Combatting Spam Using Sendmail, MIMEDefang and Perl

Copyright 2004 David F. Skoll

Roaring Penguin Software Inc.
(Booth #21)

Administrivia

- Please turn off or silence cell phones, pagers, Blackberry devices, etc...
- After the tutorial, please be sure to fill out an evaluation form and return it to the USENIX folks.

Overview

- After this tutorial, you will:
  - Understand how central mail filtering works.
  - Know how to use MIMEDefang to filter mail.
  - Be able to integrate SpamAssassin into your mail filter.
  - Know how to implement mail filtering policies with MIMEDefang and Perl.
  - Know how to fight common spammer tactics.

Outline

- Introduction to Mail Filtering
- Sendmail's Milter API
- MIMEDefang Introduction, Architecture
- Writing MIMEDefang Filters
- SpamAssassin Integration
- Advanced Filter Writing
- Fighting Common Spammer Tactics
- Advanced Topics
- Policy Suggestions

Assumptions

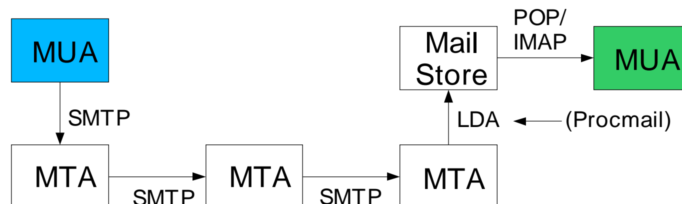
- I assume that you:
  - Are familiar with Sendmail configuration. You don't need to be a *sendmail.cf* guru, but should know the basics.
  - Are familiar with Perl. Again, you don't need to be able to write an AI program in a Perl one-liner, but should be able to read simple Perl scripts.
  - Are running the latest version of Sendmail 8.12 or 8.13 on a modern UNIX or UNIX-like system.

Why Filter Mail?

- The old reason: to stop viruses.
- The new reason: to stop spam and inappropriate content.
- Blocking viruses is easy. Block .exe and similar files, and test against signature databases.
- Blocking spam is hard, but becoming increasingly important. Organizations can even face lawsuits over inappropriate content.

Mail filtering is required for many reasons. In addition to the reasons given on the slide, you might need to filter outgoing mail as well to prevent virus propagation, dissemination of sensitive information, etc.

One-slide Overview of Mail Delivery



MUA = Mail User Agent. What you use to send and read mail. Eg: Mozilla, Evolution
MTA = Mail Transfer Agent. Eg: Sendmail, Postfix, Exim, qmail
Mail Store = Where mail is stored until it is read. Usually in /var/spool/mail
LDA = Local Delivery Agent. Program which performs final delivery into the mail store
POP = Post Office Protocol. A method for retrieving mail from a mail store
IMAP = Internet Message Access Protocol. A better protocol than POP/POP3.
SMTP = Simple Mail Transfer Protocol. The language used by MTA's to talk to each other.

In general, e-mail may traverse a series of MTA's before arriving at the final MTA. The path an e-mail takes can depend on network conditions, whether or not any servers have crashed, etc.

Every arrow on the slide represents an opportunity for filtering mail.

Filtering on the “LDA” arrow is the procmail approach: Filter during final delivery to the mail store.

Filtering on the “POP/IMAP” arrow is the most common approach. This is the approach taken by many MUA's, as well as programs like POPFile.

Filtering on the SMTP arrow is the MIMEDefang approach. It does real-time filtering during the SMTP conversation.

Filtering with Procmail

- The local delivery agent does filtering.
- PRO: Easy to customize rules per-recipient. Well-known and tested.
- CON: Inefficient. Messages sent to several users are processed several times.
- CON: Local users only. Difficult to use procmail on a store-and-forward relay.
- CON: SMTP transaction is complete by the time Procmail is invoked.

8

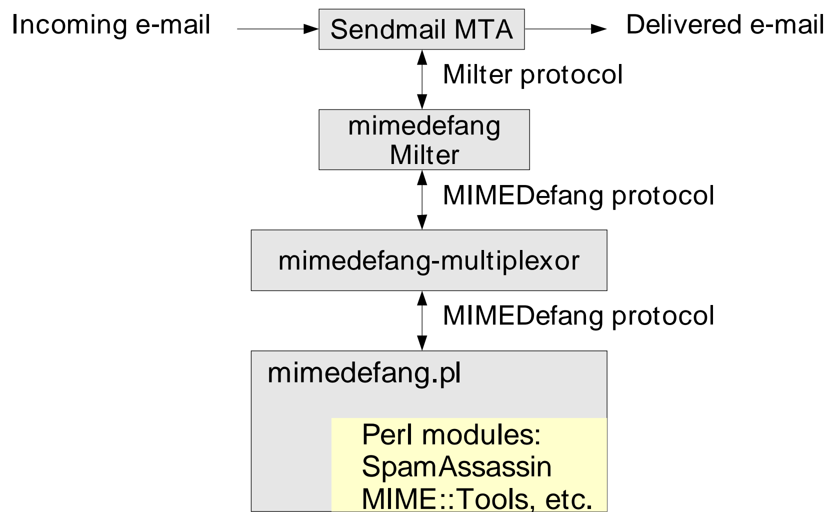
Procmail is an old and well-established approach to doing mail filtering. It filters mail during delivery into the local mail store.

