

`action_accept()`, `action_accept_with_warning(`*`string`*`)`
: Accept the current message part, possibly adding a warning to the message.

`action_drop()`, `action_drop_with_warning(`*`string`*`)`
: Drop the current message part, possibly adding a warning to the message.

`action_replace_with_warning(`*`string`*`)`
: Replace the current message part with a warning message.

`action_quarantine(`*`entity,string`*`)`
: Drop and quarantine the current message part, and add a warning to the message.

`action_quarantine_entire_message(`*`string`*`)`
: Quarantine the entire message, and add a warning to the administrator notification if one is generated. This action only quarantines; it does not also discard or bounce the message. You must call `action_discard()` or `action_bounce()` afterward.

`action_bounce(`*`string[,SMTP reply code[,DSN code]]`*`)`
: Instruct sendmail to reject the message with *`string`* returned to the sender as the reason for rejection. You can optionally specify an SMTP reply code (which defaults to 554) and a DSN code (which defaults to 5.7.1). Bouncing a message does not stop MIMEDefang from continuing to process other message parts; the bounce occurs after all parts have been processed.

`action_tempfail(`*`string[,SMTP reply code[,DSN code]]`*`)`
: Instruct sendmail to temporarily reject the message with *`string`* returned to the sender as the reason for rejection. You can optionally specify an SMTP reply code (which defaults to 450) and a DSN code (which defaults to 4.7.1).

`action_discard()`
: Discard the entire message silently once all parts have been processed.

`action_notify_sender(`*`string`*`)`
: Generate an email notification back to the message sender containing the given string, which may consist of multiple lines.

`action_notify_administrator(`*`string`*`)`
: Generate an email notification back to the MIMEDefang administrator containing the given string, which may consist of multiple lines.

`action_add_part(`*`entity,type,encoding,data,fname,disposition[,offset]`*`)`
: Add a new MIME part to the message represented by *`entity`*. The new part will have a MIME content-type of *`type`* and content-encoding of *`encoding`*. The new part itself should be stored in *`data`* and its associated filename in *`fname`*. The MIME content-disposition is given by *`disposition`*. The optional *`offset`* specifies where to add the part; it defaults to –1 (add at end). This action may be performed in `filter_end()`.

`action_add_header(`*`header,value`*`)`
: Add a new header to the message. The header's name is given in *`header`*, without a trailing colon, and the value to set the header to is given in *`value`*. It is possible to add multiple headers with the same name.

`action_change_header(`*`header,value[,index]`*`)`
: Change a header in the message. The header's name is given in *`header`*, without a trailing colon, and the new value to set the header to is given in *`value`*. If *`index`* is given, changes the *`index`*'th header with that name. Changing a header that does not exist will add a new header.

`action_delete_header(`*`header[,index]`*`)`
: Delete a header in the message. The header's name is given in *`header`*, without a trailing colon. If *`index`* is given, deletes the *`index`*'th header with that name instead of the first one.

`action_delete_all_headers(`*`header`*`)`
: Deletes all headers in the message with a given name. The header's name is given in *`header`*, without a trailing colon.

If you call one of the notification functions (e.g., `action_notify_sender`), MIMEDefang creates a notification message and sends it by invoking sendmail in its *deferred mode*; sendmail will enqueue the notification message in its client mail queue rather than sending it immediately. You must run a sendmail process that periodically sends messages in the client queue. One way to do so is to issue the following command at system boot (via a boot script):

```
/usr/sbin/sendmail -Ac -q5m
```

See the sendmail documentation for more information about deferred mode and client queue runners.

By calling these functions, you can configure MIMEDefang to suit nearly any email management policy you wish to institute.

When you make changes to the `mimedefang-filter` script, you must signal `mimedefang-multiplexor` to reread the configuration and restart its slave processes. The easiest way to signal the multiplexor is to use the `md-mx-ctrl reread` command. Another way is to use the `kill -INT` *`process-id`* command to send a SIGINT signal to the multiplexor process; you can identify the process ID from *ps* output or by examining the pid file if the multiplexor was started with the `–p` option.

SpamAssassin Integration

MIMEDefang expects to find a SpamAssassin configuration file called *sa-mimedefang.cf* in your sitewide configuration directory (usually */etc/mail/spamassassin*). If it doesn't, it will also look for *local.cf* in the same directory. This gives you the flexibility of creating different SpamAssassin configurations to be used when SpamAssassin is invoked by MIMEDefang and when SpamAssassin is invoked by local users or scripts.

> If you're going to be invoking SpamAssassin only through MIMEDefang, or if there should be no differences in the configuration file based on how MIMEDefang is invoked, consider making a hard or symbolic link from *local.cf* to *sa-mimedefang.cf*. MIMEDefang will find the configuration file it first looks for, and you will avoid the possibility of later creating two different configurations.

When running SpamAssassin via MIMEDefang, you may not use any of SpamAssassin's configuration directives that modify a mail message. Attempting to modify the *Subject* header or add new headers using SpamAssassin directives will not work. All such changes must be performed by MIMEDefang in the `mimedefang-filter` script.

If you want SpamAssassin to perform network-based tests (such as DNSBL lookups), you must add a line to `mimedefang-filter` (just after the `$AdminName` setting works well) to set the `$SALocalTestsOnly` variable to 0, like this:

```
$SALocalTestsOnly = 0;
```

The section of the default `mimedefang-filter` that handles spam-tagging appears in the `filter_end( )` function and is agreeably easy to read. It is presented in Example 5-7.

Example 5-7. Spam-tagging section of mimedefang-filter

```
# Spam checks if SpamAssassin is installed
if ($Features{"SpamAssassin"}) {
    if (-s "./INPUTMSG" < 300*1024) {
        # Only scan messages smaller than 300kB.  Larger messages
        # are extremely unlikely to be spam, and SpamAssassin is
        # dreadfully slow on very large messages.
        my($hits, $req, $names, $report) = spam_assassin_check( );
        my($score);
        if ($hits < 40) {
            $score = "*" x int($hits);
        } else {
            $score = "*" x 40;
        }
        # We add a header which looks like this:
        # X-Spam-Score: 6.8 (******) NAME_OF_TEST,NAME_OF_TEST
        # The number of asterisks in parens is the integer part
        # of the spam score clamped to a maximum of 40.
        # MUA filters can easily be written to trigger on a
        # minimum number of asterisks...
        action_change_header("X-Spam-Score", "$hits ($score) $names");
        if ($hits >= $req) {
            md_graphdefang_log('spam', $hits, $RelayAddr);
```

Example 5-7. Spam-tagging section of mimedefang-filter (continued)

```
                # If you find the SA report useful, add it, I guess...
                action_add_part($entity, "text/plain", "-suggest",
                                "$report\n",
                                "SpamAssassinReport.txt", "inline");
            } else {
                # Delete any existing X-Spam-Score header?
                action_delete_header("X-Spam-Score");
            }
        }
    }
```

First, the code checks to be sure that MIMEDefang detected SpamAssassin on the system when it started. It then checks to be sure that the *INPUTMSG* file, which contains the message to scan, is smaller than 300 kilobytes. If that's the case, the code calls MIMEDefang's `spam_assassin_check( )` function, which uses *Mail::SpamAssassin* to check the message and returns the number of hits, number of required hits for tagging, names of tests hit, and the text of SpamAssassin's spam report for the message. The code creates a `$score` variable containing one asterisk for each hit (up to 40).

Next, the code in Example 5-7 calls the MIMEDefang `action_change_header( )` function to change (or add) the *X-Spam-Score* header. The header will include the number of hits (expressed numerically and as a line of asterisks) and the names of tests that matched.

If the number of hits is greater than or equal to the required number to declare the message spam, the code calls MIMEDefang's `md_graphdefang_log( )` function to make a log entry and then adds the SpamAssassin report text to the message as an additional MIME part using the `action_add_part( )` function. If the number of hits is less than the required number for tagging, the script removes the *X-Spam-Score* header.

You might customize this code in `filter_end( )` in several easy ways to suit your needs. By commenting out the `action_delete_header( )` line, you can have the *X-Spam-Score* header added to all messages, spam or not. If you want to modify the *Subject* header of spam messages as SpamAssassin does, add the following code before the `action_add_part( )` line:

```
action_change_header("Subject", "*****SPAM***** $Subject");
```

The `$Subject` variable will already contain the message subject.

Remember that you must signal the MIMEDefang milter to reread *mimedefang-filter* whenever you change it or any Perl modules on which it depends—including SpamAssassin and its configuration. If you update SpamAssassin or modify settings in */etc/mail/spamassassin/sa-mimedefang.cf*, you should signal the milter.

Adding sitewide Bayesian filtering

Adding a sitewide Bayesian filter for use with MIMEDefang is relatively easy. Use the usual SpamAssassin use_bayes and bayes_path directives in *sa-mimedefang.cf*, and ensure that the *defang* user has permission to create the databases in the directory named in bayes_path. One way to do this is to create a directory for the databases that is owned by *defang*, such as */var/spool/MD-Bayes*. Another option is to locate the databases in a directory owned by another user but to create them ahead of time and chown them to *defang*. If local users need access to the databases (e.g., they will be running sa-learn), you may have to make the databases readable or writable by a group other than *defang* and adjust the bayes_file_mode, or make them world-readable or world-writable. Doing so, however, puts the integrity of your spam-checking at the mercy of the good intentions and comprehension of your users.

Adding sitewide autowhitelisting

In SpamAssassin 3.0, autowhitelisting is easy to enable. You need only add the usual autowhitelist directives to *sa-mimedefang.cf* to determine where and how the autowhitelist database will be stored. Be sure to enable the use_auto_whitelist configuration option to turn on autowhitelisting.

Using a sitewide autowhitelist database in SpamAssassin 2.63 requires just a bit more effort. In addition to adding the SpamAssassin autowhitelist directives to *sa-mimedefang.cf*, you must modify *mimedefang.pl* to provide SpamAssassin with an address list factory, as discussed in Chapter 4. Example 5-8 shows the spam_assassin_init( ) function in *mimedefang.pl*. Add the emphasized lines to support autowhitelisting. Don't forget to signal mimedefang-multiplexor to reread its configuration after making these changes.

Example 5-8. Adding an address list factory to mimedefang.pl

```
sub spam_assassin_init (;$) {

    unless ($Features{"SpamAssassin"}) {
        md_syslog('err', "$MsgID: Attempt to call SpamAssassin function, but SpamAssassin
is not installed.");
        return undef;
    }

    if (!defined($SASpamTester)) {
        my $config = shift;
        unless ($config)
        {
            if (-r "/etc/mail/spamassassin/sa-mimedefang.cf") {
                $config = "/etc/mail/spamassassin/sa-mimedefang.cf";
            } elsif (-r "/etc/mail/spamassassin/local.cf") {
                $config = "/etc/mail/spamassassin/local.cf";
            } else {
```

Example 5-8. Adding an address list factory to mimedefang.pl (continued)

```
                $config = "/etc/mail/spamassassin.cf";
            }
        }

        $SASpamTester = Mail::SpamAssassin->new({
            local_tests_only   => $SALocalTestsOnly,
            dont_copy_prefs    => 1,
            userprefs_filename => $config});

        require Mail::SpamAssassin::DBBasedAddrList;
        my $awl = Mail::SpamAssassin::DBBasedAddrList->new( );
        $SASpamTester->set_persistent_address_list_factory ($awl);
    }

    return $SASpamTester;
}
```

Adding per-domain or per-user streaming

By default, MIMEDefang processes each message once and applies SpamAssassin's spam determination to the message. This process works well if you run a small mail server for a single domain, but it presents a problem for mail gateways, virtual hosts, and larger servers. What should be done when an email message is received for multiple recipients—possibly at multiple domains? MIMEDefang provides two functions that you can use to implement solutions to this problem, `stream_by_recipient( )` and `stream_by_domain( )`. Each works in the same way.

If you add a call to `stream_by_recipient( )` to the `filter_begin( )` function, `stream_by_recipient( )` checks to see if a message has only a single recipient. If so, it returns 0, and the filter should continue to work on the message. If the message has multiple recipients, `stream_by_recipient( )` *reinjects* the message by connecting to sendmail and resubmitting the message as a series of new messages, one for each recipient of the original message. Figure 5-1 illustrates this process. In this case, `stream_by_recipient( )` returns 1, and the original, multirecipient message should be discarded. When the new single-recipient messages arrive at the filter, they will pass through `stream_by_recipient( )` and continue on to the rest of the filter, which can now safely perform per-recipient functions (such as using personal whitelists and blacklists or other user preferences).

`stream_by_domain( )` works similarly but only reinjects one new copy of a message for each recipient domain in the original message. The rest of the filter can behave differently for different recipient domains, which permits virtual hosting providers to apply different spam criteria for different domains they host.

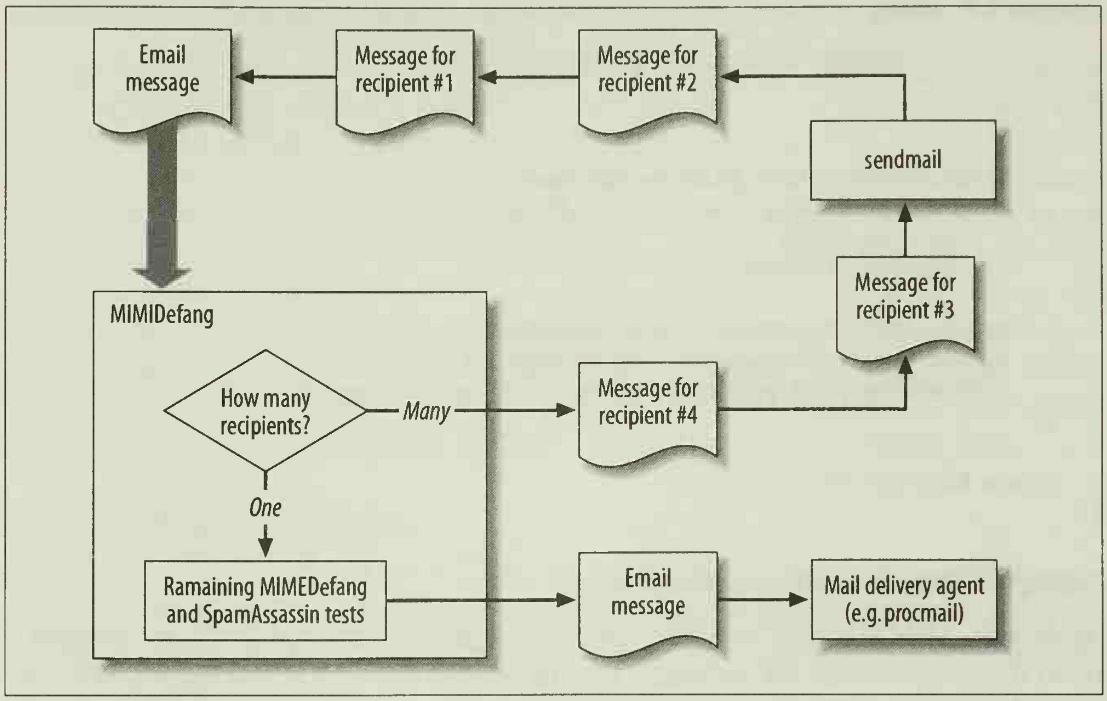


Figure 5-1. Streaming by recipients

Although some MIMEDefang features will work with sendmail 8.11, `stream_by_domain( )` and `stream_by_recipient( )` *require* sendmail 8.12. Moreover, locally submitted messages must be sent via SMTP for these functions to work (*sendmail* must be running as user *smmsp* rather than as user *root*).

Example 5-9 shows how you could use `stream_by_domain( )` to offer different policies to different recipient domains. Policies are stored in a Berkeley database file */etc/mail/spampolicy.db* that is generated from a text file */etc/mail/spampolicy* using the standard sendmail `makemap` program. Each line of the text file should contain a domain name, white space, and a policy, which should be either `TAG` (tag spam at SpamAssassin's default level), `TAG`*n* (tag messages with over *n* hits), `BLOCK` (reject spam at SpamAssassin's default level), `BLOCK`*n* (reject messages with over *n* hits), or `IGNORE` (do no spam-checking). *spampolicy.db* must be owned by *defang*.

Example 5-9. Using stream_by_domain()

```
use DB_File;

sub getpolicy {
  # Where do we find the policy db?
  my $policydb = '/etc/mail/spampolicy.db';
  # If a domain isn't listed, what's the default policy?
  my $default_policy = 'TAG';
  my $host = shift;
```

Example 5-9. Using stream_by_domain() (continued)

```
  tie %policy, 'DB_File', $policydb, O_RDONLY, 0640, $DB_HASH;
  my $policy = $policy{"\L$host"};
  untie %policy;
  return defined($policy) ? "\U$policy" : $default_policy;
}

sub filter_begin ( ) {
    if ($SuspiciousCharsInHeaders) {
        md_graphdefang_log('suspicious_chars');
        return action_discard( );
    }

    # Per-domain streaming is turned on here so we get the $Domain var
    # set later on.
    return if stream_by_domain( );
    ...
}

sub filter_end ($) {
    my($entity) = @_;
    send_quarantine_notifications( );

    # No sense doing any extra work
    return if message_rejected( );

    # Spam checks if SpamAssassin is installed
    if ($Features{"SpamAssassin"}) {
        if (-s "./INPUTMSG" < 100*1024) {
          # Spam policy selection, based on $Domain, using a Berkeley db lookup
          my $spampolicy = getpolicy($Domain);
          action_add_header("X-Spam-Policy", "$spampolicy $Domain");
          if ($spampolicy ne "IGNORE") {
            my($hits, $req, $names, $report) = spam_assassin_check( );
            $req = $1 if ($spampolicy =~ /(\d+)/);
            if ($hits >= $req) {
                md_graphdefang_log('spam', $hits, $RelayAddr);
                if ($spampolicy =~ /BLOCK/) {
                  action_bounce("Message rejected by SpamAssassin");
                  return;
                }
                my($score);
                if ($hits < 40) {
                    $score = "*" x int($hits);
                } else {
                    $score = "*" x 40;
                }
                action_change_header("X-Spam-Score", "$hits ($score) $names");
                action_add_part($entity, "text/plain", "-suggest",
                                "$report\n",
                                "SpamAssassinReport.txt", "inline");
            } else {
                action_delete_header("X-Spam-Score");
```

Example 5-9. Using stream_by_domain() (continued)

```
                }
            }
        }
    }
}
```

You could similarly use `stream_by_recipient( )` in an environment where you want to read SpamAssassin user preferences for each recipient from an SQL database. The *Mail::SpamAssassin* object used in *mimedefang-filter* is named `$SASpamTester`. A simple approach is to call the `load_scoreonly_sql( )` method on that object, passing the recipient's email address as an argument, like this:

```
    # @Recipients in mimedefang-filter is an array of recipient emails,
    # but if you're using stream_by_recipient, there should only be a single
    # recipient at this point.
    my $recip = $Recipient[0];
    # If your SQL database uses usernames rather than email addresses, uncomment:
    # $recip =~ s/@.*//;
    $SASpamTester->load_scoreonly_sql($recip);
```

This approach creates a new database connection for each mail message. A more complicated, but more efficient approach would be to set up a database connection in `filter_begin( )` and write SQL queries by hand in `filter_end( )`. On the other hand, using SpamAssassin's own functions, like `load_scoreonly_sql( )`, ensures that your code will be compatible with future SpamAssassin releases that might change the database format.

Although `stream_by_recipient( )` and `stream_by_domain( )` solve an important problem, they do so at a cost in performance. Messages that arrive for multiple recipients (or domains) will have to be split up and reinjected, considerably increasing the overall load on the mail server.

Building a Spam-Checking Gateway

By combining sendmail, MIMEDefang, and SpamAssassin, you can build a complete spam-checking gateway. Such systems are increasingly popular as external mail exchangers, receiving messages from the Internet and relaying them to internal mail servers that don't perform their own spam-checking (either for performance reasons or because they run operating systems that don't provide cost-effective antispam solutions). I assume that users relay outgoing mail through an internal mail server, rather than through the spam-checking gateway. Figure 5-2 illustrates this topology.

The example gateway in this section is based on actual gateways in operation on the Internet. Although I provide complete configuration files for the example, I discuss only those aspects of configuration directly relevant to spam-checking.

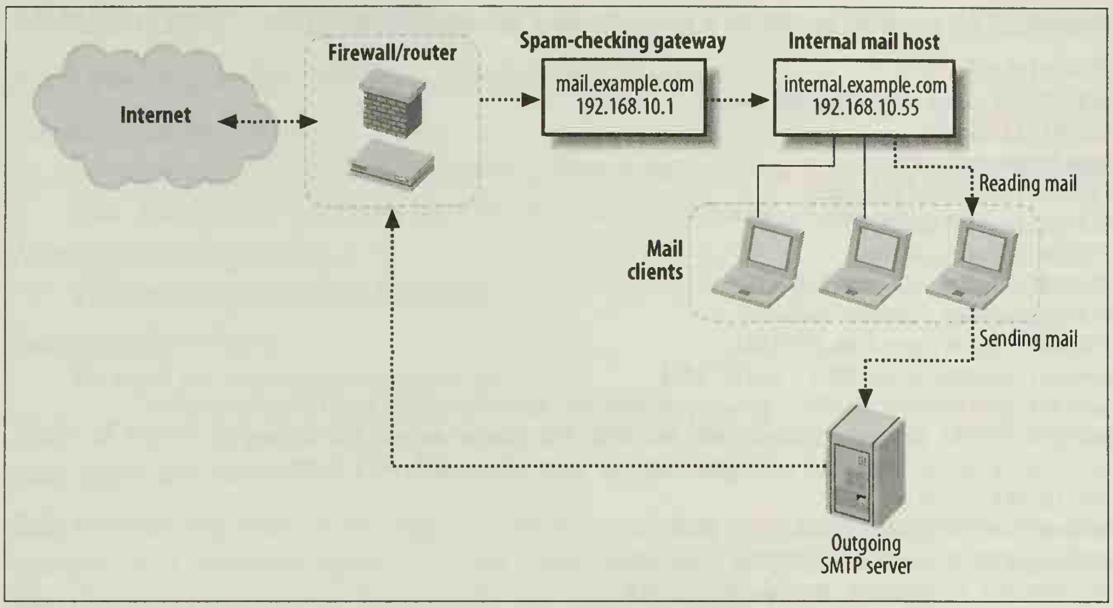


Figure 5-2. Spam-checking gateway topology

sendmail Configuration

In our scenario, the spam-checking gateway should accept messages for our domains, check them for spam, and relay them to an internal mail server. Accordingly, I include the following in our sendmail configuration:

- The *mailertable* feature, which we may use to indicate the internal server to which we'll relay checked messages.
- A one-hour timeout before sending a warning message about delayed delivery, and a seven-day timeout before bouncing messages. If the internal server should fail and need to be replaced, senders will quickly know that their messages have been delayed, but messages won't be bounced unless you can't replace the internal server within a week.
- Several configuration options to limit sendmail's resource usage. We limit sendmail to 60 forked child processes and 10 connections per second. We limit messages to 10Mb and 500 recipients each.
- An `INPUT_MAIL_FILTER` definition for MIMEDefang.

Example 5-10 is the *sendmail.mc* configuration file for the gateway and is used to generate */etc/mail/sendmail.cf*.

Example 5-10. sendmail.mc file for a spam-checking gateway

```
divert(-1)
#
# Spam-checking gateway configuration
#
```

Example 5-10. sendmail.mc file for a spam-checking gateway (continued)

```
divert(0)dnl
VERSIONID(`Spam-checking gateway')
OSTYPE(linux)dnl
DOMAIN(generic)dnl
FEATURE(virtusertable)dnl
FEATURE(mailertable)dnl
FEATURE(access_db)dnl
FEATURE(always_add_domain)dnl
FEATURE(nouucp,`reject')dnl
FEATURE(`relay_based_on_MX')dnl
define(`confDEF_USER_ID',``8:12'')dnl
define(`confPRIVACY_FLAGS',`goaway,noreceipts,restrictmailq,restrictqrun,noetrn'
dnl Since this is for a gateway MX, we keep the queue around for a long
dnl time without bouncing messages, but we warn about delivery delay
dnl rather quickly
define(`confTO_QUEUERETURN',`7d')dnl
define(`confTO_QUEUEWARN_NORMAL',`1h')dnl
dnl Options to prevent denial-of-service
define(`confMAX_DAEMON_CHILDREN',`60')dnl
define(`ConfMAX_MESSAGE_SIZE',`10000000')dnl
define(`confMAX_CONNECTION_RATE_THROTTLE',`10')dnl
define(`confMAX_RCPTS_PER_MESSAGE',`500')dnl
INPUT_MAIL_FILTER(`mimedefang', `S=unix:/var/spool/MIMEDefang/mimedefang.sock, T
MAILER(smtp)dnl
MAILER(local)dnl
MAILER(procmail)dnl
```

Because mail destined for the *example.com* domain should not be delivered locally on the external gateway, do not include *example.com* as one of the gateway's local hostnames in the */etc/mail/local-host-names* (*/etc/mail/sendmail.cw* on some systems) file.

SpamAssassin Configuration

Store the SpamAssassin configuration for a gateway in */etc/mail/sa-mimedefang.cf*. In addition to setting the typical options, it's a wise idea to use the `trusted_networks` (and, in SpamAssassin 3.0, `internal_networks`) directive to define the boundary between trusted and untrusted networks. Example 5-11 shows the *sa-mimedefang.cf* file on a system configured to use a sitewide Bayesian database.

Example 5-11. sa-mimedefang.cf file for a spam-checking gateway

```
required_hits 5

# These are hosts that we control
internal_networks 192.168.10/24

# This is a backup MX that's offsite
trusted_networks 111.222.333.444

bayes_path /var/spool/MD-Bayes/bayes
```

MIMEDefang Configuration

After installing MIMEDefang, set up three directories:

/var/spool/MIMEDefang
: To contain MIMEDefang's working directories, and to hold the socket and pid files. Mount this directory on a RAM disk for increased performance.

/var/spool/MD-Quarantine
: To contain quarantine directories.

/var/spool/MD-Bayes
: To hold the Bayesian database files.

Each of these directories should be owned by the user under which MIMEDefang runs (typically, *defang*).

Edit *mimedefang-filter* to configure it for your gateway. Example 5-12 shows the first portion of a *mimedefang-filter* script corresponding to the example gateway I'm describing in this chapter. Each of the key variables in the file is defined.

Example 5-12. mimedefang-filter configuration for a spam-checking gateway

```
#***********************************************************************
# Set administrator's e-mail address here.  The administrator receives
# quarantine messages and is listed as the contact for site-wide
# MIMEDefang policy.  A good example would be 'defang-admin@mydomain.com'
#***********************************************************************
$AdminAddress = 'postmaster@example.com';
$AdminName = "Example.com Postmaster";

#***********************************************************************
# Set the e-mail address from which MIMEDefang quarantine warnings and
# user notifications appear to come.  A good example would be
# 'mimedefang@mydomain.com'.  Make sure to have an alias for this
# address if you want replies to it to work.
#***********************************************************************
$DaemonAddress = 'mimedefang@example.com';

# Allow SpamAssassin to use network-based tests
$SALocalTestsOnly = 0;
```

Routing Email

Mail from the Internet for *example.com* should be sent to the spam-checking gateway *mail.example.com*. To accomplish that, add a DNS mail exchanger (MX) record for the *example.com* domain that points to *mail.example.com*.

Once received by *mail.example.com*, messages will be spam-checked and should then be relayed to *internal.example.com*. You can accomplish that relaying in one of two ways:

Using DNS

Provide *mail.example.com* with an MX record for *example.com* pointing to *internal.example.com* and having a lower preference value (more preferred) than the *mail.example.com* MX record. This requires that you provide different results to DNS queries from Internet hosts versus queries from *mail.example.com*. Do so by running so-called split DNS, or by using BIND 9's `view` directives. Internet hosts should see only the *mail.example.com* MX record, but *mail.example.com* (and probably all internal hosts and clients) should see the *internal.example.com* MX record.

