The Open Source Solution to SPAM

Covers SpamAssassin version 3.0

SpamAssassin

O'REILLY®

Alan Schwartz

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 *gzip*ped *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.

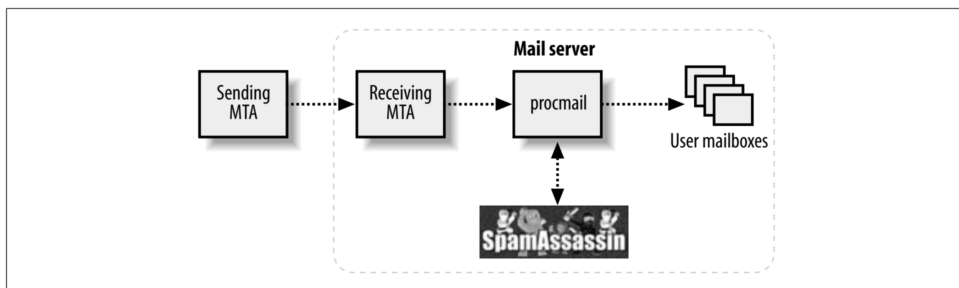


Figure 2-1. Invoking SpamAssassin with procmail

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.

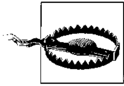
--ident-timeout *number-of-seconds*
: Specify the number of seconds to wait for the ident server to respond. If the response doesn't come after this number of seconds, spamd will refuse to perform spam-checking for the connection.

--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.

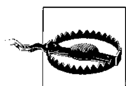
Handling problems

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%.