Procmail's rule language is powerful, but not as well known as Perl. If you invoke external programs from a procmail filter, you pay a large performance penalty – an external program must be invoked for each delivery.

Procmail cannot influence the SMTP conversation. By the time it is invoked, the STMP conversation is almost over – we are at the end of the DATA phase.

Central Filtering (Topic of This Talk)

- The MTA does the filtering.
- PRO: Efficient. Each message filtered once, regardless of number of recipients.
- PRO: Can modify the SMTP dialog. Amazingly useful, as you'll see...
- PRO: Can filter relayed mail, not just local.
- CON: Harder (but not impossible) to implement per-recipient filter rules.



E-mail is accepted,queued and routed as usual by Sendmail.

Sendmail communicates with the **`mimedefang`** program using the Milter protocol. **`mimedefang`** is linked against libmilter.

`mimedefang` passes requests to **`mimedefang-multiplexor`** using its own MIMEDefang protocol.

`mimedefang-multiplexor` picks a free **`mimedefang.pl`** program to actually do the filtering.

SpamAssassin is integrated into **`mimedefang.pl`** at the Perl module level. Various other Perl modules, such as Anomy::HTMLCleaner, may be integrated into **`mimedefang.pl`**.

The MIMEDefang Philosophy

- MIMEDefang provides *mechanism*; you provide *policy*.
- MIMEDefang gives you lots of little tools to manipulate mail. It's up to you to hook them together in just the way you want.
- MIMEDefang design and implementation emphasize security, flexibility and performance, in that order.

Why a Filtering API?

- Sendmail's configuration file rule sets are probably Turing-complete.
- But they are awkward (to say the least!) for real-world filtering tasks.
- Hence, Sendmail authors created an API for external filters to process mail.
- Allows the use of more appropriate filtering languages (although reference implementation is in C only.)

Why Perl?

- Perl is much better than C for text manipulation.
- CPAN has many, many e-mail manipulation modules (and lots of other useful code) available for free.
- Many more system administrators know Perl than C.
- There are milters in other languages (Python, Tcl), but they lack access to the incredible resources of CPAN.

Why Perl? (2)

- Hastily-designed configuration languages inevitably accumulate warts and bugs.
- Perl was an ideal extension language for MIMEDefang, as well as an implementation language.
- Perl gives the ultimate in flexibility, and we don't need to worry about parsing configuration files.

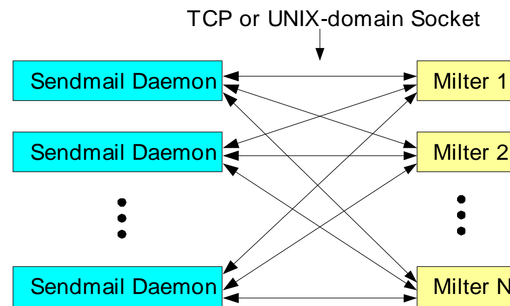
Sendmail's Milter API

- Milter (Mail fILTER) is both a *protocol* and a *library*.
- Designed by the Sendmail development team, and has been part of Sendmail since 8.10. Matured in 8.12.
- Lets filters "listen in" to the SMTP conversation and modify SMTP responses.
- Extremely powerful and can let you do amazing things with e-mail.

15

Milter is intimately connected with the SMTP conversation. It can modify responses to SMTP commands, as well as alter mail contents.

Milter Architecture



Sendmail is single-threaded, but forks into multiple processes. The milter library is multi-threaded, and a given Sendmail installation can have multiple mail filters (filters 1 through N above.)

Milter API Operation

- At various stages of the SMTP conversation, Sendmail sends a message over the socket to the milter.
- The milter library invokes a *callback* into your code. It then sends a reply message back to Sendmail containing the return value from your callback.
- In addition, you can call milter library functions that send special messages to Sendmail to modify aspects of the message.

Typical SMTP Conversation

```
C: Connect to server
S: 220 server_hostname ESMTP Sendmail 8.12.7/8.12.7...
C: HELO client_hostname
S: 250 server_hostname Hello client_hostname, pleased...
C: MAIL FROM:<dfs@roaringpenguin.com>
S: 250 2.1.0 <dfs@roaringpenguin.com>... Sender ok
C: RCPT TO:<foo@roaringpenguin.com>
S: 250 2.1.5 <foo@roaringpenguin.com>... Recipient ok
C: RCPT TO:<bar@roaringpenguin.com>
S: 250 2.1.5 <bar@roaringpenguin.com>... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: (transmits message followed by dot)
S: 250 2.0.0 h0AJVcGM007686 Message accepted for delivery
C: QUIT
S: 221 2.0.0 server_hostname closing connection
```