Using mailertable

Add `FEATURE(`mailertable')` to the *sendmail.mc* file, and create a */etc/mail/mailertable* file that instructs sendmail where to forward messages destined for *example.com*:

```
example.com    esmtp:internal.example.com
```

or:

```
example.com    esmtp:[192.168.10.55]
```

After editing *mailertable*, be sure to use `makemap` to build the *mailertable.db* database from the *mailertable* file.

Internal Server Configuration

Once the external mail gateway is in place, you can configure the internal mail server to accept only SMTP connections from the gateway (for incoming Internet mail). If you don't have a separate server for outgoing mail, the internal mail server should also accept SMTP connections from hosts on the internal network. This restriction is usually enforced by limiting access to TCP port 25 using a host-based firewall or a packet-filtering router.

Testing

You should now have a complete, spam-checking gateway. Test the gateway by sending spam and non-spam messages to *user@example.com*. Messages should arrive at *internal.example.com* with *Received* headers that show that they were first received by *mail.example.com* and then by *internal.example.com*, and *X-Scanned-By* headers that mention MIMEDefang. Spam messages should have *X-Spam-Status* headers added as well.

CHAPTER 6

Integrating SpamAssassin with Postfix

Postfix is a mail transport agent written by security researcher Wietse Venema. Not surprisingly, Postfix is designed from the ground up to be a highly secure system. It consists of several components, each of which runs with least privilege and none of which trust data from the other without validating it themselves. Despite the extensive security emphasis in the system's architecture, Postfix is capable of very good performance in normal conditions; because of architectural decisions, it is also fault tolerant and capable of good performance under adverse conditions such as resource starvation. It has become a popular replacement for sendmail because it provides a compatible command-line interface.

This chapter explains how to integrate SpamAssassin into a Postfix-based mail server to perform spam-checking for local recipients or to create a spam-checking mail gateway.

Postfix is a complex piece of software, and, like most MTAs, offers scores of configuration options. This chapter assumes that you are running Postfix 1.1 or 2.x (recommended) and does not cover how to securely install, configure, or operate Postfix itself. For that information, see the Postfix documentation and the book *Postfix: The Definitive Guide* by Kyle D. Dent (O'Reilly).

Postfix Architecture

Several different Postfix components play roles in receiving messages from the Internet. The `master` daemon is responsible for the coordination of the components. Messages from the Internet typically enter the mail server via the `smtpd` daemon, which listens on port 25 and conducts the SMTP transaction with the remote sender. `smtpd` passes each message to the `cleanup` daemon, which performs sanity checks, fixes missing headers, and (with the help of the trivial-rewrite program) rewrites addresses. `cleanup` then deposits each message in the incoming mail queue and alerts the `qmgr` daemon. `qmgr` moves messages from the incoming queue to the active queue, and then calls `local` for delivery to local recipients (or `smtp` for relaying to remote recipients by SMTP). Figure 6-1 illustrates the flow of email through Postfix components.

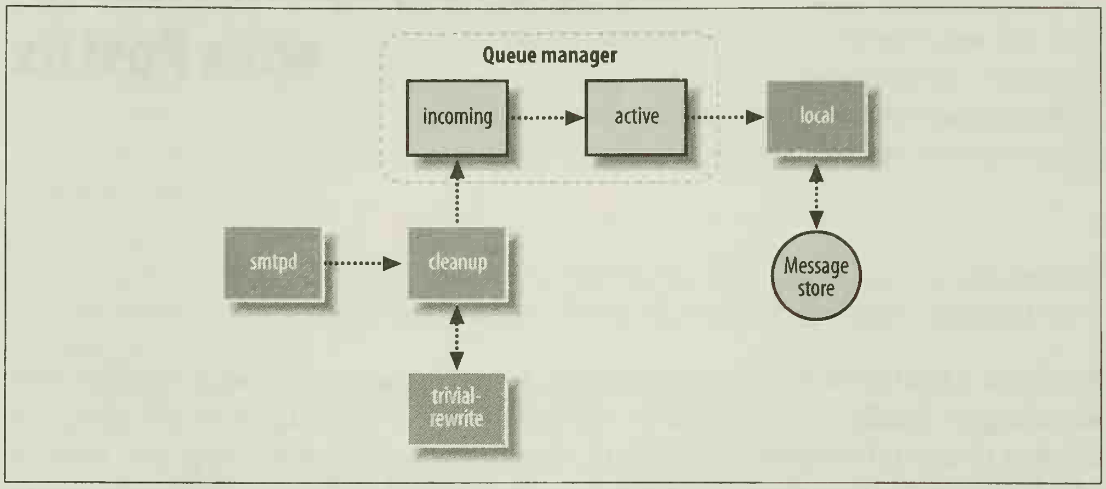


Figure 6-1. The Postfix architecture during message receipt

Most systems keep Postfix's configuration files in */etc/postfix*. The most important files are *main.cf*, which contains nearly all of the configuration directives for Postfix, and *master.cf*, which configures the `master` daemon and determines how the various Postfix components will be run. After you make changes to either of these files, you should issue the `postfix reload` command to cause Postfix to re-read the files.

Spam-Checking During Local Delivery

The easiest way to add SpamAssassin to a Postfix system is to configure Postfix to use procmail as its local delivery agent, rather than the Postfix `local` program. Then add a procmail recipe for spam-tagging to */etc/procmailrc*.

The advantages of this approach are:

- It's very easy to set up.
- You can run `spamd`, and the procmail recipe can use `spamc` for faster spam-checking.
- User preference files, autowhitelists, and Bayesian databases can be used.

However, Postfix runs a local delivery agent only for email destined for a local recipient. You cannot create a spam-checking gateway with this approach.

Instructions for configuring procmail for spam-checking can be found in Example 2-6 in Chapter 2. To configure Postfix to use procmail as the local delivery agent, use the `mailbox_command` directive in *main.cf*:

```
mailbox_command = procmail -a "$EXTENSION"
```

If you configure Postfix to use procmail as the local delivery agent, you must be sure that you have an alias for *root* in your *aliases* file (typically in */etc* or */etx/postfix*). The alias should point to another local user. Without such an alias, Postfix may be unable to deliver mail to *root* using procmail.

Spam-Checking All Incoming Mail

If you want to set up a spam-checking gateway for all recipients, local or not, you need a way to perform spam-checking as mail is received, before final delivery. Postfix provides a general-purpose filtering directive called `content_filter`.

The `content_filter` directive specifies a mail transport that Postfix will invoke after receiving a message. The mail transport hands the message to a filtering program. The filter checks the message and then either refuses it (which will cause Postfix to generate a bounce message), discards it, or reinjects the (possibly modified) message into Postfix for further delivery. Messages that pass the filter are reinjected so that Postfix can operate on them almost as if they were new messages; this allows Postfix to behave properly if the content filter rewrites message headers. You can use the `content_filter` directive in *main.cf*, in which case the directive will be used by both `smtpd` (for email received via SMTP) and `pickup` (for email received locally). You can also specify `content_filter` as an invocation option to `smtpd` or `pickup`, which is useful when you only want to filter email received from outside (or inside) the system.

Content filters can be programs that are invoked for each message. They read a message on standard input and reinject filtered messages via the `sendmail` program. They can also be daemons that listen on a local TCP port, receiving messages via SMTP or LMTP (Local Mail Transfer Protocol), and reinjecting filtered messages via SMTP by communicating with a second instance of `smtpd` listening on a local port.

Don't confuse Postfix's `sendmail` program with sendmail. sendmail is an entirely different MTA that also uses an executable named `sendmail` to perform nearly all of its functions. Postfix's `sendmail` program is much more limited but is designed to serve as a replacement for sendmail's to facilitate converting systems from sendmail to Postfix.

SpamAssassin itself is not suitable for use as a content filter, because it doesn't know how to reinject a tagged message. However, SpamAssassin can be invoked by a content filter in several ways.

Using a Program as a Content Filter

The simplest content filters are programs that accept messages on standard input, perform spam-checking, and either exit with an error status code or reinject the message to Postfix. When you use a program as a content filter, you do not need to run any additional daemons—Postfix invokes the program for each message. If your system receives a lot of mail, you are likely to get better performance by using a daemonized content filter, which is discussed in the next section.

To use a program as a content filter requires a series of steps:

1. Create a new system user that Postfix will use to run the filter program or shell script. SpamAssassin will use this user's SpamAssassin preferences (in the *.spamassassin/user_prefs* file in their home directory) when checking messages that have multiple recipients. In the following steps, assume the user is named *spamfilt*.
2. Create a program (or shell script) that can accept an email message on standard input, perform filtering, and pass the modified message to `sendmail`'s standard input. The filter should also return an appropriate status code, usually the exit code from `sendmail`, which Postfix will understand. Your program (or shell script) should expect to receive command-line arguments consisting of the sender's email address and a space-separated list of recipient email addresses.

   Here's an example of a filter script called `pf-spamfilt` that calls SpamAssassin using `spamc`. If the message being checked has only a single recipient, `spamc`'s `-u` option is used to load the per-user preferences. When the message has multiple recipients, the script runs `spamc` without `-u`, and, because the script will be running as the *spamfilt* user, `spamc` will use *spamfilt*'s preferences file.

   ```
   #!/bin/sh
   #
   # pf-spamfilt: An example spam filtering script for postfix
   #
   sender=$1
   shift
   recip="$@"
   if [ "$#" -eq 1 ]; then
     /usr/bin/spamc -u $recip
   else
     /usr/bin/spamc
   fi | /usr/sbin/sendmail -i -f $sender -- $recip
   exit $?
   ```

   Because this filter uses the `spamc` client, you must be running a `spamd` server. Save the filter somewhere publicly accessible (e.g., */usr/local/bin/pf-spamfilt*) and set its permissions to allow anyone to read and execute it.
3. Define a new mail transport in *master.cf* that invokes the filter you created in step 2. The following example shows how you add a transport called `spamcheck`, defined as a Unix service. By defining the transport as shown, you specify that the mail transport will use Postfix's `pipe` command to run */usr/local/bin/pf-spamfilt* as user *spamfilt*, and will pass the email address of the sender and the email addresses of recipients as command-line arguments to *pf-spamfilt*. The flag

argument includes the R flag (add a *Return-Path* header) and the q flag (quote the sender and recipient addresses for use in the command line).

```
# ==========================================================================
# service type  private unpriv  chroot  wakeup  maxproc command + args
#               (yes)   (yes)   (yes)   (never) (50)
# ==========================================================================
spamcheck  unix  -       n       n       -       -       pipe
  flags=Rq user=spamfilt argv=/usr/local/bin/pf-spamfilt ${sender} ${recipient}
```

4. Direct Postfix to use the new mail transport as a content filter for the smtpd daemon. Replace this line in *master.cf*:

   ```
   smtp      inet  n       -       -       -       -       smtpd
   ```

   with these two lines:

   ```
   smtp      inet  n       -       -       -       -       smtpd
     -o content_filter=spamcheck:
   ```

5. If you always want to use per-user preferences, instruct Postfix to call the spamcheck transport with only a single recipient per message by adding this line to *main.cf*:

   ```
   spamcheck_destination_recipient_limit = 1
   ```

6. Run postfix reload to re-read the configuration files. Test the system by sending an email from the Internet and see whether SpamAssassin is called to check the message.

Using a Daemon as a Content Filter

Although it's more complicated to run a daemonized content filter, most larger sites will want to do so in order to avoid the overhead associated with starting the content filter for each email and running sendmail for reinjection. In the daemonized approach, the filter listens on a TCP port bound to the loopback address (127.0.0.1). On receiving a message from the Internet, Postfix connects to the filter daemon and relays the message using the SMTP or LMTP protocol.

The daemon can reject the message during the SMTP/LMTP transaction, which will cause Postfix to bounce the message, or the daemon can accept the message, modify it, and reinject it by SMTP. To prevent mail loops, Postfix must run a second smtpd daemon, bound to another TCP port on the loopback address. The second smtpd is configured to accept messages without rerunning the filter (or performing the checks that would normally be performed on a message received from the Internet).

To use a daemon as a content filter requires five steps:

1. Install a daemon that performs content-filtering in the fashion that Postfix expects. The section later in this chapter titled "Building a Spam-Checking G ateway" provides an example. Typically, you will need to know or configure:
   - The port on which the daemon accepts incoming messages to check (e.g., 10024).

    - The protocol (SMTP or LMTP) by which the daemon expects to receive an incoming message.
    - The port to which the daemon will connect to reinject a message to Postfix (e.g., 10025).
    - The user that will run the daemon. Ideally, you should run daemons under a single-purpose, non-*root* user.

2. Define a new mail transport in *master.cf* that sends mail to the daemon. In the following example, the transport is called `spamcheck` and is defined as a Unix service that will use Postfix's `smtp` command. You can use the `disable_dns_lookups` option to save overhead, as you know that the transport will be configured to relay mail to your loopback IP address, so the daemon will never need to perform a DNS MX lookup. The example uses the maxproc feature in *master.cf* to limit the number of messages that can use this mail transport at one time to two.

```
# ==========================================================================
# service type  private unpriv  chroot  wakeup  maxproc command + args
#               (yes)   (yes)   (yes)   (never) (50)
# ==========================================================================
spamcheck  unix   -       -       n       -       2       smtp
  -o disable_dns_lookups=yes
```

> If you are using Postfix 2.0 or later, you can define the `spamcheck` transport to use Postfix's `lmtp` command instead of `smtp`. The LMTP protocol has some advantages over SMTP—notably, LMTP servers (including `amavisd`) can return individual accept/refuse codes for each message recipient during an LMTP transaction. Postfix's `lmtp` client can also cache connections to an LMTP server for greater performance. Bugs in the `lmtp` client existed in Postfix versions earlier than 2.0 so using `smtp` is recommended with these versions.

3. Define a new mail transport that receives mail from the daemon in *master.cf*. This transport will use Postfix's `smtpd` daemon and is defined by the IP address and port number on which it will listen (127.0.0.1 and 10025, respectively). `smtpd` is an `inet` service, and many option parameters are provided to prevent further filtering and to restrict access to this mail transport to the local host only. Here is an example of such a definition in *master.cf*:

```
# ==========================================================================
# service type  private unpriv  chroot  wakeup  maxproc command + args
#               (yes)   (yes)   (yes)   (never) (50)
# ==========================================================================
127.0.0.1:10025  inet   n      -       n       -       -       smtpd
    -o content_filter=
    -o myhostname=localhost.yourdomain
    -o local_recipient_maps=
    -o relay_recipient_maps=
    -o smtpd_restriction_classes=
    -o smtpd_client_restrictions=
          -o smtpd_helo_restrictions=
          -o smtpd_sender_restrictions=
          -o smtpd_recipient_restrictions=permit_mynetworks,reject
          -o mynetworks=127.0.0.0/8
          -o strict_rfc821_envelopes=yes
          -o smtpd_error_sleep_time=0
          -o smtpd_soft_error_limit=1001
          -o smtpd_hard_error_limit=1000
```

> The `-o myhostname=localhost.`*`yourdomain`* option is important if the content filter issues the SMTP `HELO` command with the same hostname that it originally received from Postfix. If Postfix sees a `HELO` from itself, it rejects the connection to avoid a mail loop. By telling the new `smtpd` that its hostname is something else, you prevent this problem.

4. Direct Postfix to the use the daemon's mail transport as a content filter for mail received by the primary `smtpd` daemon. Replace this line in *master.cf*:

   ```
   smtp      inet  n       -       -       -       -       smtpd
   ```

   with these two lines:

   ```
   smtp      inet  n       -       -       -       2       smtpd
     -o content_filter=spamcheck:[127.0.0.1]:10024
   ```

   The primary `smtpd` daemon will filter incoming messages by passing them to the `spamcheck` mail transport that is listening on port 10024 of the loopback address 127.0.0.1.

5. Run `postfix reload` to re-read the configuration files. Test the system by sending email from the Internet.

Figure 6-2 illustrates this configuration.

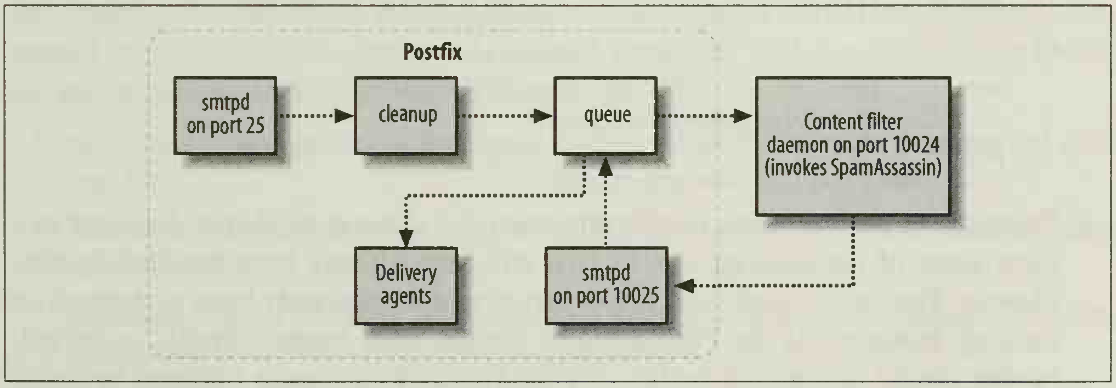


Figure 6-2. Postfix with a daemonized content filter

Filtering Before Address-Rewriting

The Postfix queue manager invokes content filters once it has queued a message. A potential problem with the simple content-filtering approaches outlined earlier is that the messages to be filtered have passed through the cleanup service on their way to the queue, and cleanup performs virtual address lookups and address canonicalization—that is, cleanup may rewrite addresses in message headers. Accordingly, the message that Postfix sends to the content filter (and thus to SpamAssassin) to check is not exactly the same as the message that Postfix received. The changes to addresses may rob SpamAssassin's rules (or the Bayesian classifier) of useful determinants of spam.

If you are running Postfix 2.0 or later, you can fix this problem by setting up a separate, pre-cleanup service that does not perform address canonicalization. Messages received by Postfix's smtpd and pickup can be routed through the pre-cleanup and then to the queue. Filter-checked messages received by the second smtpd instance can then be routed through the standard cleanup service for address-rewriting before returning to the queue for further delivery processing.

To use a two-cleanup design, set up a daemonized filter configuration as described in the previous section and then make the following configuration changes:

1. Add a new pre-cleanup service to */etc/postfix/master.cf* that calls the cleanup daemon but turns off address canonicalization:

```
pre-cleanup         unix  n      -      n      -      0      cleanup
        -o canonical_maps=
        -o sender_canonical_maps=
        -o recipient_canonical_maps=
        -o masquerade_domains=
        -o virtual_alias_maps=
```

2. Configure smtpd and pickup to use the pre-cleanup service in */etc/postfix/master.cf* by changing their entries from

```
smtp     inet  n      -      -      -      -      smtpd
pickup   fifo  n      -      -      60     1      pickup
```

to

```
smtp     inet  n      -      -      -      -      smtpd
  -o cleanup_service_name=pre-cleanup
pickup   fifo  n      -      -      60     1      pickup
  -o cleanup_service_name=pre-cleanup
```

3. To improve performance, modify the entry for cleanup so that it does not perform some of the message checks that will have already been handled by pre-cleanup. You can turn off any checks that would have already been performed on message headers (via the Postfix header_checks, mime_header_checks, or nested_header_checks options) or bodies (via the Postfix body_checks options) by defining each option to be empty:

```
cleanup            unix  n      -      n      -      0      cleanup
    -o header_checks=
    -o mime_header_checks=
    -o nested_header_checks=
    -o body_checks=
```

Figure 6-3 illustrates this configuration.

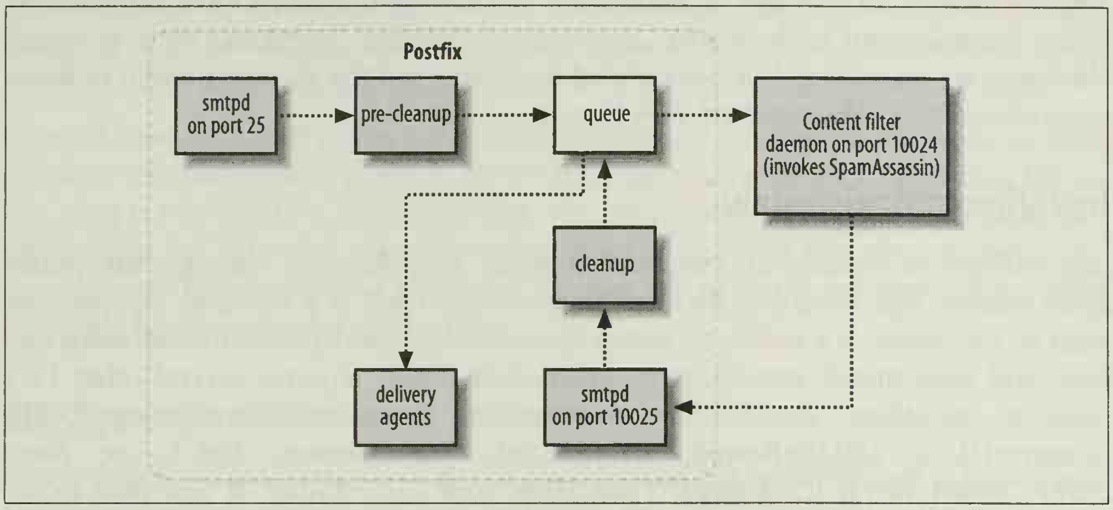


Figure 6-3. Postfix with a daemonized content filter and two cleanup services

Building a Spam-Checking Gateway

Several content-filtering daemons that call SpamAssassin are available for Postfix. This section provides a complete sample installation of amavisd-new, a particularly efficient filter that supports both spam-checking and virus-checking. amavisd-new is written in Perl and available at *http://www.ijs.si/software/amavisd/*. The version used in this chapter's example is 20030616-p9, which supports both SpamAssassin 2.63 and SpamAssassin 3.0.

amavisd-new is based on amavis, another virus-scanning package that is also actively developed and widely used. Although amavisd-new's most important program is also named `amavisd`, amavisd-new has developed separately and is a significantly different package. Some of amavisd-new's features include:

- avisd-new was specifically developed and tested for Postfix as a daemonized content filter.
- Messages can be rejected based on MIME type or extensions of attached filenames.
- Messages can be checked with multiple virus scanners, and messages carrying viruses can be refused, discarded, or quarantined.
- SpamAssassin can be invoked on a message, and spam can be refused, discarded, quarantined, or tagged.
- Per-user configuration of amavisd-new is possible through an SQL or LDAP database.

The rest of this chapter details the installation, configuration, and operation of amavisd-new as an example of a full-scale, daemonized, content filter approach to using SpamAssassin with Postfix. amavisd-new's other functions, such as virus-checking, are mentioned but not covered in detail; read the documentation to learn more about these other amavisd-new features.

Installing amavisd-new

amavisd-new is written in Perl, and invokes SpamAssassin through the *Mail::SpamAssassin* Perl modules. Because amavisd-new itself is a daemon, you do not need to run `spamd`. It's easiest to install SpamAssassin (and your antivirus software) first, and then install amavisd-new. amavisd-new also requires several other Perl modules, including: *Archive::Tar*, *Archive::Zip*, *Compress::Zlib*, *Convert::TNEF*, *Convert::UUlib*, *MIME::Base64*, *MIME::Tools*, *Mail::Internet*, *Net::Server*, *Net::SMTP*, *Digest::MD5*, *IO::Stringy*, *Time::Hires*, and *Unix::Syslog*. If you plan to do per-user configuration of amavisd-new through SQL or LDAP, you'll need appropriate Perl modules for database access (*DBI* and a *DBD::* module for SQL, or *Net::LDAP* for LDAP). You can install most of these Perl modules using CPAN as described in Chapter 2.

The standard version of *MIME::Tools* 5.411a has bugs. Install *MIME::Tools* 6 or later from *http://search.cpan.org/dist/MIME-tools*.

Begin the install process by creating a new user account and group for running amavisd-new; the usual name for both the user and group is *amavis*. This user will own amavisd-new's files, and the user (or group) must have access to SpamAssassin's configuration and database files as well. The user's home directory is traditionally */var/amavis*, but you can create it anywhere that fits your system's needs.

amavisd-new uses several important directories. It keeps two files in its home directory, one containing its current process ID, and the other used for locking. It uses a working directory for unpacking email messages and scanning them; by default, this is the home directory or the *tmp* subdirectory of the home directory. For optimal performance, this directory should be on a fast disk—even a RAM disk if your operating system supports it and you have enough memory to spare. amavisd-new stores quarantined email messages in */var/virusmails* by default, but you can select any directory for this purpose. Speed is not so critical with this directory, and it should never be located on a RAM disk because you will often want to be sure that you can access quarantined files. If you plan to physically locate these directories somewhere unusual (e.g., to mount new disk partitions or a RAM disk as */var/amavis/tmp*), you should do so before you install amavisd-new. The directories should be owned by user and group *amavis* and should not be world-readable or world-searchable.

Next, download the amavisd-new source code from *http://www.ijs.si/software/amavisd/* and unpack it. As *root*, copy the `amavisd` script to a suitable directory for executable daemons (e.g., */usr/bin*, */usr/local/sbin*, etc.), `chown` it to *root*, and use `chmod` to set its permissions to 0755 (readable and executable by all users, writable only by *root*).

Copy the *amavisd.conf* file to a suitable directory for configuration files (e.g., */etc*, */etc/amavis*, */usr/local/etc*, etc.). By default, `amavisd` expects to find this file in */etc*, and if you locate it anywhere else, you will have to add an extra command-line option (`-c` *`filename`*) when invoking `amavisd` to tell it the new location. The *amavisd.conf* file should also be owned by *root* and should have permissions 0644 (readable by all, writable only by *root*).

Configuring amavisd-new

amavisd-new is configured through the *amavisd.conf* file. *amavisd.conf* is parsed as a Perl script and can contain any legal Perl code. Because it is parsed as Perl, you must escape any at sign (@), question mark ($), or backslash (\) characters that appear in double-quoted strings by prepending a backslash. For example:

```
$some_email = "sample\@example.com";
```

Email addresses must be specified without surrounding brackets and without RFC 2821 quoting.

Edit *amavisd.conf* to set the (many) available configuration options to control `amavisd`. The file is organized in logical sections; the most important options are in Section I, but you'll need to read through the entire file to customize the system completely. The following sections explain commonly modified portions of the configuration file in the order that you'll encounter them.

Essential options

Example 6-1 shows the first portion of the configuration file and the settings of the essential options. Set `$MYHOME` to the *amavis* user's home directory. Set `$mydomain` to your domain name. Set `$daemon_user` and `$daemon_group` to name of the *amavis* user and group. Set `$TEMPBASE` to the directory to use for unpacking messages; for improved performance, this directory should be a mounted RAM disk.

Example 6-1. Essential settings in amavisd.conf

```
# Section I - Essential daemon and MTA settings
#

# $MYHOME serves as a quick default for some other configuration settings.
# More refined control is available with each individual setting further down.
# $MYHOME is not used directly by the program. No trailing slash!
$MYHOME = '/var/amavis';     # (default is '/var/amavis')

# $mydomain serves as a quick default for some other configuration settings.
```

Example 6-1. Essential settings in amavisd.conf (continued)

```
# More refined control is available with each individual setting further down.
# $mydomain is never used directly by the program.
$mydomain = 'example.com';   # (no useful default)

# Set the user and group to which the daemon will change if started as root
# (otherwise just keeps the UID unchanged, and these settings have no effect):
$daemon_user  = 'amavis';    # (no default;  customary: vscan or amavis)
$daemon_group = 'amavis';    # (no default;  customary: vscan or amavis)

# Runtime working directory (cwd), and a place where
# temporary directories for unpacking mail are created.
# (no trailing slash, may be a scratch file system)
#$TEMPBASE = $MYHOME;        # (must be set if other config vars use is)
$TEMPBASE = "$MYHOME/tmp";   # prefer to keep home dir /var/amavis clean?
```

MTA options

Example 6-2 shows the settings of the MTA options. Set `$forward_method` to the method you will use to reinject checked mail to the MTA. For Postfix, this method should be of the form `smtp:`*`ipaddress`*`:`*`portnumber`*, where *`ipaddress`* is the IP address of the Postfix system (usually 127.0.0.1) and *`portnumber`* is the TCP port number on which the second `smtpd` instance is running. Because amavisd-new was designed with Postfix in mind, you may not need to change this section at all.

Example 6-2. MTA options in amavisd.conf

```
# MTA SETTINGS, UNCOMMENT AS APPROPRIATE,
# both $forward_method and $notify_method default to 'smtp:127.0.0.1:10025'

# POSTFIX, or SENDMAIL in dual-MTA setup, or EXIM V4
# (set host and port number as required; host can be specified
# as IP address or DNS name (A or CNAME, but MX is ignored)
$forward_method = 'smtp:127.0.0.1:10025';  # where to forward checked mail
$notify_method = $forward_method;          # where to submit notifications
```

Daemon process options

Example 6-3 shows the daemon process settings. The most important setting is `$max_servers`, which you should set to the same number of `smtp` processes you have configured Postfix to use concurrently to send messages to amavisd-new.

Example 6-3. Daemon process settings in amavisd.conf

```
# Net::Server pre-forking settings
# You may want $max_servers to match the width of your MTA pipe
# feeding amavisd, e.g. with Postfix the 'Max procs' field in the
# master.cf file, like the '2' in the:  smtp-amavis unix - - n - 2 smtp
#
$max_servers  =  2;   # number of pre-forked children          (default 2)
```

Distinguishing local domains

amavisd-new distinguishes local domains from remote domains. Recipients at local domains can take advantage of several per-user features that are not directly available to remote recipients, including local customization of SpamAssassin settings. Example 6-4 shows that part of *amavisd.conf* that bears on per-user customization.

You can provide your local domain information in several ways. You can set the `@local_domains_acl` array to a list of domain names that should be considered local. You can set the `%local_domains` hash instead, providing local domain names as keys and 1 as their values, or use the `read_hash` function to read in a list of local domain names from an external file. Finally, you can define local domain names by invoking the `new_RE` function with a regular expression that matches the local domain names and assigning the result to `$local_domains_re`. No matter which method you use, adding a period (.) to the beginning of a domain name means that the domain and any subdomains should all be considered local.

Example 6-4 shows this section of the configuration file, using the `@local_domains_acl` variable to define local domains.

Example 6-4. Setting local domains in amavisd.conf

```
# Lookup list of local domains (see README.lookups for syntax details)
#
# NOTE:
#   For backwards compatibility the variable names @local_domains (old) and
#   @local_domains_acl (new) are synonyms. For consistency with other lookups
#   the name @local_domains_acl is now preferred. It also makes it more
#   obviously distinct from the new %local_domains hash lookup table.
#
# local_domains* lookup tables are used in deciding whether a recipient
# is local or not, or in other words, if the message is outgoing or not.
# This affects inserting spam-related headers for local recipients,
# limiting recipient virus notifications (if enabled) to local recipients,
# in deciding if address extension may be appended, and in SQL lookups
# for non-fqdn addresses. Set it up correctly if you need features
# that rely on this setting (or just leave empty otherwise).
#
# With Postfix (2.0) a quick reminder on what local domains normally are:
# a union of domains specified in: $mydestination, $virtual_alias_domains,
# $virtual_mailbox_domains, and $relay_domains.
#
#@local_domains_acl = ( ".$mydomain" );  # $mydomain and its subdomains
# @local_domains_acl = qw();  # default is empty, no recipient treated as local
# @local_domains_acl = qw( .example.com );
# @local_domains_acl = qw( .example.com !host.sub.example.net .sub.example.net );
# @local_domains_acl = ( ".$mydomain", '.example.com', 'sub.example.net' );
@local_domains_acl = qw/
example.com
example.net
example.org
/;
```

Example 6-4. Setting local domains in amavisd.conf (continued)

```
# or alternatively(A), using a Perl hash lookup table, which may be assigned
# directly, or read from a file, one domain per line; comments and empty lines
# are ignored, a dot before a domain name implies its subdomains:
#
#read_hash(\%local_domains, '/var/amavis/local_domains');

#or alternatively(B), using a list of regular expressions:
# $local_domains_re = new_RE( qr'[@.]example\.com$'i );
```

Postfix-specific options

Section II of *amavsid.conf* specifies options that differ by MTA and is shown in Example 6-5. Because amavisd-new was designed with Postfix in mind, you need to modify relatively few options. Set the `$inet_socket_port` variable to the TCP port number on which `amavisd` should listen for SMTP connections from Postfix. To prevent this port from being accessed by remote hosts, set `$inet_socket_bind` to `'127.0.0.1'`, which will cause `amavisd` to listen only on the loopback interface and not on other network interfaces. If you want to allow access by a set of remote hosts (if, for example, you want to run `amavisd` on a different host than your Postfix MTA), don't set `$inet_socket_bind` but do set `@inet_acl` to a list of IP addresses for hosts that should be permitted to connect. This list is checked in order; the first match wins. You may specify these IP addresses as single addresses or as CIDR-style *`address/netmask`* (e.g., `192.168.1/255.255.255.0`) or *`address/bits`* (e.g., `192.168.1/24`) ranges.[*] You may prepend an IP address with an exclamation point (`!`) to disallow connections from that address, even if a larger range that contains the address is permitted (e.g., `!192.168.0/24 192.168/16` to allow all 192.168.*.* addresses except 192.168.0.* addresses).

Example 6-5. Postfix-specific options in amavisd.conf

```
# SMTP SERVER (INPUT) PROTOCOL SETTINGS (e.g. with Postfix, Exim v4, ...)
#   (used when MTA is configured to pass mail to amavisd via SMTP or LMTP)
$inet_socket_port = 10024;        # accept SMTP on this local TCP port
                                  # (default is undef, i.e. disabled)
# multiple ports may be provided: $inet_socket_port = [10024, 10026, 10028];

# SMTP SERVER (INPUT) access control
# - do not allow free access to the amavisd SMTP port !!!
#
# when MTA is at the same host, use the following (one or the other or both):
$inet_socket_bind = '127.0.0.1';  # limit socket bind to loopback interface
                                  # (default is '127.0.0.1')
@inet_acl = qw( 127.0.0.1 );      # allow SMTP access only from localhost IP
                                  # (default is qw( 127.0.0.1 ) )
```

[*] "CIDR" stands for Classless Interdomain Routing.

Logging options

Section III of *amavisd.conf* deals with logging and is shown in Example 6-6. amavisd can log using *syslog*, or it can log to a file. Set $DO_SYSLOG to 1 to instruct amavisd to use *syslog* for logging; you can change the *syslog* facility and priority using the $SYSLOG_LEVEL variable. Set $DO_SYSLOG to 0 to instruct amavisd to log to a file; set $LOGFILE to specify the filename. The log file must be in a directory the *amavis* user can write to.

The $log_level variable controls the amount of detail that amavisd logs. A log level of 0 results in minimal logging; a log level of 5 produces highly verbose logging.

Example 6-6. Logging options

```
# Section III - Logging
#

# true (e.g. 1) => syslog;  false (e.g. 0) => logging to file
$DO_SYSLOG = 1;                   # (defaults to false)
#$SYSLOG_LEVEL = 'user.info';     # (defaults to 'mail.info')

# Log file (if not using syslog)
$LOGFILE = "$MYHOME/amavis.log";  # (defaults to empty, no log)

#NOTE: levels are not strictly observed and are somewhat arbitrary
# 0: startup/exit/failure messages, viruses detected
# 1: args passed from client, some more interesting messages
# 2: virus scanner output, timing
# 3: server, client
# 4: decompose parts
# 5: more debug details
$log_level = 1;  # (defaults to 0)
```

Spam-handling options

Most of Section IV of *amavisd.conf* focuses on detailed configuration of how amavisd will handle detected viruses and spam. Only those options related to spam handling are discussed in detail here.

When amavisd detects a spam email, it logs a message to its log file by default. It can also quarantine the email and/or notify an administrator. It can then generate a bounce message to the sender. Finally, it can either accept and deliver the message, or discard the message. Many different configuration variables are involved in these decisions. Unfortunately, the order of the variables in the file is largely the reverse of the order in which they are checked during the spam-handling process.

Enable a spam quarantine by setting the following two variables:

$QUARANTINEDIR
: Set this variable to the directory or mailbox file in which to store the quarantined messages.

`$spam_quarantine_method`
: Set this variable to "`local:spam-%b-%i-%n`", to specify the filename format for quarantined spam messages. In that format, `%b` expands to a digest of the message body, `%i` expands to the date and time, and `%n` expands to the `amavisd` message identifier.

To control the spam quarantine on a per-recipient basis, set the `$spam_quarantine_to` variable to a reference to a hash, keyed by the recipient's address, like this:

```
$local_delivery_aliases{'sam-spam'} = '/home/sam/mail/spam';

$spam_quarantine_to =
  { 'example.net' => undef
    'jane@example.com' => 'spam@jane.example.com',
    'sam@example.com' => 'sam-spam',
    'example.com' => 'spam-quarantine',
  };
```

If the hash value is undefined or empty, spam is not quarantined. In this example, spam sent to *example.net* will not be quarantined at all. If the hash value contains an asterisk (`@`), spam will be forwarded. Spam sent to *jane@example.com* will be forwarded to *spam@jane.example.com*. Otherwise, the hash value is looked up in the `%local_delivery_aliases` hash, and the spam is quarantined in the file or directory returned from that lookup. If the lookup fails, `amavisd` logs a warning and doesn't quarantine the message. Several default local delivery aliases are defined in `amavisd`, including `spam-quarantine`, which quarantines a message in `$QUARANTINEDIR`. In the preceding example, spam to *sam@example.com* will be quarantined in the */home/sam/mail/spam* mailbox (or mail directory), and other spam to *example.com* will be quarantined in the default directory.

You can also write your `$spam_quarantine_to` policies with regular expressions:

```
$spam_quarantine_to = new_RE(
  [qr/^sam@example\.com$/i => 'sam-spam'],
  [qr/^jane@example\.com$/i => 'spam@jane.example.com'],
  [qr/@example\.com$/i => 'spam-quarantine'],
  [qr/@example\.net$/i => undef]
);
```

Because regular expressions are matched in the order that you list them, you must put the most specific matches first (`/^sam@example\.com/` before `/@example\.com/`). Because regular expression matches are case sensitive, you should generally include the `i` (case-insensitive) modifier to the `qr//` operator.

Spam to recipients that don't match any entry in `$spam_quarantine_to` will not be quarantined, so if you want to quarantine all spam by default, you should either provide a rule for each domain you receive mail for, or use the regular expression approach and include a rule for the regular expression `qr/.*/` at the end.

amavisd-new is smart about per-recipient policies like `$spam_quarantine_to`. If some message recipients choose to quarantine spam and some do not, amavisd-new will honor those preferences. If multiple recipients choose the same quarantine destination, a message sent to two or more of those recipients is written only once to the quarantine destination .

You can also make quarantine decisions based on a spam's sender in an analogous way using `$spam_quarantine_bysender_to`, but this alternative is rarely useful, as spammers often falsify their sending addresses or use throwaway accounts.

To notify an administrator when spam is received, set `$spam_admin` to the address of the administrator. These notifications are disabled by default. Consider carefully before setting `$spam_admin` to the email address of a real person; given the amount of spam on the Internet today, it's easy to get hundreds of notifications or more, and difficult to know what to do about them. An alternative that might be useful for service providers is to set `$spam_admin` to a reference to a hash based on the spam sender's address, in order to detect outgoing spam from customers. For example, to notify the security staff about spam being sent from the *example.com* domain but nowhere else, use:

```
$spam_admin =
  { '.example.com' => 'security@example.com',
    '.' => undef
  };
```

The `$final_spam_destiny` variable controls the final disposition of spam recognized by `amavisd`. Although this variable appears first in this section of the configuration file, it is consulted last during spam-handling. When using amavisd-new with Postfix, there are three useful settings for `$final_spam_destiny`:

- Set `$final_spam_destiny` to `D_PASS` to accept and deliver all spam. Use this strategy when your goal is simply to tag spam and let clients do their own filtering. If you set `$warnspamsender` to 1, you will also generate a bounce message to the sender. I don't recommend this, however, as spammers often falsify return addresses.
- Set `$final_spam_destiny` to `D_DISCARD` to discard spam that scores above a "kill level" (specified in Section VII of *amavisd.conf*); spam below the kill level will be tagged and accepted. Use this strategy when your goal is to reduce bandwidth or storage space by dropping messages that are very likely to be spam and tagging others.
- Set `$final_spam_destiny` to `D_BOUNCE` to generate a bounce message to the sender and then discard the message. Because spammers often falsify their return addresses, you will rarely want to use this setting.

Recipient whitelists

Section V of the *amavisd.conf* file focuses on spam policy controls for individual recipients or recipient domains. Its function is analogous to SpamAssassin's `whitelist_to` feature. You can prevent any spam-checking at all, or you can continue to perform spam-checking but prevent spam-handling actions for detected spam.

To prevent any spam-checking at all for email sent to a recipient, set the `@bypass_spam_checks_acl`, `%bypass_spam_checks`, or `$bypass_spam_checks_re` variables. You may use domain names instead of recipient addresses to whitelist all mail sent to a given domain. Here's how you'd set the `@bypass_spam_checks_acl` array to a list of recipients that want to opt out of spam-checking:

```
@bypass_spam_checks_acl = qw( chris@example.com robin@example.com);
```

To use the `%bypass_spam_checks` hash instead, provide recipient addresses as keys and 1 as their values. You might prefer this approach to using `@bypass_spam_checks_acl` if you have a very long list of recipients, because searching a hash is much faster than searching a long list. You can also use the `read_hash` function to read in a list of recipients from an external file and assign them to `%bypass_spam_checks`. This is useful when you want to keep a long list of recipients separate from the *amavisd.conf* file. For example:

```
read_hash(\%bypass_spam_checks, '/var/amavis/bypass_spam');
```

Finally, you can define recipients to opt out by providing a list of regular expressions that match recipient addresses to the `new_RE` function and assigning the result to `$bypass_spam_checks`. This method is useful when you can parsimoniously specify your whitelisted recipients with a regular expression or two. For example:

```
$bypass_spam_checks = new_RE(qr'^(chris|robin)@example\.com'i);
```

Spam checks are bypassed only if all of the recipients of a message have been added to one of these variables. If even one recipient is not listed, spam-checking will still be performed. To ensure that spam is still delivered to whitelisted recipients in such cases, use the "spam_lovers" features discussed next.

If spam checks are bypassed, SpamAssassin's Bayesian classifier will not have an opportunity to learn from a message, whether or not it is spam.

To prevent spam-handling (e.g., tagging or quarantine) from being performed for a recipient when a message has been checked and designated as spam, set the `@spam_lovers_acl`, `%spam_lovers`, or `$spam_lovers_re` variables. These variables are set analogously to the `@bypass_spam_checks_acl`, `%bypass_spam_checks`, and `$bypass_spam_checks_re` variables.

In Example 6-7, *jane@example.com* always receives every message, spam or not, and spam-tagging is skipped when messages are addressed to her alone. In addition, if a message is destined for a domain other than *example.com* (i.e., it's outgoing mail from our domain), spam-tagging is skipped. *postmaster@example.com* also receives every message, but spam-checking is still performed.

Example 6-7. Whitelisting by recipient

```
# Avoid running a spam check if jane is the only recipient, or if
# all recipients are outside of example.com
@bypass_spam_checks_acl = ('jane@example.com', '!.example.com');

# Even if we run a check, don't act on the results for jane or postmaster
@spam_lovers_acl = ('jane@example.com', 'postmaster@example.com');
```

Sender whitelists and blacklists

amavisd can maintain whitelists and blacklists of message senders. It uses a message's envelope address (the one provided in the SMTP MAIL FROM command) as the sender address. Whitelisting ensures that amavisd will allow mail from a whitelisted sender to continue to its intended recipients; blacklisting ensures that amavisd will treat mail from a blacklisted sender as spam.

> amavisd's whitelist and blacklist features do not interact in the same manner as SpamAssassin's. For example, if an address is both whitelisted and blacklisted in SpamAssassin, neither takes effect. If an address is both whitelist and blacklisted in amavisd, both take effect—the message is marked as spam and also allowed to pass to the recipient.

As with other amavisd address-matching features, you can specify addresses to globally whitelist by an array, keys of a hash, or by a set of regular expressions. Set the @whitelist_sender_acl array to a list of sender addresses to whitelist. To use the %whitelist_sender hash instead, provide sender addresses as keys and 1 as their values, or use the read_hash function to read in a list of senders from an external file. Finally, you can specify senders to whitelist by providing a list of regular expressions that match the sender addresses to the new_RE function and assigning the result to $whitelist_sender_re. You may use domain names instead of sender addresses to whitelist all mail sent from a given domain.

You can use a similar set of variables for globally blacklisting senders. The array is @blacklist_sender_acl, the hash is %blacklist_sender, and the regular expression version is $blacklist_sender_re.

The default *amavisd.conf* defines $blacklist_sender_re and %whitelist_sender as shown in Example 6-8. Many username patterns typical of spammers are blacklisted, such as *investments*; many addresses of well-known security and vendor mailing lists are whitelisted. You can modify these definitions or use one of the other variables to add additional sender addresses to the whitelist or blacklist.

Example 6-8. Default blacklist and whitelist entries in amavisd.conf

```
$blacklist_sender_re = new_RE(
    qr'^(bulkmail|offers|cheapbenefits|earnmoney|foryou|greatcasino)@'i,
    qr'^(investments|lose_weight_today|market.alert|money2you|MyGreenCard)@'i,
    qr'^(new\.tld\.registry|opt-out|opt-in|optin|saveonlsmoking2002k)@'i,
    qr'^(specialoffer|specialoffers|stockalert|stopsnoring|wantsome)@'i,
    qr'^(workathome|yesitsfree|your_friend|greatoffers)@'i,
    qr'^(inkjetplanet|marketopt|MakeMoney)\d*@'i,
);

map { $whitelist_sender{lc($_)}=1 } (qw(
  cert-advisory-owner@cert.org
  owner-alert@iss.net
  slashdot@slashdot.org
  bugtraq@securityfocus.com
  NTBUGTRAQ@LISTSERV.NTBUGTRAQ.COM
  security-alerts@linuxsecurity.com
  amavis-user-admin@lists.sourceforge.net
  notification-return@lists.sophos.com
  mailman-announce-admin@python.org
  owner-postfix-users@postfix.org
  owner-postfix-announce@postfix.org
  owner-sendmail-announce@Lists.Sendmail.ORG
  owner-technews@postel.ACM.ORG
  lvs-users-admin@LinuxVirtualServer.org
  ietf-123-owner@loki.ietf.org
  cvs-commits-list-admin@gnome.org
  rt-users-admin@lists.fsck.com
  clp-request@comp.nus.edu.sg
  surveys-errors@lists.nua.ie
  emailNews@genomeweb.com
  owner-textbreakingnews@CNNIMAIL12.CNN.COM
  spamassassin-talk-admin@lists.sourceforge.net
  yahoo-dev-null@yahoo-inc.com
  returns.groups.yahoo.com
));
```

amavisd-new also supports per-recipient blacklists and whitelists of senders. Per-recipient lists override the global lists. Use the `$per_recip_blacklist_sender_lookup_tables` and `$per_recip_whitelist_sender_lookup_tables` variables to specify these lists. Each variable is a reference to a hash keyed by the recipient's address (or domain). The hash value should be a reference to an array of sender addresses, a reference to a hash keyed on sender addresses (with hash values of 1), a call to the `read_hash` function to read the addresses from a file, or a call to `new_RE` with a list of regular expressions to match sender addresses against. For example, you could add the following code to *amavisd.conf* to maintain a list of whitelisted senders for *jane@example.com* in the file */etc/mail/jane-whitelist*:

```
$per_recip_whitelist_sender_lookup_tables =
  { 'jane@example.com' => read_hash('/etc/mail/jane-whitelist')
  };
```

SpamAssassin settings

Several variables in *amavisd.conf* affect the way that `amavisd` invokes SpamAssassin or the actions it takes based on a message's score from SpamAssassin:

`$sa_local_tests_only`
: Set this variable to 1 if you want SpamAssassin to skip network-based tests. It defaults to 0 (perform network-based tests).

`$sa_auto_whitelist`
: Set this variable to 1 to enable SpamAssassin's autowhitelist feature. It defaults to 0 (no autowhitelist). Specify the location of the autowhitelist database in SpamAssassin's sitewide configuration file, *local.cf*. Be sure that the *amavis* user has permission to read from and write to the database.

`$sa_mail_body_size_limit`
: If you set this variable to a size (in bytes), messages larger than the given size will not be checked for spam. This conserves system resources, as SpamAssassin can take a long time to check large messages, and large messages are rarely spam. The variable is undefined by default, which implies no limit. A reasonable value might be 65536 (64Kb) or 102400 (100Kb).

`$sa_tag_level_deflt`
: This variable determines the spam score at or above which *X-Spam-Status* and *X-Spam-Level* headers will be added to the message to show the spam status and level of the message. The default is 3, which is suitable for seeing which tests are and are not being triggered for suspicious messages. If you like to see the spam status of all messages, set this value to –10 or so.

  This variable can be defined on a per-recipient basis much like `$per_recip_blacklist_sender_lookup_tables`. Set `$sa_tag_level_deflt` to a reference to a hash keyed on recipient addresses, with the tag level as the hash value.

`$sa_tag2_level_deflt`
: This variable determines the spam score at or above which `amavisd` adds an *X-Spam-Flag: YES* header and an *X-Spam-Report* header to the message. It may also modify the *Subject* header to tag the message as spam. The default is 6.3.

  This variable can be defined on a per-recipient basis much like `$per_recip_blacklist_sender_lookup_tables`. Set `$sa_tag2_level_deflt` to a reference to a hash keyed on recipient addresses, with the tag2 level as the hash value.

`$sa_kill_level_deflt`
: This variable determines the spam score at or above which `amavisd` will perform spam-handling on the message, such as quarantining the message, discarding it, notifying administrators, etc. By default, this variable is set to the value of `$sa_tag2_level_deflt` so spam-handling is performed on all spam detected. If you want to discard messages that are extremely likely to be spam and tag messages

that are less likely to be spam, set this variable to a higher score (e.g., 12), and only messages above that level will be subject to special handling.

The variable can be defined on a per-recipient basis much like `$per_recip_blacklist_sender_lookup_tables`. Set `$sa_kill_level_deflt` to a reference to a hash keyed on recipient addresses, with the kill level as the hash value.

`$sa_spam_modifies_subj`
: If this variable is set to 1, `amavisd` may modify the *Subject* header of messages with spam scores above the `$sa_tag2_level_dflt` setting. You can also set this variable to a reference to a list of recipients who should have their *Subject* headers modified, a reference to a hash table keyed on recipients who should have their headers modified (with hash values of 1), or the return value of a `new_RE()` call on a list of regular expressions to match against recipients who should have their headers modified. This variable is not defined by default.

`$sa_spam_subject_tag`
: Set this variable to the string to prepend to the *Subject* header of spam messages when `$sa_spam_modifies_subj` is true. If you do not define this, *Subject* headers will never be modified. It is not defined by default; a common definition would be `'*****SPAM***** '`.

Storing recipient preferences in external databases

It's possible to store amavisd-new recipient preferences in an SQL or LDAP database. This can be useful if you want to permit users to modify their own preferences, particularly if you already use an SQL- or LDAP-based user directory. SQL and LDAP lookups override variables defined in *amavisd.conf*.

Database entries indicate user preferences, including whether a user has opted out of spam-checking, whether `amavisd` should modify the *Subject* of spam messages, and user spam tag levels (tag, tag2, kill). Database entries may also specify sender addresses that the recipient wants to blacklist or whitelist.

To enable SQL lookups, define the variable `@lookup_sql_dsn` in *amavisd.conf*. This variable should contain a list of references to three-element arrays that represent database connections. The first element of each array is a Perl *DBI* DSN that defines the database driver to use, the database name, and the name of the database server host. The second element is a database username that `amavisd` will provide for identification to the database server, and the third element is the associated password for authentication. The distributed *amavisd.conf* file provides the following commented-out example:

```
# @lookup_sql_dsn =
#   ( ['DBI:mysql:database=mail;host=127.0.0.1;port=3306', 'user1', 'passwd1'],
#     ['DBI:mysql:database=mail;host=host2', 'username2', 'password2'] );
```

In this example, `amavisd` will first attempt to connect to the MySQL database server on port 3306 of the local host in order to access the *mail* database. It will log into the database server as *user1* with password *passwd1*. If this connection fails, amavisd will try the next database server, a MySQL server running on *host2*, using user *username2* and password *password2*.

The file *README_FILES/README.lookups* in the amavisd-new source code provides definitions for a set of SQL tables that are suitable for configuring user policies and whitelists and blacklists in amavisd. You can add these tables to your SQL database and follow the instructions in *README.lookups* to add appropriate database queries to *amavisd.conf*.

amavisd-new's SQL support should not be confused with SpamAssassin's SQL support. Each controls different aspects of mail-processing.

The amavisd-new source code includes an LDAP schema for an auxiliary class (amavisAccount) that can be added to user accounts. The class defines attributes that determine whether a user has opted out of spam-checking, whether amavisd should modify the *Subject* of spam messages, a user's desired spam tag levels (tag, tag2, kill), and sender addresses to blacklist or whitelist for a user.

To enable LDAP lookups, set the $enable_ldap variable in *amavisd.conf* to 1, and provide LDAP server information in the $default_ldap variable as a reference to a hash:

```
$default_ldap = {
  hostname => 'ldap-server-hostname',
  tls => 1,
  base => 'base DN for ldap searches',
  query_filter => '(&(objectClass=amavisAccount)(mail=%m))'}
};
```

For each preference for which amavisd can perform an LDAP query, you must define additional query parameters to specify (at minimum) the result attribute to be returned from the LDAP database to amavisd. Parameters left undefined will prevent LDAP queries from being performed for that preference. The amavisd source code provides the examples in Example 6-9.

Example 6-9. Defining LDAP query parameters for user preferences

```
$bypass_spam_checks_ldap  = {res_attr => 'amavisBypassSpamChecks'};
$spam_tag_level_ldap      = {res_attr => 'amavisSpamTagLevel'};
$spam_kill_level_ldap     = {res_attr => 'amavisSpamKillLevel'};
$spam_whitelist_sender_ldap = {
  query_filter => '(&(objectClass=amavisAccount)(mail=%m)
                   (amavisWhitelistSender=%s))',
  res_filter => 'OK'};
$spam_blacklist_sender_ldap = {
  query_filter => '(&(objectClass=amavisAccount)(mail=%m)
```

Example 6-9. Defining LDAP query parameters for user preferences (continued)

```
                    (amavisBlacklistSender=%s))',
  res_filter => 'OK'};
```

See the file *README_FILES/README.lookups* in the source code for more information.

Basic Operations

Once you've configured the options in *amavisd.conf*, you're ready to test amavisd. Start amavisd either as the *amavis* user or as *root* (in which case it will change its UID and GID to that of *amavis* during startup).