Typical SMTP Conversation with Milter

```
C: Connect to server                                          *
S: 220 server_hostname ESMTP Sendmail 8.12.7/8.12.7...
C: HELO client_hostname                                       *
S: 250 server_hostname Hello client_hostname, pleased...
C: MAIL FROM:<dfs@roaringpenguin.com>                         *
S: 250 2.1.0 <dfs@roaringpenguin.com>... Sender ok
C: RCPT TO:<foo@roaringpenguin.com>                           *
S: 250 2.1.5 <foo@roaringpenguin.com>... Recipient ok
C: RCPT TO:<bar@roaringpenguin.com>                           *
S: 250 2.1.5 <bar@roaringpenguin.com>... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself           *
C: (transmits message followed by dot)
S: 250 2.0.0 h0AJVcGM007686 Message accepted for delivery
C: QUIT
S: 221 2.0.0 server_hostname closing connection
```

*** = response-modification opportunity** (green) *** = filtering opportunity** (red)

Milter C API Functions

- Initialization:
  - **smfi_register** Register a filter
  - **smfi_setconn** Specify socket
  - **smfi_settimeout** Set timeout
  - **smfi_main** Enter main milter loop
- You “register” a filter and tell what kinds of callbacks you're interested in, and whether or not you might modify the message headers and body.

A milter can communicate with Sendmail over TCP, so milter programs do not need to run on the same machine as Sendmail.

Milter API Functions: Data Access

- **smfi_getsymval** Get value of a Sendmail macro
- **smfi_getpriv** Get arbitrary private data
- **smfi_setpriv** Set arbitrary private data
- **smfi_setreply** Set SMTP reply code
- Sendmail macros lets you access a lot of useful info.
- Private data useful for storing thread-specific data.
- Can set SMTP reply to anything you like.

The **smfi_getsymval** routine gives you access to Sendmail macros. This can be quite powerful in conjunction with Sendmail rulesets. You can use rulesets to set macro values and then retrieve these macro values in the milter.

smfi_getpriv and **smfi_setpriv** let you keep per-connection state.

smfi_setreply lets you set the numeric and textual SMTP reply code and text.

Milter API: Message Modification

- **smfi_addheader** Add a header
- **smfi_chgheader** Change a header
- **smfi_addrcpt** Add a recipient
- **smfi_delrcpt** Delete a recipient
- **smfi_replacebody** Replace message body
- Recipient functions affect *envelope recipients*, not headers.

22

The **smfi_addrcpt** and **smfi_delrcpt** modify Sendmail's internal recipient list. They do not modify any message headers.

smfi_replacebody lets you replace the entire message body, and forms the basis for many content-filtering functions.

Milter API: Callbacks

- **`xxfi_connect`** Called when connection made
- **`xxfi_helo`** Called after HELO
- **`xxfi_envfrom`** Called after MAIL FROM:
- **`xxfi_envrcpt`** Called after RCPT TO:
- **`xxfi_header`** Called for each header
- **`xxfi_eoh`** Called at end of all headers
- **`xxfi_body`** Called for each “body block”
- **`xxfi_eom`** Called at end of message
- **`xxfi_abort`** Called if message aborted
- **`xxfi_close`** Called when connection closed

These callbacks are not defined in the milter library. Instead, *you* must define them in the filter. At the appropriate phase of the SMTP dialog, the callback functions are called with appropriate arguments. Their return value determines how the SMTP conversation proceeds.

In a typical SMTP conversation, the callbacks go like this:

Connect:	**`xxfi_connect`**
HELO	**`xxfi_helo`**
MAIL FROM:	**`xxfi_envfrom`**
RCPT TO:	**`xxfi_envrcpt`**
RCPT TO:	**`xxfi_envrcpt`**
DATA	(nothing)
Message sent	**`xxfi_header`**
	`xxfi_header`
	`xxfi_eoh`
	`xxfi_body`
	`xxfi_body`
	`xxfi_eom`
QUIT	**`xxfi_close`**

Milter API: Threading

- Milter library is multi-threaded, but you have no control over generation of threads.
- Milter library takes care of creating threads as required.
- Callbacks must therefore be thread-safe, and must use the **smfi_getpriv** and **smfi_getpriv** functions to maintain state.

Do not be tempted to use POSIX threads functions inside a Milter. Libmilter is free to change the way it handles threading. Instead of spawning a new thread for each connection, a future implementation of libmilter might maintain a pool of long-lived worker threads.

Callback Types

- Connection oriented: Apply to connection as a whole. **xxfi_connect**, **xxfi_helo**, **xxfi_close**.
- Recipient oriented: Apply to a single recipient only. **xxfi_envrcpt**
- Message-oriented: Apply to a message. All other callbacks are message-oriented.

Callback Return Values

- **SMFIS_CONTINUE**: Continue processing
- **SMFIS_REJECT**: For connection-oriented callback, close the whole connection. For message-oriented, reject the message. For recipient-oriented, reject only this recipient.
- **SMFIS_DISCARD**: For message- or recipient-oriented routine, silently discard message.
- **SMFIS_ACCEPT**: Accept without further filtering.
- **SMFIS_TEMPFAIL**: Return a temporary-failure code for recipient, message or connection.