During your first test, start amavisd with the debug argument. This causes amavisd to run in the foreground and produce debugging information that you can watch to be sure that it's working correctly. Example 6-10 shows a debug startup for a properly functioning configuration:

Example 6-10. Starting amavisd with the debug arguments

```
# amavisd debug
Feb  7 16:58:16 tala amavisd[924]: starting.  amavisd at tala amavisd-new-20030616-p7
Feb  7 16:58:16 tala amavisd[924]: Perl version               5.006001
Feb  7 16:58:16 tala amavisd[924]: Module Amavis::Conf        1.15
Feb  7 16:58:16 tala amavisd[924]: Module Archive::Tar        1.08
Feb  7 16:58:16 tala amavisd[924]: Module Archive::Zip        1.09
Feb  7 16:58:16 tala amavisd[924]: Module Compress::Zlib      1.32
Feb  7 16:58:16 tala amavisd[924]: Module Convert::TNEF       0.17
Feb  7 16:58:16 tala amavisd[924]: Module Convert::UUlib      1.0
Feb  7 16:58:16 tala amavisd[924]: Module MIME::Entity        6.109
Feb  7 16:58:16 tala amavisd[924]: Module MIME::Parser        6.108
Feb  7 16:58:16 tala amavisd[924]: Module MIME::Tools         6.110
Feb  7 16:58:16 tala amavisd[924]: Module Mail::Header        1.60
Feb  7 16:58:16 tala amavisd[924]: Module Mail::Internet      1.60
Feb  7 16:58:16 tala amavisd[924]: Module Mail::SpamAssassin  2.63
Feb  7 16:58:16 tala amavisd[924]: Module Net::Cmd            2.24
Feb  7 16:58:16 tala amavisd[924]: Module Net::DNS            0.45
Feb  7 16:58:16 tala amavisd[924]: Module Net::SMTP           2.26
Feb  7 16:58:16 tala amavisd[924]: Module Net::Server         0.86
Feb  7 16:58:16 tala amavisd[924]: Module Time::HiRes         1.54
Feb  7 16:58:16 tala amavisd[924]: Module Unix::Syslog        0.99
Feb  7 16:58:16 tala amavisd[924]: Found myself: /usr/local/sbin/amavisd -c /etc/amavisd.
conf
Feb  7 16:58:16 tala amavisd[924]: Lookup::SQL code       NOT loaded
Feb  7 16:58:16 tala amavisd[924]: Lookup::LDAP code      NOT loaded
Feb  7 16:58:16 tala amavisd[924]: AMCL-in protocol code  NOT loaded
Feb  7 16:58:16 tala amavisd[924]: SMTP-in protocol code  loaded
Feb  7 16:58:16 tala amavisd[924]: ANTI-VIRUS code        loaded
Feb  7 16:58:16 tala amavisd[924]: ANTI-SPAM  code        loaded
Feb  7 16:58:16 tala amavisd[924]: Net::Server: 2004/02/07-16:58:16 Amavis (type Net::
Server::PreForkSimple) starting! pid(924)
```

Example 6-10. Starting amavisd with the debug arguments (continued)

```
Feb  7 16:58:16 tala amavisd[924]: Net::Server: Binding to TCP port 10024 on host 127.0.
0.1
Feb  7 16:58:16 tala amavisd[924]: Net::Server: Setting gid to "110 110"
Feb  7 16:58:16 tala amavisd[924]: Net::Server: Setting uid to "2013"
Feb  7 16:58:16 tala amavisd[924]: Net::Server: Setting up serialization via flock
Feb  7 16:58:16 tala amavisd[924]: Found $file       at /usr/bin/file
Feb  7 16:58:16 tala amavisd[924]: No $arc,          not using it
Feb  7 16:58:16 tala amavisd[924]: Found $gzip       at /bin/gzip
Feb  7 16:58:16 tala amavisd[924]: Found $bzip2      at /usr/bin/bzip2
Feb  7 16:58:16 tala amavisd[924]: Found $lzop       at /bin/lzop
Feb  7 16:58:16 tala amavisd[924]: Found $lha        at /usr/bin/lha
Feb  7 16:58:16 tala amavisd[924]: Found $unarj      at /usr/bin/arj
Feb  7 16:58:16 tala amavisd[924]: Found $uncompress at /bin/uncompress
Feb  7 16:58:16 tala amavisd[924]: No $unfreeze,     not using it
Feb  7 16:58:16 tala amavisd[924]: Found $unrar      at /usr/bin/unrar
Feb  7 16:58:16 tala amavisd[924]: Found $zoo        at /usr/bin/zoo
Feb  7 16:58:16 tala amavisd[924]: Found $cpio       at /bin/cpio
Feb  7 16:58:16 tala amavisd[924]: Using internal av scanner code for (primary) Clam
Antivirus-clamd
Feb  7 16:58:16 tala amavisd[924]: No primary av scanner: KasperskyLab AVP - aveclient
...many other messages about detecting av scanners...
Feb  7 16:58:16 tala amavisd[924]: SpamControl: initializing Mail::SpamAssassin
Feb  7 16:58:16 tala amavisd[924]: SpamControl: turning on SA auto-whitelisting
Feb  7 16:58:23 tala amavisd[924]: SpamControl: done
Feb  7 16:58:23 tala amavisd[924]: Net::Server: Beginning prefork (2 processes)
Feb  7 16:58:23 tala amavisd[924]: Net::Server: Starting "2" children
Feb  7 16:58:23 tala amavisd[924]: Net::Server: Parent ready for children.
Feb  7 16:58:23 tala amavisd[929]: Net::Server: Child Preforked (929)
Feb  7 16:58:23 tala amavisd[930]: Net::Server: Child Preforked (930)
```

After the startup messages, you should begin to see `amavisd` processing incoming messages (which will produce a copious amount of debugging information). When you are satisfied that messages are being properly delivered back to Postfix, hit Ctrl-C to stop `amavisd debug` and run `amavisd` with no arguments to start the daemon in the background.

If you've chosen to locate your configuration file somewhere other than */etc*, you should either make a symbolic link to */etc/amavisd.conf* or use the `-c` */path/to/amavisd.conf* command-line option to `amavisd`.

Once `amavisd` is running and you confirm that ordinary email is being delivered correctly, test the SpamAssassin functions by sending a copy of the GTUBE string to yourself from a remote system. Because SpamAssassin assigns GTUBE a spam score of 1000, which should be higher than your spam kill level, you should see the message handled by `amavisd`'s spam-handling options.

If `amavisd` appears to work, but SpamAssassin does not, you can enable SpamAssassin debugging by editing *amavisd.conf* and setting the `$sa_debug` variable to 1.

This variable appears at the end of *amavisd.conf*. You must stop `amavisd` and restart it with the `debug` argument for SpamAssassin debugging to be performed.

> Anytime you make a change to *amavisd.conf*, you must inform `amavisd` by issuing the command `amavisd reload` (or stopping and restarting the daemon).

The amavisd-new distribution includes a script named `amavisd_init.sh` that you can use as a boot script for systems based on RedHat Linux. With a little modification, it makes a suitable boot script for other Unix systems to automatically start and stop `amavisd`.

Adding Sitewide Bayesian Filtering

You can easily add sitewide Bayesian filtering to amavisd-new. Use the usual SpamAssassin `use_bayes` and `bayes_path` directives in *local.cf*, and ensure that the *amavis* user has permission to create the databases in the directory named in `bayes_path`. One way to do this is to use a directory for the databases that is owned by *amavis*, such as */var/amavis*. Another option is to locate the databases in a directory owned by another user but to create them ahead of time and `chown` them to *amavis*. If local users need to have access to the databases (e.g., they will be running `sa-learn`), you might have to make the databases readable or writable by a group other than *amavis* and adjust the `bayes_file_mode`, or make them world readable or writable. Doing so, however, puts the integrity of your spam-checking at the mercy of the good intentions and comprehension of your users.

If users have shell accounts on the system, you can use per-user Bayesian filtering with amavisd-new instead. Configure SpamAssassin for per-user databases as usual, but ensure that each user's databases are group-owned by the *amavis* group and have group read/write permissions so that amavisd-new can use them. Doing so allows users to run `sa-learn` themselves to train their databases, while still permitting amavisd-new to access them. With SpamAssassin 3.0, you could also store per-user Bayesian data in an SQL database.

Adding Sitewide Autowhitelisting

amavisd knows how to use autowhitelisting (see the discussion of `$sa_auto_whitelist` earlier in this chapter). Just add the usual SpamAssassin `auto_whitelist_path` and `auto_whitelist_file_mode` directives to *local.cf*. As with the Bayesian databases, the *amavis* user must have permission to create the autowhitelist database and read and write to it.

Routing Email Through the Gateway

Once Postfix and amavisd-new are receiving messages for the local host and performing SpamAssassin checks on them, you can start accepting email for your domain and routing it to an internal mail server after spam-checking. Figure 6-4 illustrates this topology.

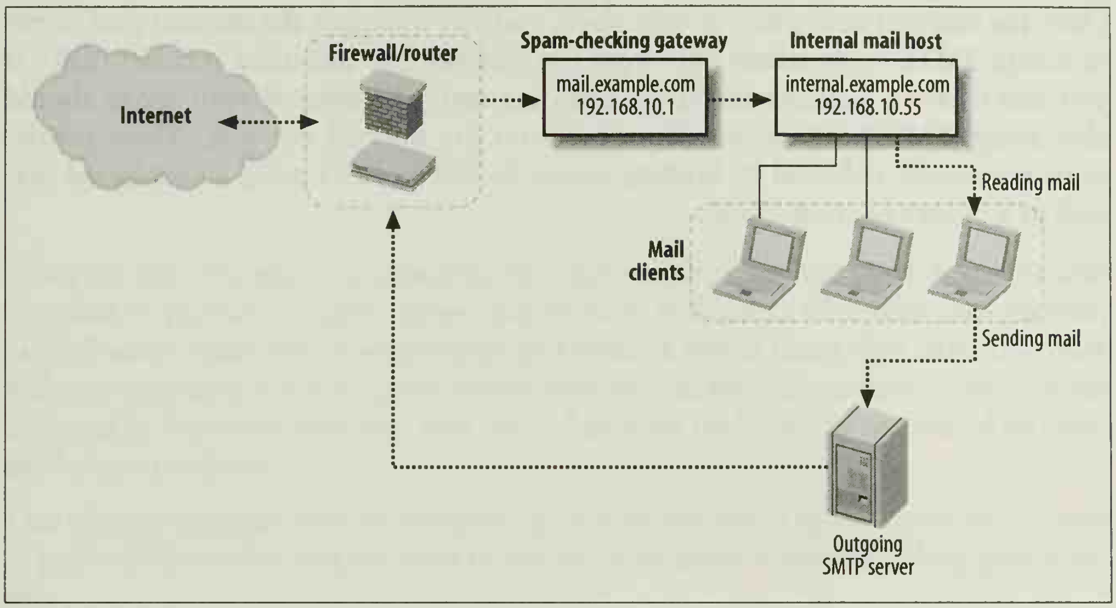


Figure 6-4. Spam-checking gateway topology

Postfix changes

To configure Postfix to relay incoming mail for *example.com* to *internal.example.com*, add the following line to */etc/postfix/main.cf*:

```
transport_maps=hash:/etc/postfix/transport
```

Then, create the */etc/postfix/transport* file, and add either:

```
example.com   smtp:internal.example.com
```

or, if *mail.example.com* cannot resolve the name *internal.example.com*, you could use

```
example.com   smtp:[129.168.10.55]
```

Run the command `postmap /etc/postfix/transport` to build the transport map from */etc/postfix/transport*, and run `postfix reload` to reload Postfix's configuration.

Routing changes

Mail from the Internet for *example.com* should be sent to the spam-checking gateway *mail.example.com*. Add a DNS MX record for the *example.com* domain that points to *mail.example.com*.

Once received by *mail.example.com*, messages will be spam-checked and should then be relayed to *internal.example.com* by Postfix. No DNS records for *internal.example.com* need be published to the Internet, but it's useful if *mail.example.com* can resolve *internal.example.com*.

Internal server configuration

Once the external mail gateway is in place, you can configure the internal mail server to accept SMTP connections only from the gateway (for incoming Internet mail). If you don't have a separate server for outgoing mail, the internal mail server should also accept SMTP connections from hosts on the internal network. These restrictions are usually enforced by limiting access to TCP port 25 using a host-based firewall or a packet-filtering router.

CHAPTER 7

Integrating SpamAssassin with qmail

qmail is a mail transport agent written by cryptography researcher Dan Bernstein and designed to provide a highly secure mail system. It consists of several components, each of which runs with least privilege and none of which trusts data from the other without validating it itself. qmail works best in concert with several other systems designed by Bernstein that take over other functions traditionally performed by standard system utilities.

This chapter explains how to integrate SpamAssassin into a qmail-based mail server to perform spam-checking for local recipients or to create a spam-checking mail gateway.

qmail is a complex piece of software and, like most MTAs, offers scores of configuration choices. This chapter assumes that you are running the netqmail 1.05 version of qmail 1.03 and does not cover how to securely install, configure, or operate qmail itself. For that information, see the qmail documentation, David Sill's *Life with qmail* web site (*http://www.lifewithqmail.org*) and *The qmail Handbook* by David Sill (Apress) or *qmail* by John Levine (O'Reilly).

This chapter assumes that you have set up your qmail system as described in *Life with qmail* and that you are using the recommended *daemontools* and *ucspi-tcp* packages.

qmail Architecture

Several different qmail components play roles in receiving messages from the Internet. Messages from the Internet typically enter the mail server via the `qmail-smtpd` daemon, which listens on port 25 and conducts the SMTP transaction with the remote sender. `qmail-smtpd` passes the messages to the `qmail-queue` program, which stores them in an outgoing queue for further processing. The `qmail-send` daemon reads the messages in the outgoing queue and attempts to deliver them using either the `qmail-lspawn` daemon (which passes it to the `qmail-local` program for local delivery) or the `qmail-rspawn` daemon (which passes them to the `qmail-remote` program for relaying to remote hosts). Figure 7-1 illustrates the flow of email through qmail components.

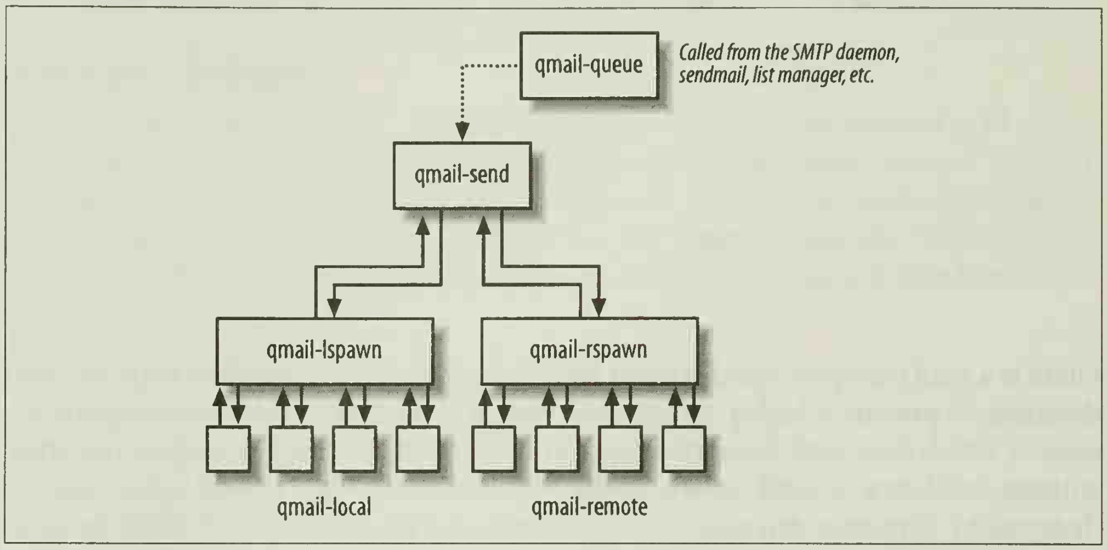


Figure 7-1. qmail architecture during message receipt

Most systems keep all of qmail's files in */var/qmail*. Configuration files reside in */var/qmail/control*.

Spam-Checking During Local Delivery

The easiest way to add SpamAssassin to a qmail system is to configure qmail to pipe messages through SpamAssassin during local delivery.

The advantages of this approach are:

- It's easy to set up.
- You can run `spamd` and can use `spamc` for faster spam-checking.
- User preference files, autowhitelists, and Bayesian databases can be used, because qmail will invoke SpamAssassin as the user to whom it is delivering a message.

However, qmail runs a local delivery agent only for email destined for a local recipient. You cannot create a spam-checking gateway with this approach.

If you're using the installation described in the *Life with qmail* web site, the */var/qmail/control/defaultdelivery* file contains a line that specifies either a directory (e.g., *./Maildir/*) or a filename (e.g., *./Mailbox*). The */var/qmail/rc* script passes the contents of *defaultdelivery* to `qmail-start`, and thence to `qmail-lspawn` and `qmail-local`.

If you deliver to a *maildir* directory, change the line in your *defaultdelivery* file to read:

```
| /usr/bin/spamc | maildir ./Maildir/
```

In this case, be sure you've installed the *safecat* package, which includes the `maildir` script. You can get *safecat* at *http://www.pobox.com/~lbudney/linux/software/safecat.html*.

If you deliver to a *mailbox* file in each user's home directory, install procmail and change the line in *defaultdelivery* to read:

```
        | preline procmail
```

In this case, the system's */etc/procmailrc* file should have a default recipe that looks like this:

```
:0fw
* <300000
|/usr/bin/spamc

:0:
$HOME/Mailbox
```

> The default delivery method is used only when users don't have their own *.qmail* files. This permits users to override spam-checking. Conversely, if you don't do spam-checking by default during local delivery, any user can add the preceding lines to her *.qmail* file and have her messages checked.

Spam-Checking All Incoming Mail

If you want to set up a spam-checking gateway for all recipients, local or not, you need a way to perform spam-checking as mail is received, before final delivery. qmail provides this capability through the QMAILQUEUE patch, which is included in the netqmail distribution of qmail (and most packaged qmail distributions).

> You can find out if your qmail installation has the QMAILQUEUE patch applied by executing the following commands:
>
> ```
> # cd /var/qmail/bin
> # strings qmail-smtpd | grep QMAILQUEUE
> QMAILQUEUE
> ```
>
> If you don't see `QMAILQUEUE` in response to the `strings` command, the patch has not been applied. You will have to recompile qmail from the netqmail source code.

With the QMAILQUEUE patch applied, the `qmail-smtpd` daemon checks to see if the environment variable `QMAILQUEUE` has been set. If so, `qmail-smtpd` hands the message off to the program specified in that variable instead of to the default `qmail-queue` program. The new program can call SpamAssassin and then pass the (possibly tagged) message to qmail-queue. Figure 7-2 illustrates this arrangement.

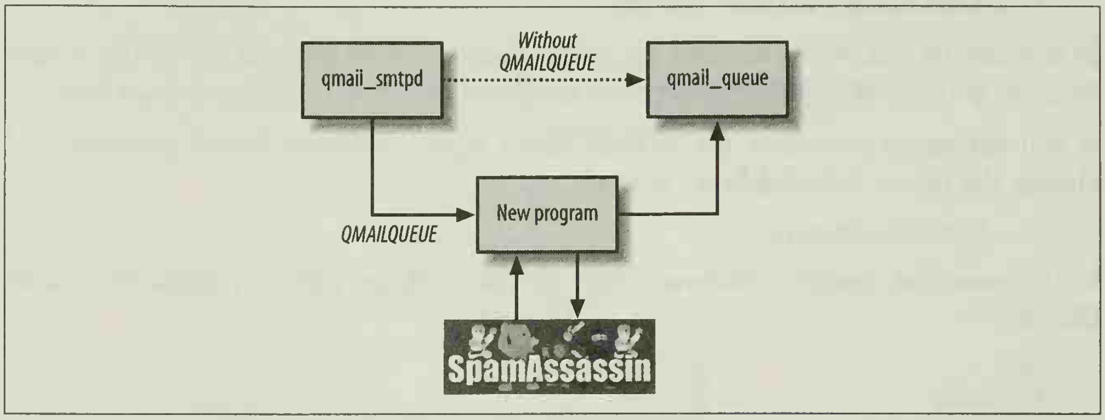


Figure 7-2. qmail configuration to check all incoming email for spam

SpamAssassin includes a small C program called qmail-spamc by John Peacock, with its source code (in the *qmail* subdirectory in SpamAssassin 2.63, and in the *spamc* subdirectory in SpamAssassin 3.0). When compiled, qmail-spamc is suitable for use as a QMAILQUEUE program; it invokes spamc on an incoming message and pipes the result to qmail-queue. Because it's written in C and is a very simple program, it runs quickly. To set up qmail-spamc, perform the following steps:

1. Compile *qmail-spamc.c*. On most systems, issue a command like the following in the directory containing *qmail-spamc.c*:

   ```
   cc -O -o qmail-spamc qmail-spamc.c
   ```

2. As *root*, install qmail-spamc in an appropriate location on your system (e.g., */var/qmail/bin* or */usr/local/bin*). Make it executable. For example:

   ```
   install -m 755 qmail-spamc /var/qmail/bin
   ```

3. Ensure that qmail-queue is on the system's default path. The easiest way to do so is usually to create a symbolic link from */var/qmail/bin/qmail-queue* to */usr/bin/qmail-queue*. Do the same for spamc if it is not already installed in */usr/bin*. For example:

   ```
   ln -s /var/qmail/bin/qmail-queue /usr/bin/qmail-queue
   ```

4. Ensure that spamd is running.

5. Ensure that qmail-smtpd has enough memory available to allow it to run qmail-spamc and spamc. Edit */var/qmail/supervise/qmail-smtpd/run* and modify the -m and/or -a arguments of softlimit to increase the number of bytes available to qmail-smtpd and its child processes to an amount sufficient to allow all of the processes to execute completely on a large message. A setting of 10MB (roughly 10,000,000) is usually sufficient, but you may have to vary the setting and keep an eye on your logs to find the right amount. If the setting is too low, you will

see errors such as the following at the end of the DATA step during SMTP transactions:

```
fatal: qq temporary problem (#4.3.0)
```

6. Edit */etc/tcp.smtp*. This file controls access to the SMTP service when you're using *ucspi-tcp*. Add or modify the line shown in bold:

   ```
   127.:allow,RELAYCLIENT=""
   :allow,QMAILQUEUE="/var/qmail/bin/qmail-spamc"
   ```

   This change causes the `QMAILQUEUE` environment variable to be set when `qmail-smtpd` is invoked by a connection from hosts outside the *127.* network (i.e., spam-checking will be performed on email from remote hosts, but not from the local host).

   With the version of `qmail-spamc` distributed with SpamAssassin 3.0, you can customize the way `spamc` is invoked by adding additional environment variables to the list in */etc/tcp.smtp*, including:

   `SPAMDSOCK="`*`/path/to/socket`*`"`
   : Direct `spamc` to use the given path to a Unix socket for connecting to `spamd`.

   `SPAMDHOST="`*`hostname`*`"`
   : Direct `spamc` to connect to `spamd` at the given host.

   `SPAMDPORT="`*`port-number`*`"`
   : Direct `spamc` to connect to `spamd` at the given TCP port number.

   `SPAMDSSL="1"`
   : Direct `spamc` to connect to `spamd` using SSL.

   `SPAMDSIZE="`*`number-of-bytes`*`"`
   : Direct `spamc` not to perform spam-checking on messages that exceed *`number-of-bytes`* in size.

   `SPAMDUSER="`*`username`*`"`
   : Direct `spamc` to supply the given username to `spamd`.

7. Update the TCP rules database by running the command `qmailctl cdb`, which is found in your */var/qmail/bin/* directory. At this point, all incoming remote SMTP connections should have their messages passed through `qmail-spamc`.

You can emulate the QMAILQUEUE approach without the QMAILQUEUE patch by renaming `qmail-queue` to `qmail-queue.orig` and writing a new `qmail-queue` script that pipes the message through SpamAssassin and then to `qmail-queue.orig`, like this:

```
#!/bin/sh
PATH=/var/qmail/bin:$PATH
| spamc | qmail-queue.orig
```

However, this approach is less flexible than using QMAILQUEUE and more prone to causing trouble later when you want to queue messages without spam-checking them.

Building a Spam-Checking Gateway

Several content-filtering daemons that call SpamAssassin are available for qmail. This section provides a complete sample installation of qmail-scanner, a particularly flexible filter that supports both spam-checking and virus-checking. qmail-scanner is written in Perl and available at *http://qmail-scanner.sourceforge.net/*. The version used in this section's example is 1.21. Some of qmail-scanner's features include:

- The filter was specifically developed and tested for qmail.
- Messages can be rejected based on MIME type or extensions of attached filenames.
- Messages can be rejected based on invalid formatting.
- Messages can be checked with multiple virus scanners, and messages carrying viruses can be refused, discarded, or quarantined.
- SpamAssassin can be invoked on a message, and spam can be refused, discarded, quarantined, or tagged.

The rest of this chapter details the installation, configuration, and operation of qmail-scanner as an example of a full-scale approach to using SpamAssassin with qmail. qmail-scanner's other functions, such as virus-checking, are mentioned but not covered in detail; read the documentation to learn more about these features.

Installation

qmail-scanner is written in Perl and invokes SpamAssassin by running `spamc`, so you must run `spamd` to use qmail-scanner. You should set up `spamd` before you install qmail-scanner. Install SpamAssassin (and your antivirus software) first, then install qmail-scanner. qmail-scanner also requires some other Perl modules, including: *Time::HiRes*, *DB_File*, and *Sys::Syslog*. You can install these Perl modules using CPAN as described in Chapter 2. You must also install the Maildrop software package (*http://www.courier-mta.org/download.php*), and if you plan to perform virus-checking, TNEF (*http://sourceforge.net/projects/tnef/*).

> qmail-scanner requires the 5.005_03 version of Perl or later. Perl must be compiled to allow *setuid* Perl scripts; often this means that a separate `suidperl` program is available on the system. If your system's Perl does not support *setuid* Perl scripts, you may be able to find a package for your system that does, you may build Perl from source code and enable support, or you may compile a *setuid* wrapper program in C (described later in this chapter).

Begin the install process by creating a new user account and group for running qmail-scanner; the usual name for both the user and group is *qscand*. The new user will own qmail-scanner's files, and the user (or group) must have access to Spam-

Assassin's configuration and database files as well. The user's home directory is traditionally */home/qscand*, but you can create it anywhere that fits your system's needs.

qmail-scanner uses several important directories and files in */var/spool/qmailscan*. For example, quarantined messages are stored in */var/spool/qmailscan/quarantine*, and qmail-scanner logs its operations in */var/spool/qmailscan/qmail-queue.log*. The directories */var/spool/qmailscan/tmp* and */var/spool/qmailscan/working* are temporary directories used for unpacking and processing messages. For optimal performance, these directories should be on a fast disk—even a RAM disk if your operating system supports it and you have enough memory to spare. In contrast, the *quarantine* directory should never be located on a RAM disk because you will often want to be sure that you can access quarantined files.

Next, download the qmail-scanner source code, unpack it, and build it. You must be *root* to configure and build qmail-scanner. The qmail-scanner build process uses the familiar `configure` command to configure and build qmail-scanner's components, which you then install.

qmail-scanner Configuration Options

qmail-scanner has only a few `configure` options related to SpamAssassin. If you don't specify any options, qmail-scanner will use `spamc -c` for spam-checking and will add *X-Spam-Status* and *X-Spam-Level* headers to messages, but will not modify the *Subject* header of spam messages.

If you specify the `--scanners 'fast_spamassassin=`*`string`*`'` command-line option to `configure`, qmail-scanner will also modify the *Subject* header of spam messages by prepending a *`string`*. A typical choice for *`string`* might be `SPAM`. If you plan to use other virus-scanners, you must specify thom in this command-line option as well or qmail-scanner will not use them. (If you've already installed qmail-scanner and want to start adding a *Subject* header tag, you can also edit the */var/qmail/bin/qmail-scanner-queue.pl* file itself; search for the line that defines the `$spamc_subject` variable, and modify it to set your subject prefix.)