Pros and Cons of Milter API

- Pro: Sendmail's implementation is not too buggy as of Sendmail >= 8.12.5.
- Pro: Quite efficient. Filters do what they need and no more.
- Pro: Can modify SMTP return codes.
- Con: Written in C. C is not the best language for parsing e-mail messages. Hard for end-users to write filters.
- Con: Multi-threaded. Easy to make dumb programming errors; some pthreads implementations leave a lot to be desired.

MIMEDefang

- MIMEDefang is a GPL'd, Perl-based mail filter. It uses Milter to interface with Sendmail.
- Used in thousands of different sites.
- Larger MIMEDefang installations process >1 million e-mail messages per day.
- MIMEDefang runs on Linux, FreeBSD, Solaris, Tru64 UNIX, HP-UX, AIX, ...
- www.mimedefang.org

Pros and Cons of MIMEDefang

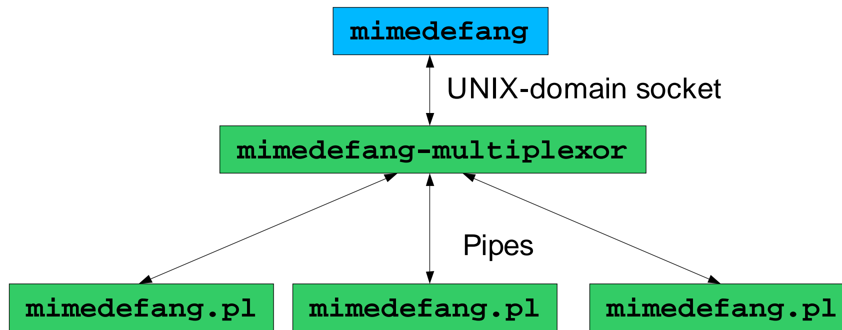
- Pro: You can write your filters in Perl instead of C.
- Pro: Filters are single-threaded. No worries about thread-safety.
- Pro: You can use all the CPAN goodies in existence to manipulate mail.
- Pro: Large user community and many ready-to-use recipes.
- Con: CPU and memory-intensive.

As you'll see, MIMEDefang uses special techniques to improve the performance of the Perl filters. In fact, it's arguable whether a C milter with all of MIMEDefang's capabilites would be much more efficient than MIMEDefang itself. String manipulation and regular-expression matching are Perl strengths, and these are what consume most of the time in most MIMEDefang installations.

Getting and Installing MIMEDefang

- Visit http://www.mimedefang.org and go to "Download"
- Make sure you have indicated prerequisites installed.
- Then it's the usual 5-step process:
  - **tar xvfz mimedefang-*version*.tar.gz**
  - **cd mimedefang-*version***
  - **./configure**
  - **make**
  - **make install**

MIMEDefang Architecture



- Multithreaded mimedefang threads talk to *single-threaded* mimedefang-multiplexor
- Multiplexor manages a pool of single-threaded Perl slaves that do the actual filtering.

We call the mimedefang process “mimedefang”.

The mimedefang-multiplexor process is called “the multiplexor”

The mimedefang.pl Perl processes are called “the slaves”.

The Multiplexor

- The multiplexor is the key to making Perl scanning work efficiently.
- Multiplexor manages Perl processes. It:
  - Feeds idle processes work.
  - Manages communication with mimedefang Milter.
  - Kills off processes after they've handled a certain number of requests (to prevent Perl memory leaks.)

Flow of Data and Commands

- When mimedefang wants some filtering done by the Perl filter, it issues a *request* to the multiplexor.
- The multiplexor finds a free slave and passes along the request.
- When the slave responds, the multiplexor passes the *response* back to mimedefang.
- The multiplexor tracks free and busy slaves.

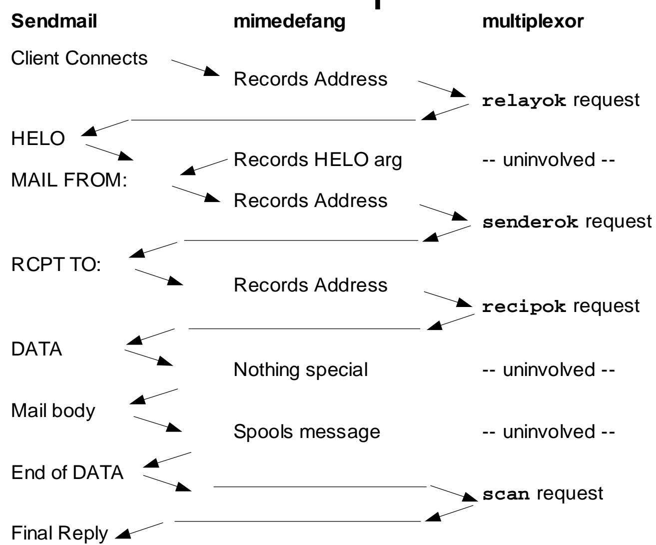
33

Not every milter callback has a corresponding request/response. For some milter callbacks, mimedefang simply notes the information in memory or in a temporary disk file. The milter callbacks are interpreted as follows:

`xxfi_connect`	**`relayok`** request.
`xxfi_helo`	Noted in memory.
`xxfi_envfrom`	**`senderok`** request.
`xxfi_envrcpt`	**`recipok`** request.
`xxfi_header`	Stored in temporary disk file.
`xxfi_eoh`	Noted in memory.
`xxfi_body`	Stored in temporary disk file.
`xxfi_eom`	**`scan`** request.
`xxfi_abort`	Clean up temprary working directory.
`xxfi_close`	Clean up temporary working directory.

The four requests above will be discussed in detail later. In brief, **`relayok`** checks whether the specified machine should be allowed to send mail. **`senderok`** checks the sender address, **`recipok`** checks each recipient address, and **`scan`** asks for the message body to be scanned. During the **`scan`** request, the message can be modified, headers can be added, changed or deleted, and recipients can be added or deleted.

Example Flow



The MIMEDefang Filter

- The **`mimedefang.pl`** program contains all the infrastructure needed to parse and manipulate MIME messages. **`mimedefang.pl`** supplies *mechanism*.
- You supply a chunk of Perl code called the filter. The filter supplies *policy*. Your filter determines exactly how MIMEDefang manipulates your e-mail.
- The distribution includes a sample filter.

The MIMEDefang Filter - 2

- Each time **`mimedefang.pl`** starts, it sources the filter file.
- During execution, **`mimedefang.pl`** calls various subroutines that you define in the filter. These subroutines generally correspond to a MIMEDefang request.
- You do not need to define all (or any) of the callbacks; if they are not defined, default no-op functions are used.

MIMEDefang Callbacks

Request	Callback Function
`relayok`	`filter_relay`
`senderok`	`filter_sender`
`recipok`	`filter_recipient`
`scan`	`filter_begin`
	`filter / filter_multipart`
	`filter_end`

Note that the **`scan`** request invokes several callback functions.

Callback Caveats

- Callbacks are processed by the *first available Perl slave*.
- That means that the **`filter_relay`** callback may occur in a *different Perl process* than the matching **`filter_sender`**.
- You cannot store state between callbacks in Perl variables.
- We'll see later how you can store state between callbacks.

38

One of the most frequent errors made by MIMEDefang novices is to set a variable in **`filter_relay`** and expect it to be available in one of the subsequent filtering functions. All of the information available to earlier filtering functions is available to later ones, so you should defer your decisions and computations until all the information you need is available, rather than trying to split it up over several callbacks.

Under unusual circumstances (for example, if a value must be calculated in **`filter_relay`** that takes a long time to compute, and might be needed later) you can store state between callbacks.

There are two things you should *never* do in your filter:

1) Call **`die`** or any other function that will terminate the Perl slave. It will cause the multiplexor to complain loudly and tempfail the mail.

2) Print anything to STDOUT. That will interfere with communication between the slave and the multiplexor, and most likely tempfail the mail. If you must print from your filter, print to STDERR.

The **`filter_relay`** Callback

- Called as: **`filter_relay(hostip, hostname)`**
- **`hostip`** is the IP address of relay.
- **`hostname`** is name (if determined by reverse-DNS lookup; otherwise **`[hostip]`**)
- Note: **`filter_relay`** is *not* called unless mimedefang invoked with **`-r`** option – saves overhead if you don't use **`filter_relay`**.

`filter_relay` Return Value

- Return value is a two-element list:
  (*code*, *message*)
- *code* is a *string*:
  - "CONTINUE" - Accept the host and keep filtering.
  - "TEMPFAIL" - Reject with a 4xx tempfail code.
  - "REJECT" - Reject with a 5xx failure code.
  - "DISCARD" - Pretend to accept, but discard message.
  - "ACCEPT_AND_NO_MORE_FILTERING" - Accept the host, and do *no further filtering* on the message.
- If *code* is "TEMPFAIL" or "REJECT", then *message* is used as the text part of the SMTP reply code.

Sample `filter_relay`

- Let's say we trust the machine 10.7.6.5 implicitly and do not want to do *any* filtering for mail from it:

```
sub filter_relay ($$) {
  my($hostip, $hostname) = @_;
  if ($hostip eq "10.7.6.5") {
      return("ACCEPT_AND_NO_MORE_FILTERING",
"");
  }
  return ("CONTINUE", "");
}
```

The **`filter_sender`** Callback

- Call: **`filter_sender(sender, hostip, hostname, helo)`**
- **`sender`** is the sender's e-mail address (as given in MAIL FROM: command)
- **`hostip`** and **`hostname`** are same as in **`filter_relay`**.
- **`helo`** is the argument to SMTP HELO or EHLO command. (We cannot tell if caller used HELO or EHLO.)
- Note: **`filter_sender`** is *not* called unless mimedefang invoked with **`-s`** option – saves overhead if you don't use **`filter_sender`**.
- Return values same as **`filter_relay`**.