If you specify the `--scanners verbose_spamassassin` command-line option to `configure`, qmail-scanner will use `spamc` without the `-c` option. This alternative runs more slowly, because the entire spam-checked message is read back from `spamc` instead of just the spam scores. The advantage of this configuration, however, is that messages will be tagged exactly as defined in the SpamAssassin rules and report templates. For example, you'll get the SpamAssassin headers that report which spam tests matched, any custom headers you've defined, and full MIME-rewriting of messages. If you plan to use other virus scanners, you must specify them in this command-line option as well or qmail-scanner will not use them.

To configure qmail-scanner, use the commands shown in Example 7-1. The example also reproduces the output you should expect.

Example 7-1. Building qmail-scanner

```
$ tar xfz qmail-scanner-1.21.tar.gz
$ cd qmail-scanner-1.21
$ su
Password: XXXXXXXX
# ./configure --install
Building Qmail-Scanner 1.21...

This script will search your system for the virus scanners it knows
about, and will ensure that all external programs
qmail-scanner-queue.pl uses are explicitly pathed for performance
reasons.

It will then generate qmail-scanner-queue.pl - it is up to you to install it
correctly.

Continue? ([Y]/N) Y

/usr/bin/uudecode works as expected on system...

The following binaries and scanners were found on your system:

mimeunpacker=/usr/local/bin/reformime
uudecode=/usr/bin/uudecode
unzip=/usr/bin/unzip

Content/Virus Scanners installed on your System

fprot=/usr/local/bin/f-prot
fast_spamassassin=/usr/local/bin/spamc

Qmail-Scanner details.

log-details=0
fix-mime=2
ignore-eol-check=0
debug=1
notify=psender,nmlvadm
redundant-scanning=no
virus-admin=postmaster@example.com
local-domains='example.com'
silent-
viruses='klez','bugbear','hybris','yaha','braid','nimda','tanatos','sobig','winevar','pal
yh','fizzer','gibe','cailont','lovelorn','swen','dumaru','sober','hawawi','holar-
i','mimail','poffer','bagle','worm.galil','mydoom','worm.sco','tanx','novarg','@mm'
scanners="fprot_scanner","fast_spamassassin"

If that looks correct, I will now generate qmail-scanner-queue.pl
for your system...
```

Example 7-1. Building qmail-scanner (continued)

```
Continue? ([Y]/N) Y

Finished. Please read README(.html) and then go over the script to
check paths/etc, and then install as you see fit.

Remember to copy quarantine-attachments.txt to /var/spool/qmailscan and then
run "qmail-scanner-queue.pl -g" to generate DB version.

              ****** FINAL TEST ******

Please log into an unpriviledged account and run
/var/qmail/bin/qmail-scanner-queue.pl -g

If you see the error "Can't do setuid", or "Permission denied", then
refer to the FAQ.

(e.g.  "setuidgid qmaild /var/qmail/bin/qmail-scanner-queue.pl -g")


That's it! To report success:

   % (echo 'First M. Last'; cat SYSDEF)|mail jhaar-s4vstats@crom.trimble.co.nz
Replace First M. Last with your name.
```

As with qmail-spamc, ensure that qmail-smtpd has enough memory available to allow it to run qmail-scanner-queue.pl, any virus checkers you have configured, and spamc. Edit */var/qmail/supervise/qmail-smtpd/run* and modify the -m and/or -a arguments of softlimit to increate the number of bytes available to qmail-smtpd and its child processes to an amount sufficient to allow all of the processes to execute completely on a large message.

To enable qmail-scanner, edit */etc/tcp.smtp*. Add or modify lines such as those shown in bold:

```
127.:allow,RELAYCLIENT=""
192.168.:allow,RELAYCLIENT="",QMAILQUEUE="/var/qmail/bin/qmail-scanner-queue.pl"
10.:allow,RELAYCLIENT="",QS_SPAMASSASSIN="on",QMAILQUEUE="/var/qmail/bin/qmail-
scanner-queue.pl"
:allow,QMAILQUEUE="/var/qmail/bin/qmail-scanner-queue.pl"
```

When you invoke qmail-scanner with qmail's RELAYCLIENT variable set, as in the line for connections from the 192.168/16 network, only virus-checking is performed, unless you also include QS_SPAMASSASSIN="on", as in the line for connections from the 10/8 network. When you invoke it without setting RELAYCLIENT, as in the line for default connections, both virus-checking and spam-checking are performed.

Be sure to run /var/qmail/bin/qmailctl cdb after updating */etc/tcp.smtp*.

No setuid Perl

When qmail-scanner's `configure` script can't find a suitable version of Perl for running *setuid* scripts, it prints out an error like this:

```
Testing suid nature of /usr/bin/suidperl...
Whoa - broken perl install found.
Cannot even run a simple script setuid

Installation of Qmail-Scanner FAILED
```

If you can't (or don't want to) install a Perl that runs *setuid* scripts, you can use a *setuid* wrapper in C instead. Follow these steps as *root*:

1. Install qmail-scanner with `./configure --skip-setuid-test --install`. This will produce an error at the end of the installation.
2. Compile and install the C wrapper with `(cd contrib; make install)`. If you're not using the default `qscand` user and group and */var/qmail/bin* directory for installation, you'll have to edit *contrib/Makefile* first.
3. Remove the setuid bit from */var/qmail/bin/qmail-scanner-queue.pl* with `chmod 0755 /var/qmail/bin/qmail-scanner-queue.pl`.
4. Edit */var/qmail/bin/qmail-scanner-queue.pl* and change the first line from `#!/usr/bin/suidperl -T` to `#!/usr/bin/perl -T`.
5. Use *qmail-scanner-queue* (the compiled C wrapper) in place of *qmail-scanner-queue.pl* in the rest of the qmail-scanner setup process.

Initialization

The first time you install qmail-scanner, you must direct it to initialize its databases. As the *qscand* user, run these commands:

```
$ /var/qmail/bin/qmail-scanner-queue.pl -z
$ /var/qmail/bin/qmail-scanner-queue.pl -g
perlscanner: generate new DB file from /var/spool/qmailscan/quarantine-attachments.
txt
perlscanner: total of 9 entries.
```

Basic Operations

qmail-scanner comes with a shell script called `test_installation.sh` that can be used to exercise an installation. Example 7-2 shows how to run the script, along with its output.

Example 7-2. Testing qmail-scanner

```
# cd contrib
# QMAILQUEUE="/var/qmai/bin/qmail-scanner-queue.pl" ./test_installation.sh -doit
```

Example 7-2. Testing qmail-scanner (continued)

```
Sending standard test message - no viruses...
done!

Sending eicar test virus - should be caught by perlscanner module...
done!

Sending eicar test virus with altered filename - should only be caught by commercial
anti-virus modules (if you have any)...

Sending bad spam message for anti-spam testing - In case you are using SpamAssassin...
Done!

Finished test. Now go and check Email for root
```

If qmail-scanner's spam-checking is operating properly, *root* (or the user that receives *root*'s email) should receive a non-spam message like this:

```
From MAILER-DAEMON Tue Mar 23 05:03:28 2004
From: Qmail-Scanner Test <example.com@example.com>
Received: from  by example.com by uid 0 with qmail-scanner-1.21
 (f-prot: 3.11/. spamassassin: 2.63.  Clear:RC:1(127.0.0.1):SA:0(0.0/5.0):.
 Processed in 5.577981 secs); 23 Mar 2004 05:03:28 -0000
To: Root Account <root@example.com>
Subject: Qmail-Scanner test (1/4): inoffensive message
Date: 23 Mar 2004 05:03:22 -0000
Delivered-To: root@example.com
X-Spam-Status: No, hits=0.0 required=5.0

Message 1/4

This is a test message. It should arrive unaffected.
```

The same user should also receive a spam message like this:

```
From MAILER-DAEMON Tue Mar 23 05:03:41 2004
Received: from  by example.com by uid 0 with qmail-scanner-1.21
 (f-prot: 3.11/. spamassassin: 2.63.  Clear:RC:1(127.0.0.1):SA:1(16.7/5.0):.
 Processed in 5.129358 secs); 23 Mar 2004 05:03:40 -0000
X-Spam-Status: Yes, hits=16.7 required=5.0
X-Spam-Level: ++++++++++++++++
Delivery-Date: Mon, 19 Feb 2001 13:57:29 +0000
Delivered-To: jm@netnoteinc.com
Received: from webnote.net (mail.webnote.net [193.120.211.219])
        by mail.netnoteinc.com (Postfix) with ESMTP id 09C18114095
        for <jm7@netnoteinc.com>; Mon, 19 Feb 2001 13:57:29 +0000 (GMT)
Received: from netsvr.Internet (USR-157-050.dr.cgocable.ca [24.226.157.50] (may
+be forged))
        by webnote.net (8.9.3/8.9.3) with ESMTP id IAA29903
        for <jm7@netnoteinc.com>; Sun, 18 Feb 2001 08:28:16 GMT
From: sb55sb55@yahoo.com
Received: from R0OUqS18S (max1-45.losangeles.corecomm.net [216.214.106.173]) by
+netsvr.Internet with SMTP (Microsoft Exchange Internet Mail Service Version
+5.5.2653.13)
        id 1429NTL5; Sun, 18 Feb 2001 03:26:12 -0500
```

```
DATE: 18 Feb 01 12:29:13 AM
Message-ID: <9PS291LhupY>
Subject: Qmail-Scanner anti-spam test (4/4): checking SpamAssassin [if present]
+(There yours for FREE!)
To: undisclosed-recipients: ;

Congratulations! You have been selected to receive 2 FREE 2 Day VIP Passes to
Universal Studios!

Click here http://209.61.190.180

As an added bonus you will also be registered to receive vacations discounted 25%-
75%!


@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
This mailing is done by an independent marketing co.
We apologize if this message has reached you in error.
Save the Planet, Save the Trees! Advertise via E mail.
No wasted paper! Delete with one simple keystroke!
Less refuse in our Dumps! This is the new way of the new millennium
To be removed please reply back with the word "remove" in the subject line.
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Note the bold lines in the messages. These are headers demonstrating that the messages were processed by qmail-scanner, and in the case of the spam message, that qmail-scanner can recognize spam.

qmail-scanner uses */var/spool/qmailscan* as a working directory and quarantine area for viruses. By default, qmail-scanner's operations are logged to the */var/spool/qmailscan/qmail-queue.log* file, which should be added to your log rotation schedule. Errors are also reported to qmail's log files.

When an SMTP session is dropped partway, temporary files may remain in */var/spool/qmailscan*. These messages can be cleared out by running `/var/qmail/bin/qmail-scanner-queue.pl -z`. Set up a `cron` job to execute this command once a day to delete older files in this directory.

Per-User Spam Preferences

qmail-scanner invokes `spamc` with the `-u` *`recipient`* argument when a message has a single recipient. Accordingly, in this case, per-user spam-checking preferences (either from users' *.spamassassin/user_prefs* files or from an SQL or LDAP database if `spamd` is so configured) will be applied when qmail-scanner checks messages. When a message has multiple recipients, qmail-scanner uses the default preferences.

Although there is no way to configure qmail to force senders to send messages with one recipient at a time, qmail itself always breaks up a multirecipient message when it is *sending* and sends copies of the message to single recipients. Ron Culler pointed out in a December 2003 message to the *qmail-scanner-general* mailing list that one way to ensure that every message has only a single recipient is to run a pair of qmail gateways. The first gateway receives messages from the Internet and can perform some general scanning (e.g., refusing viruses) before forwarding messages on to the second gateway for spam-checking. Because the first qmail server will always split up multirecipient messages before sending them, the second qmail server will always receive messages with a single recipient and can apply per-user spam preferences.

If you built qmail-scanner using the default `fast_spamassassin` configuration (described in the earlier "qmail-scanner Configuration Options" sidebar), `spamc` is invoked with the `-c` option. This limits which per-user spam preferences are applied: spam thresholds and score modifications will work, but preferences that affect the way messages or headers are rewritten will not (because `spamc -c` returns only a spam score, not a rewritten message). Use the `verbose_spamassassin` configuration if you need to enable these preferences.

Sitewide Bayesian Filtering

You can easily add sitewide Bayesian filtering to qmail-scanner. Use the usual SpamAssassin `use_bayes` and `bayes_path` directives in *local.cf*, and ensure that the *spamd* user has permission to create the databases in the directory named in `bayes_path`.

Sitewide Autowhitelisting

Adding autowhitelisting is just as easy. Add the usual SpamAssassin `auto_whitelist_path` directive to *local.cf*, and if you're using SpamAssassin 2.63, invoke `spamd` with the `--auto-whitelist` option (which is unnecessary in SpamAssassin 3.0). As with the Bayesian databases, the *spamd* user must have permission to create the autowhitelist database and read and write to it.

Routing Email Through the Gateway

Once you have qmail and qmail-scanner receiving messages for the local host and performing SpamAssassin checks on them, you can start accepting email for your domain and routing it to an internal mail server after spam-checking. Figure 7-3 illustrates this topology.

The following sections describe the changes you need to make to implement the topology shown in Figure 7-3.

qmail changes

To configure qmail to relay incoming mail for *example.com* to *internal.example.com*, add the following line to */var/qmail/control/rcpthosts*:

```
example.com
```

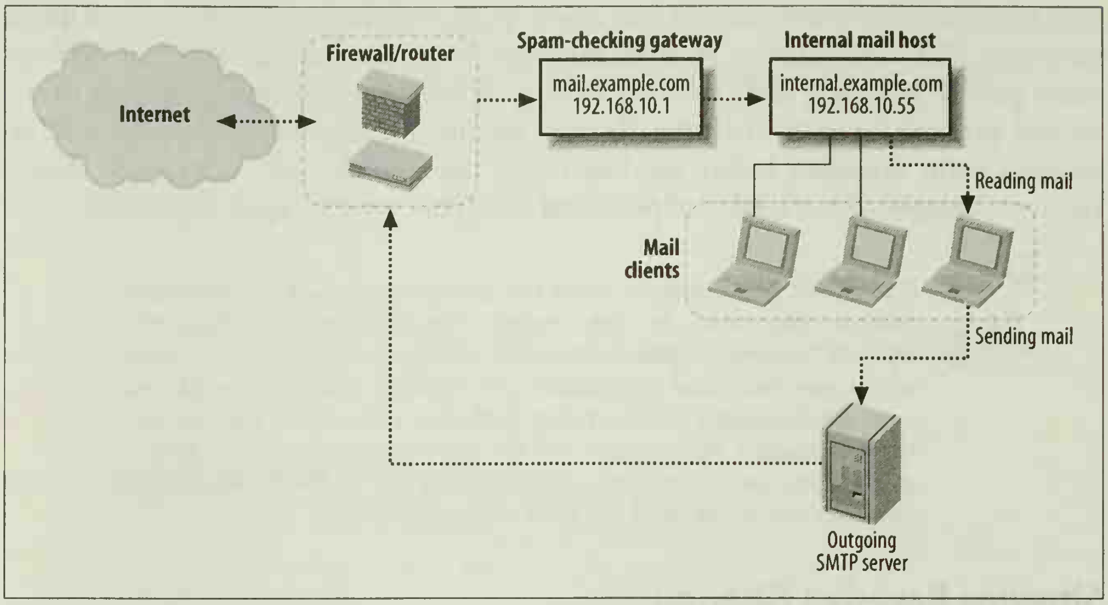


Figure 7-3. Spam-checking gateway topology

Then, create the */var/qmail/control/smtproutes* file, and add either:

```
example.com:internal.example.com
```

or, if *mail.example.com* can look up an (internal) MX record for *example.com* that points to *internal.example.com* (and possibly other internal mail servers), you could use

```
example.com:
```

Routing changes

Mail from the Internet for *example.com* should be sent to the spam-checking gateway *mail.example.com*. Add a DNS MX record for the *example.com* domain that points to *mail.example.com*.

Once received by *mail.example.com*, messages will be spam-checked and should then be relayed to *internal.example.com* by qmail. No DNS records for *internal.example.com* need be published to the Internet, but it's necessary that *mail.example.com* can resolve *internal.example.com*.

Internal server configuration

Once the external mail gateway is in place, you can configure the internal mail server to accept SMTP connections only from the gateway (for incoming Internet mail). If you don't have a separate server for outgoing mail, the internal mail server should also accept SMTP connections from hosts on the internal network. These restrictions are usually enforced by limiting access to TCP port 25 using a host-based firewall or a packet-filtering router.

CHAPTER 8

Integrating SpamAssassin with Exim

Exim is an MTA developed by Philip Hazel at the University of Cambridge. Exim is designed for Internet mail hosts and provides flexibility, performance, and strong access controls. It has become a popular replacement for sendmail because it provides a compatible command-line interface.

This chapter explains how to integrate SpamAssassin into an Exim-based mail server to perform spam-checking for local recipients or to create a spam-checking mail gateway.

Exim is a complex piece of software and, more than most MTAs, has an extensive and complicated set of configuration options. This chapter assumes that you are running Exim 4 and does not cover how to securely install, configure, or operate Exim itself. For that information, see the Exim documentation, the web site *http://www.exim.org*, and the book *The Exim SMTP Mail Server: Official Guide for Release 4* by Philip Hazel (UIT Cambridge).

Exim consists primarily of a single *setuid* executable, `exim`, that performs different functions depending on its command-line arguments. These functions include listening on the SMTP port and receiving and enqueuing incoming messages, adding locally generated messages to the queue, and processing the queue to transmit outgoing messages. When compiled from source code, `exim` is installed in */usr/exim/bin*, and the examples in this chapter assume that directory is used.

Exim's configuration file defaults to */usr/exim/configure*. The configuration file determines the behavior of Exim and defines three important logical entities: *access control lists (ACLs)*, *routers*, and *transports*. ACLs define tests that can be performed during incoming SMTP sessions to determine whether Exim will accept a message. Routers determine how messages to a given address should be delivered (or rewritten to new addresses) and queue them up for transports. Transports define delivery mechanisms—methods by which a message can be copied from Exim's queue to a local mailbox, a remote host, or elsewhere. Each of these entities has its own section in the configuration file.

While you can define ACLs and transports in any order, you must define routers in the order in which they are to run. In the default configuration, the router order is `dnslookup` (look up remote hostnames and route messages via SMTP), `system_aliases` (redirect messages on the basis of the */etc/aliases* file), `userforward` (redirect messages on the basis of user *.forward* files), and `localuser` (route message via the local delivery agent).

Spam-Checking via procmail

One easy way to add SpamAssassin to an Exim system is to configure Exim to use procmail as its local delivery agent. Then add a procmail recipe for spam-tagging to */etc/procmailrc*.

The advantages of this approach are

- It's very easy to set up.
- You can run `spamd`, and the procmail recipe can use `spamc` for faster spam-checking.
- User preference files, autowhitelists, and Bayesian databases can be used.

However, Exim runs a local delivery agent only for email destined for a local recipient. You cannot create a spam-checking gateway with this approach.

To configure Exim to use procmail for local delivery, add the following transport to the Exim configuration file (in the transports section):

```
procmail_pipe:
  driver = pipe
  command = /usr/local/bin/procmail -d $local_part
  return_path_add
  delivery_date_add
  envelope_to_add
  check_string = "From "
  escape_string = ">From "
  user = $local_part
  group = mail
```

Ensure that you provide the proper path to procmail and an appropriate group for running procmail in the definition of the `procmail_pipe` transport.

Next add a new router to direct messages to the `procmail_pipe` transport. This router should be added to the routers section of the configuration file before (or in place of) the `localuser` router, which is usually the last router.

```
procmail:
  driver = accept
```

```
        check_local_user
        transport = procmail_pipe
        cannot_route_message = Unknown user
        no_verify
        no_expn
```

Local addresses that reach the `procmail` router will be accepted and delivered via the `procmail_pipe` transport, which invokes procmail in its role as a local delivery agent.

> After any change to the Exim configuration file, you must send a SIGHUP signal to the Exim daemon to cause it to reread the configuration file. You can test configuration changes before you do this by running `exim -bV`.

Next, configure procmail to invoke SpamAssassin. If you want to invoke SpamAssassin on behalf of every user, do so by editing the */etc/procmailrc* file. Example 8-1 shows an */etc/procmailrc* that invokes SpamAssassin.

Example 8-1. A complete /etc/procmailrc

```
DROPPRIVS=yes
PATH=/bin:/usr/bin:/usr/local/bin
SHELL=/bin/sh

# Spamassassin
:0fw
* <300000
|/usr/bin/spamassassin
```

If you run `spamd`, replace the call to `spamassassin` in *procmailrc* with a call to `spamc` instead. Using `spamc`/`spamd` significantly improves performance on most systems but makes it more difficult to enable users to write their own rules.

Spam-Checking All Incoming Mail

If you want to set up a spam-checking gateway for all recipients, local or not, you need a way to perform spam-checking as mail is received, before final delivery. Exim provides three different ways to do this: via routers, via exiscan, and via defining a `local_scan( )` function.

In a router-based configuration, SpamAssassin is invoked after Exim has received a message during the process of routing each delivery address in the message. If the message is destined for a local user, SpamAssassin can use per-user preference files; if the message will be relayed to a remote user, SpamAssassin still checks the message using sitewide settings. In this configuration, SpamAssassin may be invoked several times for each message received (once for each message recipient). Figure 8-1 illustrates this configuration.

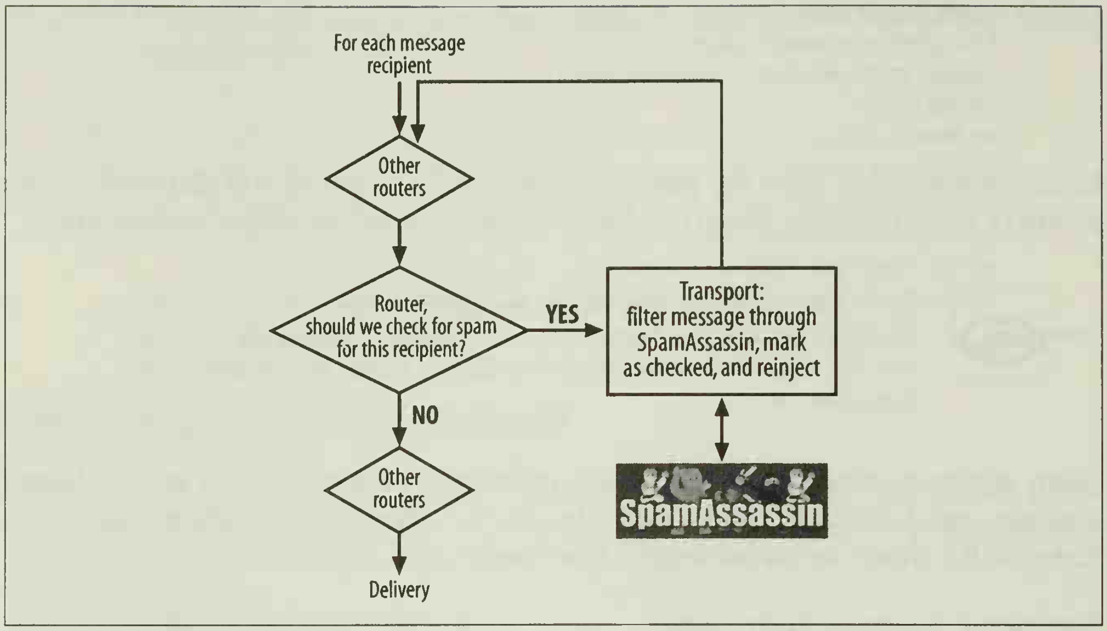

For each message recipient
Other routers
Router, should we check for spam for this recipient?
YES
Transport: filter message through SpamAssassin, mark as checked, and reinject
NO
Other routers
SpamAssassin
Delivery

Figure 8-1. A router-based configuration for spam-checking in Exim

In an *exiscan* configuration, Exim invokes SpamAssassin during the SMTP transaction by means of a new ACL. Messages that SpamAssassin considers spam can be rejected before the SMTP transaction is complete, or accepted and tagged. However, you cannot use per-user preferences in this configuration without negatively impacting performance. Figure 8-2 illustrates this approach.

In a configuration using `local_scan( )`, Exim invokes SpamAssassin during the SMTP transaction when it calls the `local_scan( )` function for the incoming message. The message can be accepted or rejected in the SMTP transaction; if `local_scan( )` accepts the message, tagging headers can be added. Other interesting effects, including teergrubing—responding very slowly during the SMTP transaction when spam is detected in order to tie up the spammer's MTA—are possible with this approach, but it is difficult to use per-user preferences in this configuration. Figure 8-3 illustrates this approach.

Each of these methods is described in detail in the following sections.

Using Routers and Transports

You can configure Exim to pass all incoming mail through SpamAssassin by writing a transport that pipes messages to SpamAssassin and then reinjects them into Exim, and a router that directs messages to the transport. To prevent the reinjected messages from being spam-checked again, you can set their `$received_protocol` to indicate they've been checked when you reinject them, and use the `$received_protocol` value as a condition to determine whether or not the router will send them for checking.

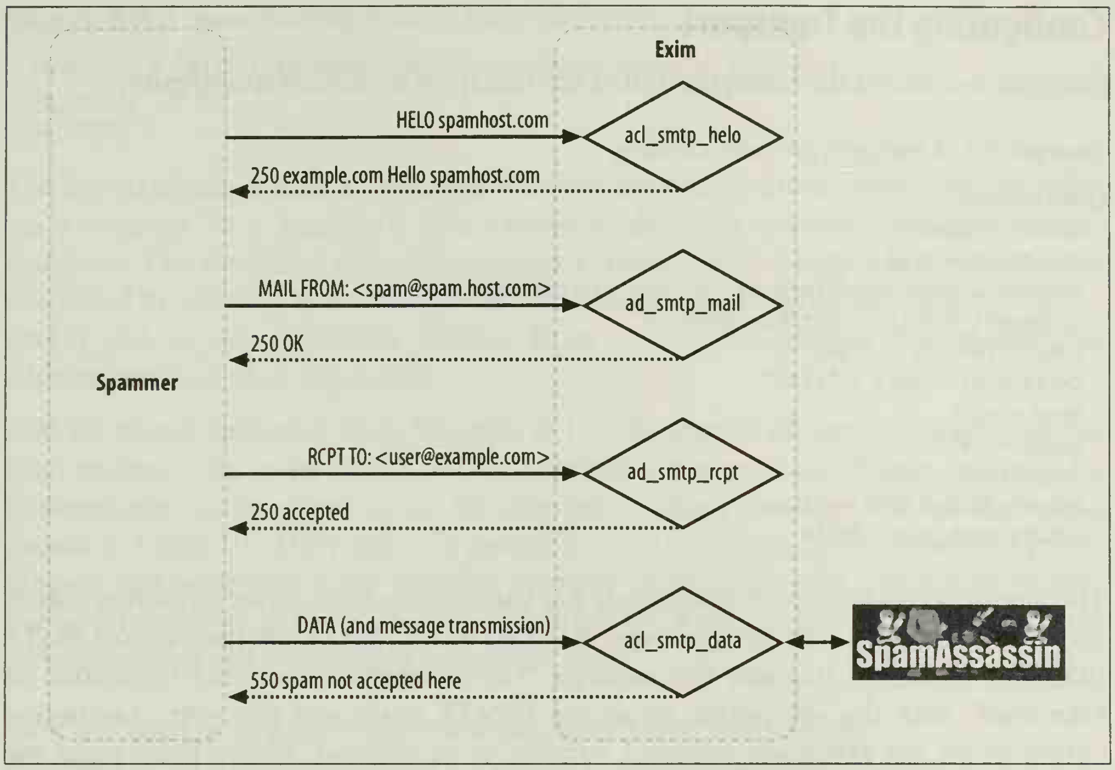

Exim
Spammer
HELO spamhost.com
acl_smtp_helo
250 example.com Hello spamhost.com
MAIL FROM: <spam@spam.host.com>
ad_smtp_mail
250 OK
RCPT TO: <user@example.com>
ad_smtp_rcpt
250 accepted
DATA (and message transmission)
acl_smtp_data
SpamAssassin
550 spam not accepted here

Figure 8-2. An exiscan-based configuration for spam-checking in Exim

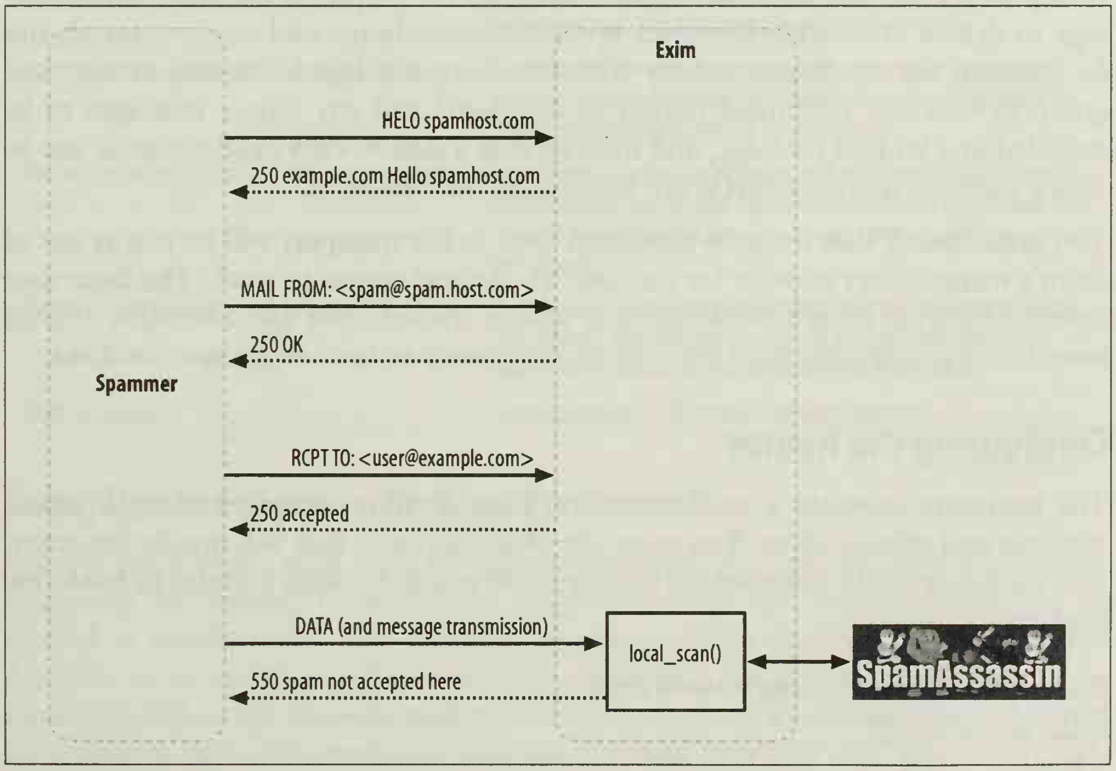

Exim
Spammer
HELO spamhost.com
250 example.com Hello spamhost.com
MAIL FROM: <spam@spam.host.com>
250 OK
RCPT TO: <user@example.com>
250 accepted
DATA (and message transmission)
local_scan()
SpamAssassin
550 spam not accepted here

Figure 8-3. A local_scan()-based configuration for spam-checking in Exim

Configuring the Transport

Example 8-2 shows the configuration of the transport in */usr/exim/configure*.

Example 8-2. A transport for spam-checking

```
spamassassin:
  driver = pipe
  use_bsmtp = true
  command = /usr/exim/bin/exim -bS -oMr sa-checked
  transport_filter = /usr/bin/spamc -f
  home_directory = "/tmp"
  current_directory = "/tmp"
  user = exim
  group = exim
  log_output = true
  return_fail_output = true
  return_path_add = false
```

The `spamassassin` transport in Example 8-2 uses Exim's `pipe` driver to deliver a message to a command. The example specifies that Exim should use the batched SMTP (BSMTP) format to transmit the message. The command is another invocation of `exim` itself, with the `-bS` option to accept BSMTP input and the `-oMr sa-checked` option to set the `$received_protocol` variable to `sa-checked`. Before Exim pipes the message to the command, it filters the message through the program specified by `transport_filter`—in this case, `spamc`—and uses the output of the filter as the message to deliver. The other transport options provide home and working directories for running the command, specify that the command should be run as user and group *exim*, cause command output to be logged and any failure messages to be included in a bounce message, and indicate that a *Return-Path* header should not be added (because this transport is not performing final delivery).

You must specify that the `exim` command used in the transport will be run as one of Exim's trusted users in order for the `-oMr sa-checked` option to work. The Exim user (specified during Exim's installation) is always trusted. You can add other trusted users in the configuration file with the `trusted_users` or `trusted_groups` directives.

Configuring the Router

The transport provides a mechanism for Exim to filter messages through SpamAssassin and reinject them. You must also define a router that will invoke this transport during delivery. Example 8-3 displays a definition for such a router in */usr/exim/configure*.

Example 8-3. A spam-checking router in Exim

```
spamassassin_router:
  driver = accept
  transport = spamassassin
```

Example 8-3. A spam-checking router in Exim (continued)

```
  condition = "${if {!eq {$received_protocol}{sa-checked}} {1} {0}}"
  no_verify
  no_expn
```

The `spamassassin_router` in Example 8-3 uses the `accept` driver, which simply delivers a message to a transport. The `transport` directive specifies our `spamassassin` transport. The `condition` directive prevents a spam-checking loop when messages are reinjected by insuring that the value of `$received_protocol` is not `sa-checked`. The `no_verify` and `no_expn` directives instruct Exim to skip this router when performing address verification or expansion.

Add the router definition from Example 8-3 to the section of */usr/exim/configure* that lists routers. The order of the router definitions is significant. Where you add the `spamassassin_router` router in the list determines which messages will be checked, as shown in Table 8-1. Most sites will probably want to add the router between `system_aliases` and `userforward` (or possibly between `userforward` and a procmail router), but spam-checking gateways are likely to need the router before `dnslookup` as nearly all of their mail will be destined for remote sites.

Table 8-1. Effect of the position of spamassassin_router in the Exim router list

Position	Effect
First in the list (before `dnslookup`)	SpamAssassin invoked on all messages, including local deliveries, outgoing messages, and messages relayed to remote hosts.
Between `dnslookup` and `system_aliases`	SpamAssassin invoked on messages with addresses in locally hosted domains. System aliases and user *.forward* files will receive messages already spam-checked (and can act on tagging).
Between `system_aliases` and `userforward`	SpamAssassin invoked on messages with addresses in locally hosted domains, unless system alias file redirected them to a remote host. User *.forward* files will receive messages already spam-checked (and can act on tagging).
Between `userforward` and `localuser`	SpamAssassin invoked only on messages that will be delivered locally. User *.forward* files will receive messages without spam-checking. Spam-checked messages will be delivered to local mailbox.
After `localuser`	Too late! Messages will already have been delivered.

Using Per-User Spam-Checking Preferences

Because Exim routes each delivery address separately, you can configure it to behave differently for messages that will be delivered locally and messages that will be relayed to remote hosts. You can take advantage of this flexibility to direct SpamAssassin to use per-user preferences when checking a message that is destined for a local user and to use sitewide preferences when checking a message that is destined for a remote user. This approach requires a second transport and a second router.

Add another transport such as that shown in Example 8-4 to your Exim configuration file.

Example 8-4. Transport for local spam-checking in Exim

```
spamassassin_local:
  driver = pipe
  use_bsmtp = true
  command = /usr/exim/bin/exim -bS -oMr sa-checked
  transport_filter = /usr/bin/spamc -f -u $local_part
  home_directory = "/tmp"
  current_directory = "/tmp"
  user = exim
  group = exim
  log_output = true
  return_fail_output = true
  return_path_add = false
```

The key addition in the spamassassin_local transport is the use of spamc's -u *user* command-line option to specify the user on whose behalf spamc is running. spamc will convey the username to spamd, which will examine the user's *.spamassassin/user_prefs* file for preferences.

For this transport to work, spamd must be able to read users' preference files. Because you should run spamd under a dedicated user and group, this user or group must be able to search the *.spamassassin* subdirectory of each user's home directory and read the *user_prefs* file. (You may instead run spamd as *root*, but using a dedicated user is a better security practice.)

You must not invoke spamd with the --nouser-config or --auth-ident options when using this transport. If you use --nouser-config, spamd will ignore spamc's -u argument, and user preferences will not be examined. If you use --auth-ident, spamd will attempt to confirm that spamc is being run by the user given in its -u argument. Because Exim runs as its own user, the authentication will fail and spamd will refuse to look up user preferences.

Next, add a router that uses the spamassassin_local transport, as shown in Example 8-5.

Example 8-5. A spam-checking router with user preferences in Exim

```
spamassassin_local_router:
  driver = accept
  transport = spamassassin_local
  condition = "${if {!eq {$received_protocol}{sa-checked}} {1} {0}}"
  no_verify
  no_expn
```

You should also modify spamassassin_router to limit its use to non-local domains. This modification is shown in Example 8-6.

Example 8-6. A spam-checking router for non-local domains in Exim

```
spamassassin_router:
  driver = accept
  transport = spamassassin
  domains = ! +local_domains
  condition = "${if {!eq {$received_protocol}{sa-checked}} {1} {0}}"
  no_verify
  no_expn
```

Arrange the routers in the following order:

1. spamassassin_router, to perform spam-checking for messages addressed to remote domains
2. dnslookup, to route messages addressed to remote domains via SMTP
3. system_aliases, to redirect messages with addresses in */etc/aliases*
4. spamassassin_local_router, to perform spam-checking for messages addressed to local users with the per-user preferences of the local user (who may, however, choose to forward the tagged message elsewhere)
5. userforward, to redirect messages with addresses in user *.forward* files
6. localuser, to route messages via the local delivery agent

To illustrate how this approach functions, consider an Exim system running on *mail.example.com* and configured to relay messages for *example.com* to an internal mail server. On *mail.example.com*, *postmaster* is an alias for the local user *chris*. When a spammer sends a message addressed to *sam@example.com* and *postmaster@mail.example.com*, Exim passes each address through its list of routers. *sam@example.com* is routed by spamassassin_router, so a copy of the message is tagged by SpamAssassin using its sitewide configuration and then reinjected. The reinjected message bypasses spamassassin_router and is routed by dnslookup, which queues it for remote delivery. Meanwhile, *postmaster@mail.example.com* is destined for a local domain and bypasses both spamassassin_router and dnslookup. The system_aliases router rewrites the address to *chris@mail.example.com*, which Exim then begins routing. This address bypasses spamassassin_router, dnslookup, and system_alias and is routed by spamassassin_local_router, which tags a copy of the message using *chris*'s SpamAssassin preferences and reinjects it. The reinjected message bypasses spamassassin_router, dnslookup, system_alias, and spamassassin_local_router, and assuming *chris* does not have a *.forward* file, Exim delivers it to *chris*'s local mailbox. Figure 8-4 illustrates this process.

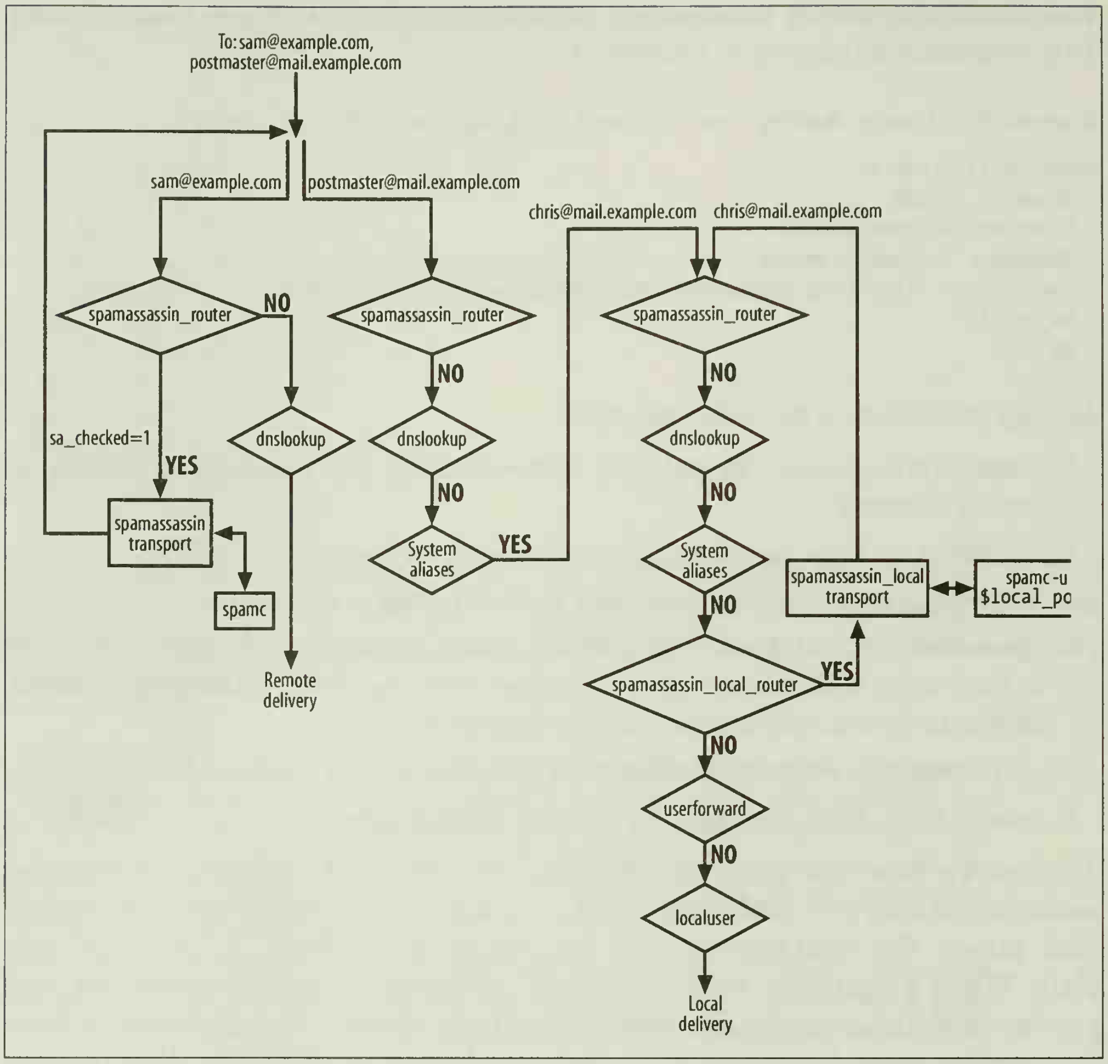


Figure 8-4. Exim router lookups during delivery

Using exiscan

One of Exim's most powerful and flexible features is its ACL system. Each ACL is a set of rules or tests that Exim performs when receiving a message; for example, an ACL is available for each stage of the SMTP transaction (start of connection, after HELO, after MAIL FROM, etc.). Rules are evaluated in order until one matches, and the associated action is then performed. Actions can include allowing the transaction to proceed, deferring the transaction, rejecting the transaction, ignoring the transaction, adding warning headers to the message, or dropping the connection altogether. If no rule matches, the ACL rejects the corresponding portion of the SMTP transaction.

exiscan is a set of patches for Exim that introduces the ability to invoke SpamAssassin in the acl_smtp_data ACL that Exim consults after the DATA step of an SMTP transaction. You can download exiscan from *http://duncanthrax.net/exiscan-acl/*; many precompiled versions of Exim (e.g., in Linux distributions) have the patch already applied. exiscan's new ACL actions also include blocking MIME attachments, virus-checking, and checking headers against regular expressions.

Installing exiscan

If you're not using a version of Exim that has exiscan already compiled in, you should download the exiscan patch file and apply it to your Exim source code with the GNU patch program. Example 8-7 shows the patch process, assuming that both the Exim source code and the patch are in */usr/local/src*. Stop and restart Exim after you install the patched version.

Example 8-7. Patching the Exim source code with exiscan

```
$ cd /usr/local/src/exim-4.30
$ patch -p1 -s < ../exiscan-acl-4.30-14.patch
$ rm -rf build-*
$ make
...Compilation messages...
$ su
Password: XXXXXXXX
# make install
```

The rm -rf build-* command removes any old Exim build directories that may be present and forces Exim's *Makefile* to recreate them and repopulate them with symbolic links to source code files. This is important, because exiscan adds new source code files that would otherwise not have links in the build directory.

Writing acl_smtp_data

exiscan extends Exim's ACL language by adding a new rule, spam, that makes a connection to spamd to request a message check on behalf of a specified user and returns true if the message would exceed the user's SpamAssassin spam threshold. Example 8-8 shows a simple acl_smtp_data that uses the spam condition to add an *X-Spam-Flag: YES* header to spam messages.

Example 8-8. Adding an X-Spam-Flag header with exiscan

```
acl_smtp_data:

  warn  message = X-Spam-Flag: YES
        spam = nobody
```

In this ACL, the condition `spam = nobody` invokes `spamc` as the user *nobody*. If the message's spam score exceeds *nobody*'s threshold, Exim takes the `warn` action, adding the *X-Spam-Flag* header. Similarly, the following ACL rule will generate a second *Subject* header with a spam tag for spam messages.

```
warn  message = Subject: *SPAM* $h_Subject
      spam = nobody
```

ACLs can add headers but cannot remove them or modify them *in situ*. To *replace* the *Subject* header with a tagged version, you must add a new header through the ACL (e.g., *X-Spam-Subject*) and direct Exim's system filter to replace the message subject with the new header if it's present. An example of how to do this is included with the exiscan documentation.

The `spam` condition also sets several useful Exim variables as a side effect:

`$spam_bar`
: If SpamAssassin gives a message a positive spam score, exiscan sets this variable to a string of plus (+) characters, with one plus for each point of spam score, up to 50. If SpamAssassin gives a message a negative spam score, exiscan sets this variable to a string of minus characters (–), with one minus for each negative point of spam score. If SpamAssassin gives a message a zero spam score, exiscan sets this variable to a slash (/) character.

`$spam_report`
: The full SpamAssassin report on a message.

`$spam_score`
: The score assigned to a message by SpamAssassin.

`$spam_score_int`
: The score assigned to a message by SpamAssassin multiplied by 10. exiscan stores this variable in the message's spool file, so Exim can use this value in later processing (e.g., in routers) to handle high-scoring messages differently than low-scoring messages.

These variables can be used with `warn` or `deny` actions to implement several kinds of spam policies. Example 8-9, adapted from the exiscan documentation, shows how you can direct Exim to add an *X-Spam-Score* header for all messages, to add an *X-Spam-Report* header for spam, and to reject a message completely if the spam score is higher than 12.

Example 8-9. Spam policies with exiscan

```
warn  message = X-Spam-Report: $spam_report
      spam = nobody

warn  message = X-Spam-Score: $spam_score ($spam_bar)
      spam = nobody:true
```

Example 8-9. Spam policies with exiscan (continued)

```
deny   message = This message scored $spam_score spam points.
       spam = nobody
       condition = ${if >{$spam_score_int}{120}{1}{0}}
```

The first rule performs spam-checking and adds the *X-Spam-Report* header if a message exceeds the spam threshold. exiscan caches the spam-checking results, so future calls to the spam condition for this message will not actually recheck the message. The second rule uses the :true option, which causes the condition to be evaluated as true regardless of the results of the spam check. Accordingly, Exim will add an *X-Spam-Score* header to all messages. Finally, Exim executes the deny action (refusing the message with the given text added to the SMTP rejection response) if the $spam_score_int is greater than 120 (which corresponds to a SpamAssassin score greater than 12.0).

Using Per-User Preferences

Because exiscan checks messages for spam just once—at message receipt after the SMTP DATA command—it's difficult to use SpamAssassin's per-user preference files. Messages may have multiple recipients, some of whom are not local, and exiscan will not be able to determine whose preferences should be used.

You can continue to use per-user preferences with exiscan in two ways, but each comes at a performance cost.

- You can ensure that each email message will have only a single recipient by writing an ACL for the SMTP RCPT TO phase that defers all recipients except the first one. The sending MTA will retry delivery to the deferred recipients but may not do so immediately. As a result, some copies of messages with multiple recipients may be significantly delayed. The exiscan documentation includes an example of how to do this.
- You can use exiscan to perform initial spam-checking and refuse messages with high scores, and then use the router/transport approach described earlier to reinvoke SpamAssassin on the remaining messages for local recipients. This approach results in an extra spamd connection for each message with a local recipient but might be worthwhile if exiscan can refuse enough very obvious spam sent to multiple recipients.

Using sa-exim

Exim calls its local_scan( ) function once just before accepting a message (via SMTP or from a local process). By default, this function does nothing—the implementation of the function in Exim's source code simply instructs Exim to accept the message. What makes local_scan( ) powerful is that you can replace Exim's version with your own code to perform custom message-checking. This function can be a good place to perform spam-checking.

Even better, you don't have to write a new `local_scan()` yourself if you want to invoke SpamAssassin. Marc Merlin has written one for you: sa-exim. sa-exim invokes `spamc` in its `local_scan()` function and can thus take advantage of all of `spamd`'s configuration options. This section describes the installation and configuration of sa-exim. You can download it at *http://sa-exim.sf.net*. It requires Exim 4.11 or later.

Buiding sa-exim for Static Integration

Once you've unpacked the source code, you can choose one of two approaches to integrating sa-exim with Exim. This section focuses on *static integration*, which embeds sa-exim within Exim at compile time. The examples in this section assume you have unpacked Exim's source code in */usr/local/src/exim-4.30* and sa-exim's in */usr/local/src/sa-exim-3.1*.

> Whichever approach you choose for integrating sa-exim, be sure that `LOCAL_SCAN_HAS_OPTIONS` has not been set to yes in Exim's *Local/Makefile* (it is not set by default).

To use the static integration approach, you edit sa-exim's *sa-exim.c* file, then replace Exim's *src/local_scan.c* file with sa-exim's *sa-exim.c* file, copy sa-exim's *sa-exim.h* file to the same location, and recompile (and reinstall) Exim. The `local_scan()` function in *sa-exim.c* replaces the default function.

Two macro definitions in *sa-exim.c* must be edited. They appear in the code under the comment "Compile time config values" and provide the location of `spamc` (by default, */usr/bin/spamc*) and sa-exim's own configuration file (by default, */etc/exim4/sa-exim.conf*, but you might change this location to */usr/exim/sa-exim.conf* or */etc/sa-exim.conf* as suits your system).

```
$ cd /usr/local/src/sa-exim-3.1
...Edit sa-exim.c in your favorite editor...
$ make sa-exim.h
echo "char *version=\"`cat version` (built `date`)\";" > sa-exim.h
$ cp sa-exim.c ../exim-4.30/src/local_scan.c
$ cp sa-exim.h ../exim-4.30/src
$ cd ../exim-4.30
$ make
$ su
Password: XXXXXXXX
# make install
```

The static integration approach is easy but requires you to recompile Exim whenever you want to update sa-exim.

Building sa-exim for Dynamic Integration

Using the *dynamic integration* approach, you patch Exim to allow the `local_scan( )` function to be dynamically loaded at runtime, and you compile sa-exim as a dynamically loadable executable. Many packaged versions of Exim are distributed with the dynamic loading patch already applied, but sa-exim includes two versions of the patches by David Woodhouse that you can apply to your Exim source code yourself. Use *localscan_dlopen_up_to_4.14.patch* to patch Exim versions 4.11 to 4.14; use *localscan_dlopen_exim_4.20_or_better.patch* to patch Exim 4.20 and later versions. Example 8-10 illustrates the patch process.

Example 8-10. Patching Exim to support dynamic loading

```
$ cd /usr/local/src/exim-4.30
$ patch -p1 < ../sa-exim-3.1/localscan_dlopen_exim_4.20_or_better.patch
patching file src/EDITME
Hunk #1 succeeded at 505 (offset 117 lines).
patching file src/config.h.defaults
Hunk #1 succeeded at 20 (offset 3 lines).
patching file src/globals.c
Hunk #1 succeeded at 108 (offset 5 lines).
patching file src/globals.h
Hunk #1 succeeded at 72 (offset 5 lines).
patching file src/local_scan.c
patching file src/readconf.c
Hunk #1 succeeded at 224 (offset 42 lines).
$ make
$ su
Password: XXXXXXXX
# make install
```

After installing the patched Exim, compile sa-exim as a dynamically loadable object file by editing its *Makefile*. Check that the definitions of `CC`, `CFLAGS`, and `LDFLAGS` are suitable for building a shared object file with your compiler. Set the following macros in the *Makefile*:

`SACONF`
: The path where you will locate sa-exim's configuration file (e.g., */etc/exim4/sa-exim.conf*, */usr/exim/sa-exim.conf*, or whatever suits your system)

`SPAMC`
: The location of `spamc` (e.g. */usr/bin/spamc*)

`EXIM_SRC`
: The path to the Exim source code's *src* directory (e.g., */usr/local/src/exim-4.30/src)*

Run `make` to compile sa-exim; `make` should produce the shared object files *sa-exim-3.1.so* and *accept.so*. The former is the sa-exim replacement for the `local_scan( )` function. The latter is a replacement for `local_scan( )` that simply accepts all messages; you can use *accept.so* to test that dynamic loading works properly without the complexities of sa-exim.

Copy these shared object files to an appropriate Exim directory (e.g., */usr/exim* or */usr/exim/libexec*), and add the following lines to the beginning of Exim's configuration file:

```
local_scan_path = /usr/exim/accept.so
#local_scan_path = /usr/exim/sa-exim-3.1.so
```

Restart Exim, and confirm that messages are being received. After you finish configuring sa-exim, edit Exim's configuration file again, comment out the *accept.so* line, uncomment the *sa-exim.so* line, and restart Exim again to activate sa-exim.

Configuring SpamAssassin for sa-exim

sa-exim invokes SpamAssassin using `spamc`, so you must be running the `spamd` daemon to use sa-exim.

sa-exim behaves as you'd expect with most of the settings you'd be likely to have in your sitewide configuration file (typically */etc/mail/spamassassin/local.cf*). One that requires particular care, however, is the `report_safe` setting.

If you set `report_safe` to 0, SpamAssassin only adds spam-tagging headers and does not modify the body of messages. This setting works with sa-exim without any additional configuration and provides the fastest message-checking performance.

If you prefer to have SpamAssassin modify the body of the message to add its report and convert the original message into an attachment, you can set `report_safe` to 1 (include original message as *message/rfc822* attachment) or 2 (include original message as *text/plain* attachment). In this case, you have to set the `SARewriteBody` variable in *sa-exim.conf* (described in the next section). Because sa-exim must read the modified body back from SpamAssassin, message-checking will be slightly slower than with `report_safe 0`. In addition, if you perform message-archiving, the archives will contain the SpamAssassin-modified message.

Finally, ensure that `spamd` is not being invoked with the `--create-prefs` option, as it should run as an unprivileged user and be unable to create user preference files anyway. You may wish to include the `--nouser-config` option as well.

Configuring sa-exim

You configure sa-exim by editing its *sa-exim.conf* configuration file. During the build of sa-exim, you should have specified a location for this file. Begin configuration by copying the *sa-exim.conf* file included with the sa-exim source code to this location. Edit the file to configure sa-exim.

The *sa-exim.conf* file is copiously commented. As the first comment describes, sa-exim is picky about the formatting of options in this file. For example, the following are examples of valid options in *sa-exim.conf*:

```
SApermreject: 12.0
SARewriteBody: 0
# The option below is commented out, and thus not set
#SApermrejectsave: /var/spool/exim/SApermreject
```

But none of this next set of options are valid:

```
# No spaces are allowed before the colon! One and only one is required after!
Sapermreject :12.0
# Only thresholds may be floating point numbers!
SARewriteBody: 0.0
# This sets the option, with an empty value! Not the way to unset it!
SApermrejectsave:
```

Later definitions of the same option override earlier ones.

The configuration file determines how sa-exim handles spam: sa-exim can accept messages (returning a 2xx SMTP code), accept and discard messages, temporarily fail messages (returning a 4xx SMTP code), reject messages (returning a 5xx SMTP code), or perform teergrubing during the SMTP connection. For each sa-exim action, you can control at what spam threshold the action is triggered, whether a message that triggered the action should be saved to an archive directory, and the location of the archive directory. sa-exim usually names files in the archive directory by concatenating the time (in seconds since 00:00:00 UTC on January 1, 1970) and the value of the *Message-ID* header of a given message.

Teergrubing

One interesting strategy that sa-exim provides for dealing with spam is *teergrubing*. Teergrube is the German word for "tar pit," and teergrubing is the practice of identifying spam while an SMTP connection is in progress and slowing down the SMTP connection. The goal is to tie up the spammers' mail server for as long as possible, reducing the rate at which they can spam.

If you want to interfere with spammers' operations, sa-exim's teergrubing features may be for you. Note that you also tie up your own SMTP server processes while connections are maintained, but these processes will consume few resources as they'll primarily be sleeping.

The following sections examine the options in the *sa-exim.conf* configuration file.

Choosing messages on which to run SpamAssassin

The `SAEximRunCond` option specifies an Exim conditional expression that will be evaluated to determine whether SpamAssassin should be invoked on a message. To disable SpamAssassin, comment the option out or set its value to 0. To enable SpamAssassin on all messages, set the option's value to 1. The configuration file presents an example of how you can set this variable to check all messages except those originating from the local host or those with an *X-SA-Do-Not-Run: Yes* header:

```
SAEximRunCond: ${if and {{def:sender_host_address} {!eq {$sender_host_address}{127.0.
0.1}} {!eq {$h_X-SA-Do-Not-Run:}{Yes}} } {1}{0}}
```

Choosing messages on which to take antispam actions

The `SAEximRejCond` option specifies an Exim conditional expression that will be evaluated to determine whether sa-exim should take actions on messages that SpamAssassin considers spam. By disabling the option, you can have messages checked by SpamAssassin (and tagged, if appropriate) but unconditionally accepted. The configuration file provides an example in which actions are taken on all spam messages except those with an *X-SA-Do-Not-Rej: Yes* header:

```
# X-SA-Do-Not-Rej should be set as a warn header if mail is sent to postmaster
# and abuse (in the RCPT ACL), this way you're not bouncing spam abuse reports
# sent to you
SAEximRejCond: ${if !eq {$h_X-SA-Do-Not-Rej:}{Yes} {1}{0}}
```

The *X-SA-Do-Not-Run* and *X-SA-Do-Not-Rej* headers can be added by the `acl_smtp_rcpt` ACL in Exim's own configuration file, using directives such as these:

```
warn    message       = X-SA-Do-Not-Run: Yes
        hosts         = +relay_from_hosts

warn    message       = X-SA-Do-Not-Run: Yes
        authenticated = *

warn    message       = X-SA-Do-Not-Rej: Yes
        local_parts   = postmaster:abuse
```

These ACL directives will add *X-SA-Do-Not-Run* headers to messages from authenticated senders or from hosts from which Exim should relay messages, and will add *X-SA-Do-Not-Rej* headers to messages to *postmaster* or *abuse*. The *X-SA-Do-Not-Run* header should be removed before messages are relayed to remote hosts; add a `headers_remove` directive in the definition of the `remote_smtp` transport:

```
remote_smtp:
  driver = smtp
  headers_remove = "X-SA-Do-Not-Run"
```

You may wish to use different header names or values to prevent spammers from guessing your header and adding it to their spam messages to bypass sa-exim.

Limiting how much of the message is fed to SpamAssassin

`SAmaxbody` determines how many bytes of a message body sa-exim will feed to Spam-Assassin for checking; it defaults to 256,000. If `SATruncBodyCond` evaluates to a false value, messages larger than `SAmaxbody` are not scanned at all. If `SATruncBodyCond` evaluates to a true value, such messages are truncated, and the first `SAmaxbody` bytes are scanned. This is generally not a good idea because proper MIME message formatting requires a closing MIME boundary string at the end of a message, and if Spam-Assassin receives a partial body missing this string, it may complain that the message is misformatted.

Allowing SpamAssassin to rewrite message bodies

If you set SpamAssassin's `report_safe` option to 1 or 2 (asking SpamAssassin to rewrite message bodies), you must set the `SARewriteBody` variable to 1.

Archiving messages when actions are taken

Archiving message bodies preserves copies of messages in case they are needed later, and archived messages can be used as a quarantine system.

The value of `SAmaxarchivebody` determines the amount of a message (in bytes) to save when archiving messages after taking action on them. It defaults to 20,971,520 (20MB), which is a reasonable value. Similarly, `SAerrmaxarchivebody` determines the number of bytes of a message to save when a message causes an error in sa-exim. It defaults to 1,073,741,824 (1GB).

If `SAPrependArchiveWithFrom` is set to 1, sa-exim will add fake *From* lines to the beginning of archived messages so that the archive file will be in standard *mbox* format. This is usually desirable because it's easy to use most mail readers to examine an *mbox* file.

Passing SMTP senders and recipients to SpamAssassin

Because sa-exim is invoked at the end of the SMTP DATA step, it has access to the list of recipients provided in the SMTP `RCPT` commands from the sending MTA. If you set `SAmaxrcptlength` to a value higher than 0, sa-exim adds an *X-SA-Exim-Rcpt-To* header containing the list of recipients as long as the list doesn't exceed the smaller of `SAmaxrcptlength` bytes or 8 KB.

sa-exim also has access to the SMTP `MAIL FROM` command and adds the SMTP sender to the message in the *X-SA-Exim-Mail-From* header

The recipient list can be useful to SpamAssassin, as messages with a large number of recipients might be more likely to indicate spam, and the true list of recipients may not appear in the message *To* and *Cc* headers. Similarly, knowing the SMTP sender might help identify a known spammer or a spammer using an invalid sender address. By setting the `SAaddSAEheaderBeforeSA` option to 1, you direct sa-exim to add these headers before invoking SpamAssassin on a message, which is the default. Set SAaddSAEheaderBeforeSA to 0 if you prefer SpamAssassin to see messages with no sa-exim headers added.

> Adding the *X-SA-Exim-Rcpt-To* header will expose recipients who were blind carbon copied (Bcc) and foil other legitimate strategies to keep the list of message recipients private. You should remove this header in your message transports (using the remove_headers directive) before messages are delivered.
>
> If you allow SpamAssassin to rewrite message bodies, however, the headers will be encapsulated in the body of spam messages and cannot be removed. This may be acceptable to you, as these messages are spam anyway, but the privacy risk in the case of a false positive should be considered.

Setting a timeout on spamc

sa-exim must wait for spamc to check messages but should not wait forever. By setting SAtimeout to a value in seconds, you ensure that if spamc should fail to check a message in a reasonable time, the message will be accepted. If you set SAtimeout to 0 (or to more than 300 seconds), Exim itself will interrupt a spamc run after five minutes, but it will cause the SMTP connection to return a temporary failure for the message, instead of accepting it. I recommend that you set SAtimeout and use a value between 60 and 240 seconds.

If a message is accepted due to a spamc timeout, and you set SAtimeoutsave to the absolute path of a directory, the message will be saved in that directory so you can see the impact of your SAtimeout settings. The directory must be writable by the Exim user; if it does not exist, sa-exim will attempt to create it.

You can limit which of these messages are saved by defining SAtimeoutSavCond to an Exim conditional expression. When spamc times out checking a message and the conditional expression returns a true value, the message will be saved. The default SAtimeoutSavCond is 1, which saves all messages when spamc times out.

Handling messages that cause sa-exim errors

Because sa-exim is a robust framework, it considers the possibility that a message might cause an error in sa-exim itself and provides the ability to handle such messages. If a message causes an error, and you set SAerrorsave to the absolute path of a directory, the message will be saved in that directory. The directory must be writable by the Exim user; if it does not exist, sa-exim will attempt to create it.

You can limit which error-causing messages are saved by defining SAerrorSavCond to an Exim conditional expression. If an error occurs and the conditional expression returns a true value, the message will be saved. The default SAerrorSavCond is 1, which saves all messages that cause sa-exim errors.

By default, sa-exim will accept messages that cause errors, which prevents mail loss. An alternative is to have sa-exim instruct Exim to temporarily fail such messages, which will cause the sending MTA to queue them and retry delivery later. To temporarily fail messages that cause errors, set `SAtemprejectonerror` to 1. Set the `SAtemprejectonerror variable` to change the message that will be returned to the sending MTA when a message is temporarily failed by setting the `SAmsgerror` variable.

Teergrubing

If you want sa-exim to perform teergrubing of a connection when spam is detected, set the `SAteergrube` variable to the SpamAssassin spam score at or above which teergrubing should take place. If you don't define this variable, sa-exim will not teergrube. See the sidebar "Teergrubing," earlier in this chapter for an explanation of that technique.

Set the `SAteergrubcond` variable to an Exim conditional expression to determine whether teergrubing should be performed when the spam score exceeds the `SAteergrube` threshold; teergrubing will be performed only when the expression evaluates to a true value. Use this variable to prevent teergrubing from affecting you or your secondary mail exchangers. The default *sa-exim.conf* file includes the following example, which prevents teergrubing of connections from 127.0.0.1 and 127.0.0.2:

```
SAteergrubecond: ${if and { {!eq {$sender_host_address}{127.0.0.1}} {!eq {$sender_
host_address}{127.0.0.2}} } {1}{0}}
```

You can configure the teergrube delay—the total amount of time, in seconds, that you want to try to tie up the sending MTA—by setting the `SAteergrubetime` variable. The default is 900 (15 minutes). Every ten seconds during the teergrubing period, sa-exim will transmit SMTP code 451 with the reason given in `SAmsgteergrubewait` (which defaults to "wait for more output"). At the end of the teergrubing period, sa-exim will temporarily fail the message with the reason given in `SAmsgteergruberej` (which defaults to "Please try again later"). sa-exim temporarily fails the messages in the hopes that the sending MTA will later attempt to resend the message and spend more time in the tar pit.

If a message qualifies a connection for teergrubing, and you set `SAteergrubesave` to the absolute path of a directory, the message will be saved in that directory. The directory must be writable by the Exim user; if it does not exist, sa-exim will attempt to create it.

You can limit which of these messages are saved by defining `SAteergrubeSavCond` to an Exim conditional expression. If the conditional expression returns a true value, the message will be saved. The default `SAteergrubeSavCond` is 1, which saves all messages that trigger teergrubing.

Because sa-exim temporarily fails teergrubed mail after the teergrubing period, the sending MTA is likely to resend the same message. If you are saving messages that trigger teergrubing, it could lead to repeatedly saving multiple copies of the same message. To prevent this, set `SAteergrubeoverwrite` to 1 (which is the default), and sa-exim will use only the message ID as the filename when saving teergrubed messages. Because resends should have the same message ID, this will result in a single copy of the message being kept, as older copies are overwritten by newer copies assigned the same filename.

Accepting and discarding spam

If you want sa-exim to accept and discard spam, set the `SAdevnull` variable to the SpamAssassin spam score at or above which messages should be accepted and discarded. If you don't define this variable, sa-exim will not take those actions.

If a message is to be discarded, and you set `SAdevnullsave` to the absolute path of a directory, the message will be saved in that directory. The directory must be writable by the Exim user; if it does not exist, sa-exim will attempt to create it.

You can limit which of these messages are saved by defining `SAdevnullSavCond` to an Exim conditional expression. If the conditional expression returns a true value, the message will be saved. The default `SAdevnullSavCond` is 1, which saves all messages that are discarded.

Rejecting spam

If you want sa-exim to reject spam during the SMTP connection, set the `SApermreject` variable to the SpamAssassin spam score at or above which messages should be rejected. If you don't define this variable, sa-exim will not take this action. You can customize the rejection explanation that is sent along with the SMTP rejection code by setting `SAmsgpermreject`.

If a message is to be rejected, and you set `SApermrejectsave` to the absolute path of a directory, the message will be saved in that directory. The directory must be writable by the Exim user; if it does not exist, sa-exim will attempt to create it.

You can limit which of these messages are saved by defining `SApermrejectSavCond` to an Exim conditional expression. If the conditional expression returns a true value, the message will be saved. The default `SApermrejectSavCond` is 1, which saves all messages that are rejected.

Temporarily failing spam

If you want sa-exim to temporarily fail spam during the SMTP connection, set the `SAtempreject` variable to the SpamAssassin spam score at or above which messages should be temporarily failed. If you don't define this variable, sa-exim will not take this action. You can customize the rejection explanation that is sent along with the SMTP rejection code by setting `SAmsgtempreject`.

If a message is to be temporarily failed, and you set `SAtemprejectsave` to the absolute path of a directory, the message will be saved in that directory. The directory must be writable by the Exim user; if it does not exist, sa-exim will attempt to create it.

You can limit which of these messages are saved by defining `SAtempmrejectSavCond` to an Exim conditional expression. If the conditional expression returns a true value, the message will be saved. The default `SAtemprejectSavCond` is 1, which saves all messages that are temporarily failed.

When sa-exim temporarily fails a message, the sending MTA is likely to resend the same message. If you are saving messages that trigger temporary rejections, this could lead to repeatedly saving multiple copies of the same message. To prevent this, set `SAtemprejectoverwrite` to 1 (which is the default), and sa-exim will use only the message ID as the filename when saving temporarily failed messages. Because resends should have the same message ID, this will result in single copies of messages being kept, as older copies are overwritten by newer copies assigned the same filename.

There are few good reasons to temporarily fail spam. If you do not want to receive spam at all, permanently reject or accept and discard it instead. If you want to tie up spammer MTAs, teergrube instead. sa-exim includes temporary failing for completeness, but I do not recommend its use.

Archiving accepted spam

When sa-exim receives a message that SpamAssassin tags as spam but that does not meet any of the sa-exim action thresholds, sa-exim will accept the (tagged) message and allow it to be delivered to the recipient.

If a message is to be accepted, and you set `SAspamacceptsave` to the absolute path of a directory, the message will be saved in that directory. The directory must be writable by the Exim user; if it does not exist, sa-exim will attempt to create it.

You can limit which of these messages are archived by defining `SAspamacceptSavCond` to an Exim conditional expression. If the conditional expression returns a true value, a message will be archived. The default `SAspamacceptSavCond` is 0, which does not archive any accepted spam messages.

Although this feature is not useful for end users, mail administrators can use it to help decide whether to lower one of the other action thresholds by examining the saved messages. If there are no false positives, you might lower the action thresholds.

Archiving non-spam messages

When sa-exim receives a message that SpamAssassin does not consider spam, sa-exim will (of course) accept the message and allow it to be delivered to the recipient.

If a non-spam message is received, and you set `SAnotspamsave` to the absolute path of a directory, the message will be saved in that directory. The directory must be writable by the Exim user; if it does not exist, sa-exim will attempt to create it.

You can limit which of these messages are saved by defining `SAnotspamSavCond` to an Exim conditional expression. If the conditional expression returns a true value, the message will be saved. The default `SAnotspamSavCond` is 0, which does not save any accepted non-spam messages.

A mail administrator might use this feature to analyze a group of non-spam messages to determine whether SpamAssassin is making too many false negative judgments, but on a busy mail site, saving extra copies of all legitimate incoming mail is probably not a good idea. sa-exim includes this feature primarily for completeness.

Debugging sa-exim

Set the `SAEximDebug` variable to a number between 1 and 9 to enable extra logging; higher numbers produce more debugging output. The distributed *sa-exim.conf* file sets this variable to 1, which will log a notice whenever sa-exim saves a new message to one of its archive directories, invokes `spamc`, rewrites message bodies, or evaluates an Exim conditional expression. Increasing `SAEximDebug` is a good idea, particularly when testing new conditional expressions.

Example 8-11 shows a complete *sa-exim.conf* file (without comments). In this example, sa-exim is configured to reject (but save) messages with spam scores higher than 15.

Example 8-11. A complete sa-exim.conf file

```
# Run SpamAssassin unless the message was submitted locally or the
# X-SA-Do-Not-Run header is set to 'secret'. We configure Exim elsewhere
# to set this header for messages from authenticated senders or hosts
# we relay for
SAEximRunCond: ${if and {{def:sender_host_address} {!eq {$sender_host_address}{127.0.0.
1}} {!eq {$h_X-SA-Do-Not-Run:}{secret}} } {1}{0}}

# Don't take action on messages if X-SpamAssassin-Do-Not-Rej header is set to
# 'secret'. We configure Exim to set this header for messages to the postmaster.
SAEximRejCond: ${if !eq {$h_X-SA-Do-Not-Rej:}{Yes} {1}{0}}

# Feed up to 300Kb to SpamAssassin, and if the message is longer, don't
# bother spam checkign
SAmaxbody: 307200
SATruncBodyCond: 0

# We don't let SpamAssassin rewrite message bodies, so we don't set this
```

Example 8-11. A complete sa-exim.conf file (continued)

```
SARewriteBody: 0

# I prefer to avoid the X-SA-Exim-Rcpt-To header, for privacy reasons.
SAmaxrcptlistlength: 0

# Allow spamc 2 minutes for each message. If it times out, don't bother
# saving messages, just accept them.
SAtimeout: 120
SAtimeoutsave:
SAtimeoutSavCond: 0

# Do save messages that cause an error in sa-exim, but accept them
SAerrorsave: /var/spool/exim/SAerrorsave
SAerrorSavCond: 1
SAtemprejectonerror: 0

# Reject messages with SpamAssassin scores of 15 or higher, but save a
# copy of them.
SApermreject: 15.0
SApermrejectSavCond: 1
SApermrejectsave: /var/spool/exim/SApermreject
```

Using Per-User Preferences

Like exiscan, sa-exim checks messages for spam just once—at message receipt after the SMTP DATA command. And like exiscan, it's difficult to use SpamAssassin's per-user preference files with sa-exim. Messages may have multiple recipients, some of whom are not local, and sa-exim will not be able to determine whose preferences should be used.

You can use per-user preferences with sa-exim in the same ways as you can with exiscan, and with the same performance costs:

- You can ensure that each email message will have only a single recipient by writing an ACL for the SMTP RCPT TO phase that defers all recipients except the first one. The sending MTA will retry delivery to the deferred recipients but may not do so immediately. As a result, some copies of messages with multiple recipients may be significantly delayed.
- You can use sa-exim to perform initial spam-checking and refuse messages with high scores, and then use the router/transport approach described earlier to reinvoke SpamAssassin on the remaining messages for local recipients. This approach results in an extra spamd connection for each message with a local recipient but might be worthwhile if sa-exim can refuse enough very obvious spam sent to multiple recipients.

Building a Spam-Checking Gateway

Any of the approaches discussed earlier can form the basis for a spam-checking Exim gateway. exiscan or sa-exim will likely yield better performance than a router/transport approach, and I recommend using them unless you need per-user preferences and are prepared to configure `spamd` to perform SQL-based lookups. The remainder of this chapter explains how to configure an Exim-based gateway for routing messages and how to add sitewide Bayesian filtering and autowhitelisting.

Routing Email Through the Gateway

Once you have Exim receiving messages for the local host and performing SpamAssassin checks on them using any of the methods outlined earlier, you can start accepting email for your domain and routing it to an internal mail server after spam-checking. Figure 8-5 illustrates this topology.

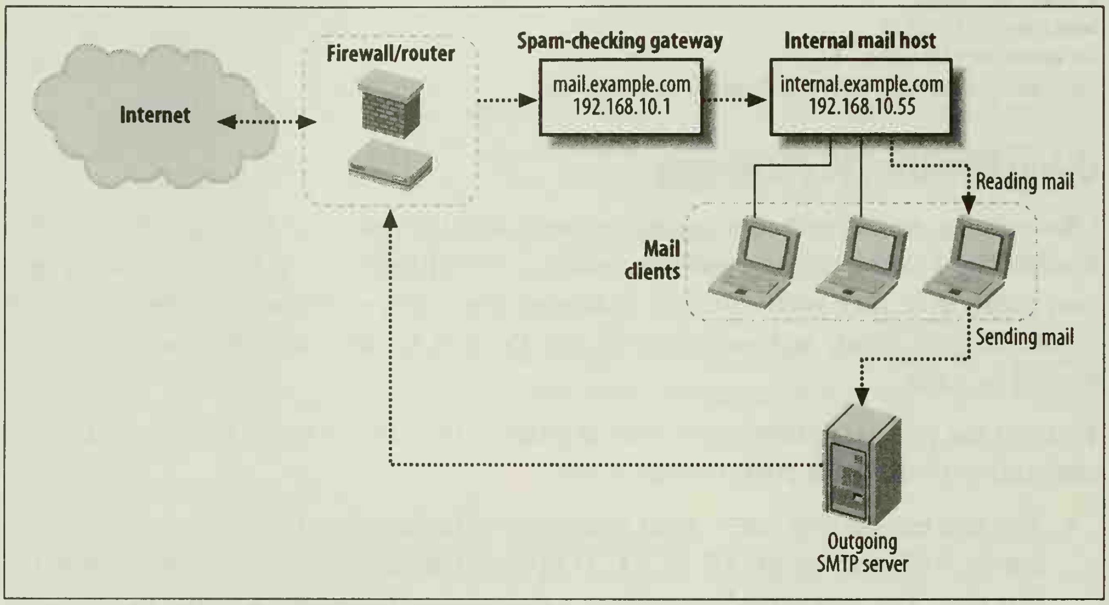


Figure 8-5. Spam-checking gateway topology

Exim domain lists

To configure Exim to relay incoming mail for *example.com* to *internal.example.com*, add the following lines to Exim's configuration file:

```
domainlist local_domains = @
domainlist relay_to_domains = example.com
```

The key feature of this configuration is that *example.com* is a domain to which Exim may relay but is not on the list of local domains (for which mail is to be delivered on this host). Remember that you must restart Exim after changing its configuration file.

Routing changes

Mail from the Internet for *example.com* should be sent to the spam-checking gateway *mail.example.com*. Add a DNS MX record for the *example.com* domain that points to *mail.example.com*.

Once received by *mail.example.com*, messages will be spam-checked and should then be relayed to *internal.example.com* by Exim. There are two ways to get Exim to relay these messages:

- Set up an internal DNS MX record for *example.com* pointing to *internal.example.com*. When Exim on *mail.example.com* attempts to deliver messages for *example.com*, the `dnslookup` router will look up this MX record and deliver the messages to the internal mail host. This configuration may require that you run a "split DNS" system or use BIND 9's views feature to ensure that different MX records for *example.com* are published to the Internet and to internal hosts.
- Set up a new Exim router using the `manualroute` driver to manually route incoming messages for *example.com* to the internal mail host. The router definition, shown in Example 8-12, should be placed in the list of routers before the `dnslookup` router. In this case, *mail.example.com* need only be able to resolve *internal.example.com* (or the IP address for *internal.example.com* could be substituted for its name in the router definition).

Example 8-12. Using a manualroute router to relay messages

```
internal_relay:
  driver = manualroute
  domains = example.com
  transport = remote_smtp
  route_list = example.com internal.example.com
```

Internal server configuration

Once the external mail gateway is in place, you can configure the internal mail server to accept only SMTP connections from the gateway (for incoming Internet mail). If you don't have a separate server for outgoing mail, the internal mail server should also accept SMTP connections from hosts on the internal network. These restrictions are usually enforced by limiting access to TCP port 25 using a host-based firewall or a packet-filtering router.

Adding Sitewide Bayesian Filtering

You can easily add sitewide Bayesian filtering to any of the Exim approaches because they are all based on `spamd`. Use the usual SpamAssassin `use_bayes` and `bayes_path` directives in *local.cf*, and ensure that `spamd` has permission to create the databases in the directory named in `bayes_path`. Use a directory for the databases that is owned by `spamd`'s user, such as */var/spamd* (or perhaps use */etc/mail/spamassassin*). If local users need access to the databases (e.g., they will be running `sa-learn`), you may have to make the databases readable or writable by a group other than `spamd`'s and adjust `bayes_file_mode`. Or you can make the databases world-readable or world-writable. Doing so, however, is unlikely to be necessary on a gateway system and puts the integrity of your spam-checking at the mercy of the good intentions and comprehension of your users.

Adding Sitewide Autowhitelisting

Adding sitewide autowhitelisting is very similar to adding a sitewide Bayesian database. Just add the usual SpamAssassin `auto_whitelist_path` and `auto_whitelist_file_mode` directives to *local.cf*. As with the Bayesian databases, `spamd`'s user must have permission to create the autowhitelist database and read and write to it. In SpamAssassin 2.6x, `spamd` must be started with the `--auto-whitelist` option; this option is not needed (and is deprecated) in SpamAssassin 3.0.

CHAPTER 9

Using SpamAssassin as a Proxy

In some environments, it makes no sense to install SpamAssassin on the mail server. For example, the mail server may be underpowered to perform content-checking. Or perhaps users have widely ranging preferences for how much (or indeed whether) spam-checking should be performed, and they may not have accounts on the mail server or any convenient way of configuring their preferences. In these environments, one way to provide those users who want the power of SpamAssassin with spam-checking is to help them install a SpamAssassin POP proxy.

Many more POP proxies are available than IMAP proxies, primarily because IMAP is a much more complex protocol and doesn't require that messages be downloaded to the client. At the time of writing, no freely distributed SpamAssassin IMAP proxies for Windows clients were available.

In addition, most extant proxies call SpamAssassin through the Perl API to avoid having to run the `spamassassin` shell script or a persistent `spamd` daemon. Because the Perl API will change in SpamAssassin 3.0, proxies written for SpamAssassin 2.63 are unlikely to continue to work until they are upgraded.

Proxy software is middleware. A proxy receives connections from a client and relays them to a server, intercepting all communication in each direction. Application proxies have been used to pierce smart holes in strong firewalls, to cache frequently accessed data, and to perform a variety of other functions.

Logically, POP proxies sit between a mail client and a POP server. Actually, these proxies typically run on the same computer as the mail client. The proxies discussed in this chapter not only relay data (email messages) between the client and server, but also invoke SpamAssassin to perform spam-checking on the email after it has been received from the server but before it is relayed to the client. Users continue to use their favorite POP client; no changes need be made at the POP server.

In this chapter, I review two SpamAssassin proxies. The first is the venerable Pop3proxy, a freely distributed command-line proxy script written in Perl and suitable for use on several operating systems. The second is the commercial proxy SAproxy Pro from Stata Labs.

POP proxies do not offer the complete functionality of POP servers; in particular, they may be limited in how they can perform authentication and secure the transaction. Using a POP proxy may result in sending your email password across the Internet in the clear.

Figure 9-1 illustrates the example topology for this chapter. *pop.example.com* is a POP mail server. *win.example.com* is a Windows-based user workstation that runs a POP mail client (e.g., Outlook Express, Eudora, Netscape Messenger). The SpamAssassin POP proxy will be installed on *win.example.com*, and the mail client will be configured to connect to the proxy rather than to the POP server. The proxy will be configured to connect to the POP server and to run SpamAssassin on messages as they are downloaded.

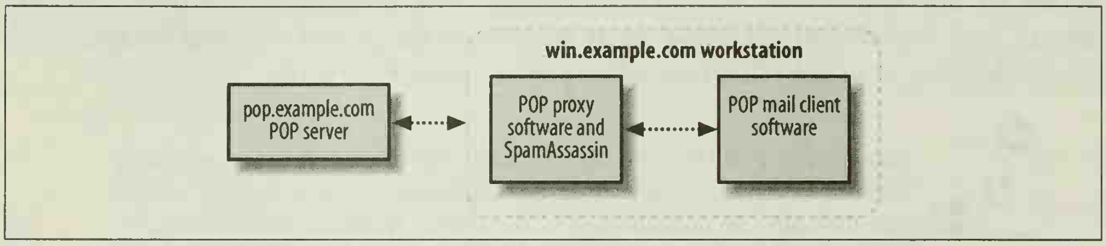

win.example.com workstation
pop.example.com POP server
POP proxy software and SpamAssassin
POP mail client software

Figure 9-1. An example POP mail topology with a client-side proxy

Using Pop3proxy

Pop3proxy, by Dan McDonald, is one of the oldest SpamAssassin POP proxies and, to its credit, still functions well with SpamAssassin 2.63. It's a no-fills proxy written in Perl and requires manual installation and configuration. It does not perform network-based SpamAssassin tests. Download Pop3proxy (and read the manual) at *http://mcd.perlmonk.org*.

Installing Pop3proxy

Follow these steps to install Pop3proxy:

1. Download *pop3proxy.zip* and unpack it into a directory of your choice. For this example, I assume you've unpacked it in *C:\pop3proxy* so a directory listing of that directory would look like this:

```
C:\pop3proxy>dir /s

 Directory of C:\pop3proxy
```

```
03/26/2004  09:56p      <DIR>          .
03/26/2004  09:56p      <DIR>          ..
03/26/2004  09:56p      <DIR>          pop3proxy
08/18/2002  05:40p              60,781 pop3proxy.pl
08/18/2002  05:40p              28,798 pop3proxy.html
               2 File(s)         89,579 bytes

 Directory of C:\pop3proxy\pop3proxy

03/26/2004  09:56p      <DIR>          .
03/26/2004  09:56p      <DIR>          ..
08/12/2002  08:19a               6,240 Artistic
08/11/2002  08:45p                 567 kill_proxy.pl
08/12/2002  08:30a                 536 hostmap.sam
               3 File(s)          7,343 bytes
```

2. Install a version of Perl for Windows that includes the *Time::HiRes* module. Several Perl distributions for Windows are available, but one that is known to work (and provides a precompiled version of the module) is ActivePerl, available at *http://www.activestate.com/Products/ActivePerl*. Either ActivePerl 5.6.1 or 5.8.3 works well with Pop3proxy. ActivePerl 5.8.3 supports Unicode. ActivePerl 5.6.1 does not support Unicode but has been extensively tested with SpamAssassin. In this example, I assume you've installed ActivePerl in *C:\perl*.

   Time::Hires can be installed through ActivePerl's Perl Package Manager. After installing ActivePerl, run the Perl Package Manager, and type `install Time::HiRes` at the `ppm>` prompt. Type `quit` to exit the Package Manager.

3. Download and unpack SpamAssassin. Copy all of the files and directories in SpamAssassin's *lib* directory to ActivePerl's *C:\perl\site\lib* directory. Copy SpamAssassin's *rules* directory and all its contents to *C:\pop3proxy\rules*. Copy the *user_prefs.template* file from the *rules* directory to *C:\pop3proxy* and rename it *user_prefs*. The *C:\pop3proxy* directory should now look like this:

```
C:\pop3proxy>dir /w /s

 Directory of C:\pop3proxy

[.]              [..]              [pop3proxy]      pop3proxy.html
pop3proxy.log    pop3proxy.pl      [rules]          user_prefs
               4 File(s)        110,825 bytes

 Directory of c:\pop3proxy\pop3proxy

[.]             [..]            Artistic        hostmap.sam     kill_proxy.pl
               3 File(s)          7,343 bytes

 Directory of C:\pop3proxy\rules

[.]                     [..]                    10_misc.cf
20_anti_ratware.cf      20_body_tests.cf        20_compensate.cf
20_dnsbl_tests.cf       20_fake_helo_tests.cf   20_head_tests.cf
20_html_tests.cf        20_meta_tests.cf        20_phrases.cf
```

```
20_porn.cf              20_ratware.cf           20_uri_tests.cf
23_bayes.cf             25_body_tests_es.cf     25_body_tests_pl.cf
25_head_tests_es.cf     25_head_tests_pl.cf     30_text_de.cf
30_text_es.cf           30_text_fr.cf           30_text_it.cf
30_text_pl.cf           30_text_sk.cf           50_scores.cf
60_whitelist.cf         languages               local.cf
regression_tests.cf     STATISTICS-set1.txt     STATISTICS-set2.txt
STATISTICS-set3.txt     STATISTICS.txt          triplets.txt
user_prefs.template
```

4. Edit *C:\pop3proxy\user_prefs* and set up SpamAssassin preferences. See Chapters 2 and 3 for configuration details.

Starting Pop3proxy

To start Pop3proxy, you must invoke Perl on the `pop3proxy.pl` script and provide command-line arguments to identify the POP server. If you allowed ActivePerl to associate its `perl.exe` program with *.pl* file extensions, you should be able to execute `pop3proxy.pl` directly. Otherwise, set up a shortcut or batch file containing:

```
c:\perl\bin\perl c:\pop3proxy\pop3proxy.pl --host pop.example.com
```

When invoked, the shortcut will open a DOS window and execute the proxy script. You can stop the proxy by typing CTRL-C in the DOS window. When you've confirmed that it's working as you like, you can replace *\perl\bin\perl* with *\perl\bin\wperl* in the shortcut. `wperl` runs the script in the background (without opening a DOS window); use it when you plan to keep Pop3proxy running all the time. You can stop the proxy by invoking the `kill_proxy.pl` script included with Pop3proxy, or by using the Windows Task Manager to kill the `wperl` process.

Here is a complete list of Pop3proxy's command-line arguments:

`--host` *`hostname[:port]`*
: Provide the hostname (and optionally, the port number) of the remote POP server to proxy.

`--logfile` *`filename`*
: Provide the name of a file to log connection and status information to. This defaults to *pop3proxy.log*. The log file can be useful in debugging problems with Pop3proxy. Example 9-1 shows a Pop3proxy log of a successful connection in which Pop3proxy downloaded two messages and classified one as spam.

`--maxscan` *`bytes`*
: Specify the largest message, in bytes, that Pop3proxy will invoke SpamAssassin on. The default is 250,000, which is reasonable. Larger sizes cause more messages to be scanned, but larger messages scan more slowly.

`--nopad`
: POP servers sometimes provide POP clients with message- and mailbox-size information. Running SpamAssassin on a message when it's between server and

client can change (typically, enlarge) the message size. Most modern clients handle this situation with no problems, but if yours does not, the `--nopad` option causes Pop3proxy to overwrite text in existing headers rather than adding new ones, maintaining a constant size at the cost of obfuscating the message headers to a small degree.

`--allowtop`
: The POP protocol provides a `TOP` command that the client uses to request only a limited amount of a message from the server (deferring the retrieval of the rest of the message until it's explicitly asked for). `TOP` doesn't interact well with spam-checking proxies, and Pop3proxy normally prevents the client from using it. If you want to try it out anyway, use the `--allowtop` argument.

`--exitport` *`portnumber`*
: The `kill_proxy.pl` script works by connecting to a second port that `pop3proxy.pl` listens on. Any connections on this port cause `pop3proxy.pl` to exit. By default, the port number is 9625, but you can use the `--exitport` option to change it if you use that port number for something else. If you change the port number, you must edit the `kill_proxy.pl` script and change the value of `$exitport` near the beginning of the file.

Example 9-1. A log from Pop3proxy

```
New connection:
From: 127.0.0.1:2094
To:   192.168.0.4:110
192.168.0.4:110 (Server) said +OK to none
+OK POP3 Ready pop.example.com
127.0.0.1:2094 (Client) said CAPA
192.168.0.4:110 (Server) said -ERR to CAPA
127.0.0.1:2094 (Client) said USER
192.168.0.4:110 (Server) said +OK to USER
127.0.0.1:2094 (Client) said PASS
192.168.0.4:110 (Server) said +OK to PASS
127.0.0.1:2094 (Client) said STAT
192.168.0.4:110 (Server) said +OK to STAT
127.0.0.1:2094 (Client) said UIDL
192.168.0.4:110 (Server) said +OK to UIDL
127.0.0.1:2094 (Client) said LIST
192.168.0.4:110 (Server) said +OK to LIST
127.0.0.1:2094 (Client) said RETR
192.168.0.4:110 (Server) said +OK to RETR
Snarfing RETR response
Detected end of snarfed multiline
35510 bytes, SPAM, Message-id: <200402062012.i16KCGBb015885@example.com>

127.0.0.1:2094 (Client) said LIST
192.168.0.4:110 (Server) said +OK to LIST
127.0.0.1:2094 (Client) said RETR
192.168.0.4:110 (Server) said +OK to RETR
Snarfing RETR response
```

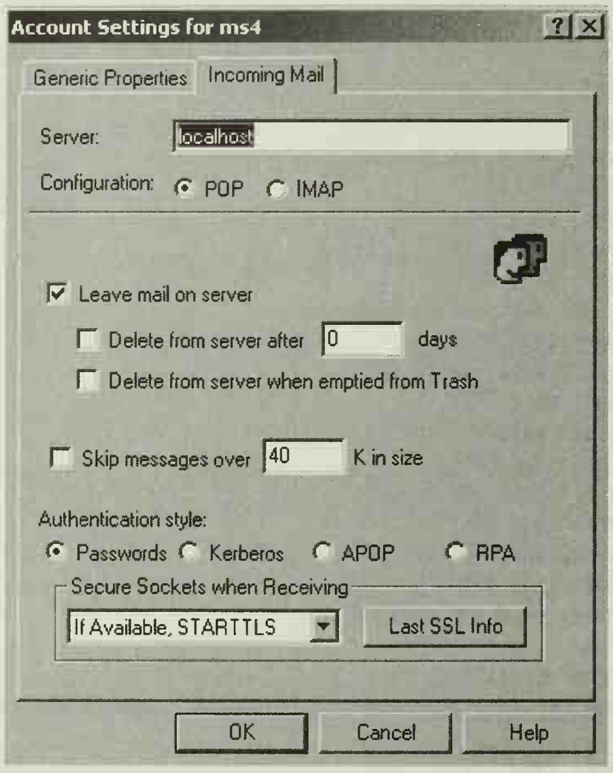
Example 9-1. A log from Pop3proxy (continued)

```
Detected end of snarfed multiline
2096 bytes, NOT spam, Message-id: <1063926361.20040206152006@oreilly.com>

127.0.0.1:2094 (Client) said QUIT
192.168.0.4:110 (Server) said +OK to QUIT

192.168.0.4:110 - socket close on read
Flushing peer on close
127.0.0.1:2094 - peer gone after write, closing
```

Pop3proxy can proxy multiple POP servers through the use of a *hostmap* file. See the Pop3proxy manual for more information about setting up such a file.

Configuring the POP Client

Finally, you must reconfigure a mail client to connect to *localhost* (or 127.0.0.1) instead of the usual POP server. Connections to *localhost* will be received by Pop3proxy and proxied to the POP server. Figure 9-2 shows the Eudora 5.1 dialog box for configuring the incoming POP server for an account.


Account Settings for ms4

Generic Properties | Incoming Mail

Server: localhost

Configuration: [✓ POP] IMAP

[✓] Leave mail on server

[ ] Delete from server after 0 days

[ ] Delete from server when emptied from Trash

[ ] Skip messages over 40 K in size

Authentication style: [✓ Passwords] Kerberos APOP RPA

Secure Sockets when Receiving: If Available, STARTTLS

Figure 9-2. Configuring Eudora to use Pop3proxy

Using SAproxy Pro

SAproxy Pro, by Stata Labs, began as a commercialized version of Pop3proxy, but it has been extensively developed with a focus on ease-of-use and access to both SpamAssassin and POP client features. It's available for Windows operating systems. At the time of this writing, the latest version is 2.5 and is selling for $29.95, which includes a year of free upgrades. A free 15-day trial is available. You can download the trial version or purchase the product at *http://www.statalabs.com/products/saproxy/*.

SAproxy Pro 2.5 includes its own Perl library and SpamAssassin 2.63, so you don't have to install either of those products separately. When SpamAssassin 3.0 is released, a future version of SAproxy Pro is likely to distribute SpamAssassin 3.0 instead, and upgrading should be relatively simple.

Installing SAproxy Pro

SAproxy Pro uses a downloadable InstallShield installer. Once you've downloaded the installer, run it. You'll be prompted to select your mail client; for several of the most widely used mail clients, the installer includes a training video that demonstrates how to configure mail accounts to use the proxy. You may be required to reboot your computer to finish the installation.

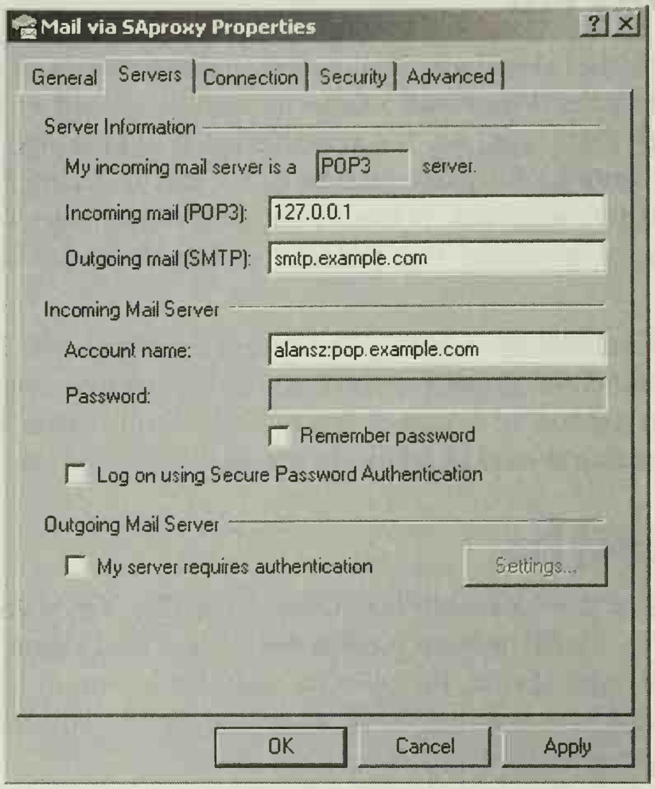
Starting SAproxy Pro

After SAproxy Pro is installed, you can start it manually from the Windows Start menu. The installer also offers to configure SAproxy Pro to start automatically on system startup. When SAproxy Pro is running, a system tray icon will appear; right-clicking this icon brings up a menu for configuring or shutting down SAproxy Pro.

Configuring the POP Client

Configuring a POP client to use SAproxy Pro is straightforward. Set the incoming POP server to *localhost* or 127.0.0.1. Set the POP login name to your usual POP account name, followed by a colon and the hostname of the remote POP server. Figure 9-3 shows an example of this configuration in Microsoft Outlook Express 6. In the example, Outlook Express will connect to 127.0.0.1 and log in as *alansz:pop.example.com*. SAproxy Pro will accept the connection and will proxy for the POP server *pop.example.com*, using *alansz* as the login name.

Stata Labs also distributes an SSL plug-in module for SAproxy Pro at *http://www.statalabs.com/products/saproxy/ssl/*. If your POP server supports SSL connections, install the plug-in and add `:ssl` to the end of the POP login name (e.g.,



Server Information

My incoming mail server is a POP3 server.

Incoming mail (POP3): 127.0.0.1

Outgoing mail (SMTP): smtp.example.com

Incoming Mail Server

Account name: alansz:pop.example.com

Password:

☐ Remember password

☐ Log on using Secure Password Authentication

Outgoing Mail Server

☐ My server requires authentication

Figure 9-3. Configuring Outlook Express 6 to use SAproxy Pro

alansz:pop.example.com:ssl) to direct SAproxy Pro to make an SSL connection to the POP server.

Configuring SAproxy Pro

SAproxy Pro really shines in its configuration interface, which is available by double-clicking the SAproxy Pro system tray icon, or by right-clicking the icon and selecting Configure SAproxy Pro. Most of the configuration options are the same as those available in Pop3proxy, and through SpamAssassin's preference files. The graphical interface makes them much easier for inexperienced users to select. The Configuration dialog box is divided into nine tabs:

Always Spam
: Use this section to add blacklist entries. You can blacklist messages by sender email address, sender domain, or keyword.

Never Spam
: Use this section to add whitelist entries. You can whitelist messages by sender email address or sender domain.

Spam Training
: Use this section to enable use of SpamAssassin's Bayesian classifier (including auto-learning). SAproxy Pro can also manually scan email folders that you specify as containing spam or non-spam messages in order to train the classifier.

Language Settings
: Use this section to limit the set of languages that you expect to receive email in; email in other languages will be treated as spam.

Safety Settings
: Use this section to configure the `report_safe` SpamAssassin directive.

Tagging Options
: Use this section to turn on subject-tagging for spam messages and to set the SpamAssassin threshold score for spam. SAproxy Pro allows thresholds between 3.5 and 6.5.

Advanced Settings
: Use this section to turn on such options as logging, proxying of the `TOP` and `AUTH` commands, and the use of SpamAssassin's network tests. DCC, Pyzor, Vipul's Razor, and DNSBL tests are available, and you can turn each on or off independently. You can use the Host Map part of this section to configure SAproxy Pro to listen on different local ports to proxy connections to different remote POP servers.

Statistics
: This section displays a line graph comparing the number of spam and non-spam messages received each day of the last month.

Help
: This section provides links to SAproxy Pro help.

APPENDIX
Resources

This appendix lists useful online resources for further information about spam, spam-filtering, SpamAssassin, each of the mail transport agents discussed in this book, and several other SpamAssassin-related software packages.

General Spam Resources

"Help! I've been spammed! What do I do?"—originally written by Chris Lewis and maintained by Greg Byshenk—is an helpful (if dated) guide to spam and spam-prevention for the beginner. Find it at *http://www.byshenk.net/ive.been.spammed.html*.

Internet Request For Comments (RFC) documents describe proposed standards for the Internet. You can get RFCs from *http://www.rfc-editor.org*. Some notable RFCs related to spam and spam filtering include

RFC 2822: Internet Message Format
: The basic document that describes the formatting of email messages.

RFC 2821: Simple Mail Transfer Protocol
: Explains SMTP, the protocol used to transfer email from system to system.

RFC 2505: Anti-spam Recommendations for Internet MTAs
: Describes a set of best practices for mail servers.

The SPAM-L FAQ, maintained by Doug Muth at *http://www.claws-and-paws.com/spam-l/*, provides information about the *SPAM-L* mailing list, one of the oldest discussion forums for spam fighters.

http://spam.abuse.net is a long-standing site with information for antispam advocates and system adminstrators.

The Coalition Against Unsolicited Commercial Email (CAUCE) has a web site at *http://www.cauce.org*. CAUCE focuses primarily on advocacy and legislation.

The groups in the *news.admin.net-abuse* Usenet hierarchy are devoted to discussing and reporting Net abuse, including spam (see particularly *news.admin.net-abuse.email*).

Spam-Filtering

http://www.openrbl.org provides a long list of DNSBLs that may be suitable for spam-filtering. It includes hit rates (but not false positive rates) against its own recently collected spam corpus for each DNSBL.

Vipul's Razor, Pyzor, and DCC are collaborative spam-filtering clearinghouses that can be consulted by SpamAssassin. Vipul's Razor is available at *http://razor.sourceforge.net*. Pyzor is at *http://pyzor.sourceforge.net*. DCC is at *http://www.rhyolite.com/anti-spam/dcc/*.

DSPAM is a spam-filtering system using statistical-algorithmic hybrid filtering—filters that are trained like SpamAssassin's Bayesian classifier. Find it at *http://www.nuclearelephant.com/projects/dspam/*.

CRM114, the Controllable Regex Multilator, is a spam-filtering system based on learning regular expressions. Download it at *http://crm114.sourceforge.net/*.

SpamAssassin

The home page for SpamAssassin itself is, of course, *http://www.spamassassin.org*. The site contains links to the *spamassassin-users* and *spamassassin-dev* mailing lists.

SAProxy Pro, a commercial client-side SpamAssassin proxy, is available from *http://www.statalabs.com*.

Mail Transport Agents

In this book, I describe how to use SpamAssassin in conjunction with several MTAs. The following sections point you to more information about each of those agents.

Sendmail

Sendmail has two primary web sites. The open source version maintained by the Sendmail consortium can be found at *http://www.sendmail.org*. Sendmail's commercial face is *http://www.sendmail.com*.

Sendmail's own antispam provisions are documented at *http://www.sendmail.org/m4/anti_spam.html*. If you're not a Sendmail guru, pick up Bryan Costales' book *Sendmail* (O'Reilly), the Sendmail bible. If you are a Sendmail guru, you probably already have a copy! Another good source is Frederick Avolio and Paul Vixie's *Sendmail: Theory and Practice* (Digital Press).

Sendmail's filtering architecture, milter, has led to the development of many filtering tools. The web site *http://www.milter.org* is the most comprehensive catalog of such filters.

Postfix

The home page for the Postfix MTA is *http://www.postfix.org*.

Two useful manuals for Postfix are *Postfix: The Definitive Guide* by Kyle Dent (O'Reilly) and *The Book of Postfix* by Ralf Hildebrandt and Patrick Koetter (No Starch Press).

qmail

The home page for the qmail MTA is *http://www.qmail.org*. The netqmail distribution of qmail includes the QMAILQUEUE patch, which is used by most SpamAssassin-integration solutions.

The online book *Life with qmail* (*http://www.lifewithqmail.org*) provides excellent documentation for qmail.

Exim

The home page for the Exim MTA is *http://www.exim.org*. The site includes a link to the Exim specification, which serves as the manual for the MTA. Those preferring a real book should purchase Philip Hazel's *The Exim SMTP Mail Server: Official Guide for Release 4* (UIT Cambridge).

Related Mail Tools

The following sections point to information about various other mail tools mentioned in this book.

procmail

procmail is a popular and powerful Unix filtering program that acts as a full-featured local email delivery agent. It's available at *http://www.procmail.org*.

MIMEDefang

MIMEDefang is a Sendmail milter application written in Perl that provides a framework for mail content scanning, including virus-scanning, spam-checking with

SpamAssassin, and MIME validation. It's available at *http://www.mimedefang.org*. Versions after 2.42 support SpamAssassin 3.0.

amavisd-new

amavisd-new is a high-performance daemonized content scanner, designed for use with Postfix, but it also supports Sendmail (including a milter version) and Exim. It's available at *http://www.ijs.si/software/amavisd/*. Don't confuse amavisd-new with amavis, amavis-perl, amavisd, or amavis-ng, all of which are other content scanners, most not actively in development, that now share little code in common with amavisd-new.

sa-exim

sa-exim is a *local_scan.c* replacement for Exim that provides SpamAssassin message-scanning during a SMTP transaction. It offers many options for how to handle spam that it detects, including teergrubing. sa-exim's home page is *http://marc.merlins.org/linux/exim/sa.html*.

exiscan-acl

exiscan-acl is a patch for Exim 4 that adds new content-scanning ACL directives to Exim. These directives can be used to invoke SpamAssassin on messages during a SMTP transaction. Some prepackaged Exim distributions already have this patch added. You can download it at *http://duncanthrax.net/exiscan-acl/*.

qmail-scanner

qmail-scanner is a content scanner for qmail that can run SpamAssassin on messages early in the delivery process. It contains its own implementation of `spamc` for faster checking. Find it at *http://qmail-scanner.sourceforge.net*.

Index

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

About the Author

Alan Schwartz is an associate professor of clinical decision making in the departments of Medical Education and Pediatrics at the University of Illinois at Chicago. He is the author of *Managing Mailing Lists*, and coauthor of *Stopping Spam* and *Practical Unix and Internet Security*, Third Edition (O'Reilly). In his spare time, he develops and maintains the PennMUSH MUD server. He and his wife also develop and maintain their son. As mail administrator for a number of organizations, he deals with unsolicited email on a daily basis. Turn-ons for Alan include sailing, programming in Perl, playing duplicate bridge, and drinking Anchor Porter. Turn-offs include spam (obviously!) and watery American lagers.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *SpamAssassin* is a King vulture. King vultures (*Sarcoramphus papa*) are found in tropical lowland forests from Mexico to Argentina. Vultures serve a useful purpose in disposing of dead and decaying animal remains.

Their extremely thick and strong bills are well adapted for tearing, as are their long, thick claws for holding meat. King vultures have a very heavy beak, a necessary asset for tearing into large, thick-skinned animals. The absence of feathers on a vulture's head helps the bird "clean up" after a messy meal.

King vultures do not build nests. The female simply deposits her single egg in the hollow of a rotten tree trunk or in a crack caused by age or lightning. Both parents take turns incubating the egg. When hatched, King vulture chicks have no feathers, but soon are covered with a white down. They do not acquire their adult plumage until they are 18 months old.

While they have very keen eyesight, King vultures have a poor sense of smell. They often rely on other vulture species to locate food. Much larger and stronger than other vultures, the King vulture is useful to the others because it is capable of tearing open tough skin. When other vultures have found carrion, a King vulture or two often arrive and immediately dominate the carcass—hence the name "King."

King vultures will sometimes glide in wide circles for long periods, spying on their domain below and searching for food. By making perfect use of the air currents, vultures are able to soar for hours at a time, without once flapping their wings.

Darren Kelly was the production editor, Nancy Crumpton was the copyeditor, and Jan Fehler was the proofreader for *SpamAssassin*. Reg Aubry and Claire Cloutier provided quality control. Nancy Crumpton provided production services and wrote the index.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

Melanie Wang designed the interior layout, based on a series design by David Futato. This book was converted by Andrew Savikas to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Darren Kelly.

Network Administration/Email

O'REILLY®

SpamAssassin

Want to work from home and get paid well? Everyday can be payday! Perhaps you'd prefer to PROTECT YOUR PET FROM FLEAS AND TICKS? Tired Of Online Dating? Meet Someone Real! Tired with life? Buy Xanax!

Spam. The scourge of the Internet. It fills our inboxes, burns our bandwidth, brings raw and licentious images into the sanctity of our homes, and is single-handedly responsible for almost totally destroying the utility of electronic mail. How do you fight this rising electronic tide of email sewage? For many, the answer lies in a free and open source tool known as SpamAssassin.

SpamAssassin is perhaps the most widely deployed anti-spam tool on the Internet today. Let Alan Schwartz, experienced mail administrator, show you how to apply this powerful tool.
You'll learn how to:

- Customize SpamAssassin's rules, and even create new ones
- Train SpamAssassin's Bayesian classifier, a statistical engine for detecting spam, to optimize it for the sort of email that you typically receive
- Block specific addresses, hosts, and domains using third-party blacklists like the one maintained by Spamcop.net
- Whitelist known good sources of email, so that messages from clients, coworkers, and friends aren't inadvertently lost
- Configure SpamAssassin to work with newer spam-filtering methods such as Hashcash (*www.hashcash.org*) and Sender Policy Framework (SPF)

Alan also shows how to install SpamAssassin in a variety of different configurations. SpamAssassin integrates with all major mail transport and delivery agents, including the venerable sendmail, procmail, postfix, qmail, and Exim.

International cell calls for a nickel? Forget that. SpamAssassin is free. And even better, SpamAssassin has proven to be effective. Buy this book. Install SpamAssassin. Take back your inbox.

"Detailed, accurate, and informative—recommended for spam-filtering beginners and experts alike."

—Justin Mason, SpamAssassin development team

Visit O'Reilly on the Web at *www.oreilly.com*

ISBN 0-596-00707-8
US $24.95
CAN $36.95
90000
9 780596 007072
6 36920 00707 4