Sample `filter_sender`

- Silly example: Ban mail from <nogood@baddomain.net>

```
sub filter_sender ($$$$) {
   my($sender, $hostip, $hostname, $helo) = @_;
   if ($sender eq '<nogood@baddomain.net>') {
        return ("REJECT", "I don't care for you.");
   }
   return ("CONTINUE", "");
}
```

- Note the single quotes around the e-mail address... Perl can bite you if you don't watch out. If you use double-quotes, it tries to interpolate the array **`@baddomain`**.

Resulting SMTP Conversation

```
telnet server 25
220 server ESMTP Sendmail... etc
HELO me
250 server Hello... etc
MAIL FROM:<nogood@baddomain.net>
554 5.7.1 I don't care for you.
```

The **`filter_recipient`** Callback

- Called as: **`filter_recipient(recipient, sender, hostip, hostname, first, helo, rcpt_mailer, rcpt_host, rcpt_addr)`**
- **`recipient`** is the recipient's e-mail address (as given in MAIL FROM: command)
- **`first`** is the *first* recipient of this message's e-mail address.
- **`rcpt_mailer`**, **`rcpt_host`** and **`rcpt_addr`** are the Sendmail mailer/host/addr triple associated with address.
- Rest are same as **`filter_sender`**.
- Only called if **`-t`** option used.

Sample `filter_recipient`

- Silly example: Ban mail from <nogood@baddomain.net> unless recipient is postmaster or abuse at our site.

```
sub filter_recipient ($$$$$$$$$) {
   my($recip, $sender, $rest_of_the_junk) = @_;
   if ($recip eq '<abuse@oursite.net>' or
        $recip eq '<postmaster@oursite.net>') {
        return ("CONTINUE", "");
   }
   if ($sender eq '<nogood@baddomain.net>') {
        return("REJECT", "I don't care for you.");
   }
   return ("CONTINUE", "");
}
```

Resulting SMTP Conversation

```
telnet server 25
220 server ESMTP Sendmail... etc
HELO me
250 server Hello... etc
MAIL FROM:<nogood@baddomain.net>
250 2.1.0 <nogood@baddomain.net>... Sender
ok
RCPT TO:<abuse@oursite.net>
250 2.1.5 <abuse@oursite.net>... Recipient
ok
RCPT TO:<bob@oursite.net>
554 5.7.1 I don't care for you.
```

More on Callback Return Values

- The three callback functions discussed so far can actually return a list with up to *five* elements: (code, msg, smtp_code, smtp_dsn, delay)
  - code is one of “CONTINUE”, “REJECT”, etc.
  - msg specifies the message string in the SMTP reply.
  - smtp_code is the three-digit SMTP reply code (eg 451)
  - dmtp_dsn is the SMTP DSN code (eg 4.7.1)
  - delay tells mimedefang to delay for that many seconds before replying to Sendmail. This delay ties up a milter thread, *not* a slave. Used for primitive tar-pitting.
- If you return fewer than 5 elements, mimedefang picks sensible defaults for the missing ones.

The **`scan`** request

- **`scan`** is different from other requests; it sets in motion a much more complicated set of actions.
- When the Perl slave receives a **`scan`** request, it:
  1. Reads and parses the mail message using MIME::Tools. It creates a MIME::Entity to represent the message.
  2. Calls **`filter_begin`**.
  3. For each MIME part, calls **`filter`** or **`filter_multipart`**.
  4. Calls **`filter_end`**.
- All of these actions happen in sequence in a *single slave process*.

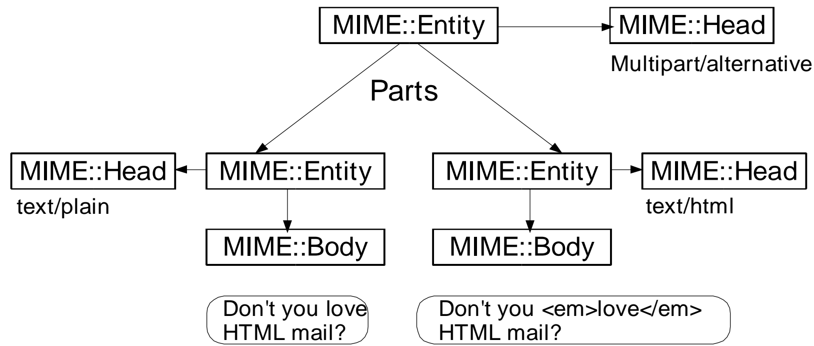
Digression: MIME::Tools

- MIME::Tools is a Perl module for parsing and generating MIME messages.
- Provides an object-oriented interface for accessing and setting MIME headers and body parts.
- Main objects are:
  - MIME::Entity, representing a MIME “entity” -- either a part or a container.
  - MIME::Head, representing the headers of a MIME entity.
  - MIME::Body, representing decoded body contents

Sample MIME Message

```
From: Someone <someone@domain.net>
To: David F. Skoll <dfs@roaringpenguin.com>
Subject: HTML Mail
MIME-Version: 1.0
Content-Type: multipart/alternative; boundary="foo"

This is a multipart message in MIME format.

--foo
Content-Type: text/plain; charset="iso-8859-1"
Content-Transfer-Encoding: quoted-printable

Don't you love HTML mail?

--foo
Content-Type: text/html; charset="iso-8859-1"
Content-Transfer-Encoding: quoted-printable

Don't you <em>love</em> HTML mail?

--foo--
```

MIME::Tools Representation



MIME::Entity

- Represents an overall MIME message. Has methods for:
  - Accessing headers (returns a MIME::Head object.)
  - Accessing body if not multipart (returns a MIME::Body object.)
  - Accessing parts if multipart (returns a list of MIME::Entity objects.)
- You can also build up a MIME::Entity with sub-parts and bodies and have MIME::Tools generate a valid MIME message.
- MIME::Tools can both *parse* and *generate* MIME.

MIME::Head

- Represents the headers of a MIME entity.
- Contains convenient methods for accessing and setting header values.
- Can parse MIME header keyword/value pairs like "charset=iso-8859-1; filename="foo.bar"
- Can set individual keyword/value pairs in headers.

MIME::Body

- Represents the body (the actual meat) of a MIME::Entity.
- Can access the *decoded* contents of the part. Very important for dealing sanely with base64 encoding, quoted-printable encoding, etc.
- Can store the body contents in memory or in a file. As used in MIMEDefang, always stores body contents in a file.
- Your filter can access the file and read (or write) the body contents.

Back to the **`scan`** request

- mimedefang passes scan information to the Perl filter in three files. Every mail message has a dedicated, unique *working directory*. Before calling your filter functions, **`mimedefang.pl`** changes to that directory.
- When the **`scan`** request is initiated, the directory contains three files:
  - **`INPUTMSG`** is the raw input mail message.
  - **`HEADERS`** contains the message headers, one per line (wrapped headers are “unwrapped”.)
  - **`COMMANDS`** conveys additional information to the filter.

The INPUTMSG File

- This file contains the entire message as received by Sendmail, except the NVT ASCII line terminators “\r\n” are replaced with newlines “\n”. Most UNIX programs are happier with this style of line termination.
- You can open and read this file from your filter if you wish.

The **HEADERS** File

- This file contains the e-mail headers, one per line. Multi-line headers are "unwrapped" and appear on a single line. For example, the header:

  ```
  Content-Type: application/pdf;
    name="foo.pdf"
  ```

  is unwrapped to:

  ```
  Content-Type: application/pdf;
  name="foo.pdf"
  ```

- This makes parsing headers simple; just open and read **./HEADERS**. Each line you read corresponds to exactly one header.

The **`COMMANDS`** File

- This file conveys extra information to the filter, such as:
  - Envelope sender and recipients.
  - Message subject and Message-ID.
  - Relay host IP address and host name.
  - Argument to HELO/EHLO command.
  - Sendmail queue identifier.
  - Values of certain Sendmail macros.
- This file is handled by **`mimedefang.pl`** and isn't for general consumption. It is documented in the mimedefang-protocol man page.

Scan phase 1: Parse

- The **`INPUTMSG`** file is parsed and split into its various MIME parts. These are stored in a subdirectory called **`Work/`**.
- Parsing is done by MIME::Tools and is transparent to your filter.

Scan phase 2: Call **filter_begin**

- The **filter_begin** function is called. It can access **INPUTMSG**, **HEADERS**, **COMMANDS** and all the message parts under **Work/**.
- Appropriate for initializing global variables and doing "global" analysis of the message. Example:

```
sub filter_begin () {
    $FoundVirus = 0;
    if (message_contains_virus()) {
        # Remember this for later!
        $FoundVirus = 1;
    }
}
```

Scan phase 3: Call `filter` and `filter_multipart`

- mimedefang.pl executes the following pseudocode:

```
foreach $entity (@All_MIME_Entities) {
   $fname = get_filename($entity);
   $ext = get_extension($fname);
   $type = get_mime_type($entity);
   if (is_multipart($entity)) {
      filter_multipart($entity, $fname, $ext, $type);
   } else {
      filter($entity, $fname, $ext, $type)
   }
}
```

Scan phase 4: Rebuild message and call **`filter_end`**

- Because the **`filter`** function can modify the message (as you'll see later), mimedefang.pl rebuilds the message as a new MIME::Entity object.
- It then calls **`filter_end`** with a single argument: The rebuilt MIME::Entity.
- This is the place for “global” message modifications that depend on information collected in earlier phases.

Built-in Variables

- During a **`scan`** request, several Perl global variables are defined:
  - **`$Sender`**: The envelope sender address.
  - **`@Recipients`**: A list of envelope recipients.
  - **`$Subject`**: Contents of the Subject: header.
  - **`$RelayAddr`**: IP address of sending relay.
  - **`$RelayHostname`**: Host name of sending relay.
  - **`$Helo`**: Arguments to SMTP HELO/EHLO command.
  - **`$QueueID`**: Sendmail Queue-ID.
  - **`$MessageID`**: Contents of Message-ID: header.
- ... plus others in mimedefang-filter man page.

Built-in Functions

- MIMEDefang has many built-in functions useful for your filter. They are divided into classes:
  - Functions to accept, modify or reject individual parts.
  - Functions that accept or bounce entire messages.
  - Virus-scanner integration routines.
  - Header modification routines.
  - Recipient modification routines (add/del recipients.)
  - Notification routines.
  - Message modification in filter_end.
  - SpamAssassin integration routines.
  - Miscellaneous functions.

Accepting or Rejecting Parts

- Inside **`filter`**, you can accept or reject parts. Example:

```
sub filter {
    my ($entity, $fname, $ext, $type) = @_;

    # Boss says .xls are OK
    return action_accept if ($ext eq '.xls');

    # But we know .exes are bad
    return action_drop_with_warning('Evil .exe removed')
        if ($ext eq '.exe');

    # Common Windoze virus vector
    return action_drop_with_warning('Evil (?) audio')
        if ($type eq 'audio/x-wav');

    # Default is action_accept if we drop off function
}
```

66

Complete list of part-oriented functions:

- **`action_accept`**: Accept the part. This is the default.
- **`action_accept_with_warning`**: Accept the part, but add a warning note to the message.
- **`action_defang`**: “Defang” the part by changing its filename and MIME type to something innocuous. Not recommended; annoys recipients.
- **`action_drop`**: Silently delete the part from the message.
- **`action_drop_with_warning`**: Delete the part from the message and add a warning note to the message.
- **`action_external_filter`**: Run the part through an external program and replace it with the filtered results.
- **`action_quarantine`**: Same as **`action_drop_with_warning`**, but also quarantines the part in a special quarantine directory.
- **`action_replace_with_url`**: Same as **`action_drop`**, but places entity in a special place where it can be served by a Web server, and puts the URL in the mail message.
- **`action_replace_with_warning`**: Replace the part with a plain-text warning. Differs from **`action_drop_with_warning`** in that the warning is placed inline instead of the part, rather than at the end of the message.,

Accepting or Rejecting Entire Message

- You can bounce or discard the entire message:

```
sub filter {
   my ($entity, $fname, $ext, $type) = @_;

   # Common virus
   if ($fname eq 'message.zip' &&
       $Subject =~ /^your account/i) {
      action_bounce('We do not accept viruses');
   }

   # Send rude message to /dev/null
   if ($Subject =~ /enlarge.*penis/i) {
      action_discard();
   }
}
```


Complete list of message-oriented functions:

- **`action_bounce`**: Reject the message with a 5xx SMTP failure code. Note that **`action_bounce`** does *not* actually *generate* a bounce message. It simply fails the SMTP transaction.
- **`action_discard`**: Accept the message with a 2xx SMTP success code, but discard it (it never gets delivered.)
- **`action_tempfail`**: Tempfail the message with a 4xx SMTP temporary-failure code.
- **`action_quarantine_entire_message`**: Quarantine the entire message, but do not affect delivery. You can follow this with one of the previous three functions if you want delivery affected.

All of these functions can be called from **`filter_begin`**, **`filter`**, **`filter_multipart`** and **`filter_end`**.

Virus-Scanner Integration

- There are two virus-scanner functions:
  - **`message_contains_virus`** scans the entire message in one invocation. This is generally quicker than scanning individual parts. If all you want to do is bounce/reject/quarantine virus messages, this is fine.
  - **`entity_contains_virus`** scans individual parts. If you just want to remove viruses but allow the rest of the message through, use the **`entity_contains_virus`** function. (But this is of dubious value with modern viruses, whose messages rarely contain anything useful.)
- Installed virus scanners are detected by configure script, and/or can be specified in your filter code.

Virus-Scanner Example

```
sub filter_begin {
   if (message_contains_virus()) {
      return action_discard();
   }
}
```

Header Modification Routines

- You can add, modify or delete headers:
  - **`action_add_header`** adds a header.
  - **`action_delete_header`** deletes a header.
  - **`action_delete_all_headers`** deletes every occurrence of the specified header.
  - **`action_change_header`** changes the value of an existing header, or inserts a new header if no such header exists.

Example of Header Modification

```
sub filter_end {
   my($entity) = @_;

   # Remove any X-MD-Host headers
   action_delete_all_headers('X-MD-Host');

   # And add our own
   action_add_header('X-MD-Host', 'mdbox.mydomain.net');
}
```

Recipient Modification Routines

- You can add or delete envelope recipients:
  - **add_recipient** adds a recipient.
  - **delete_recipient** deletes a recipient.
- Note: These routines affect only *envelope* recipients. That is, they affect mail delivery, but do *not* modify headers. They don't adjust the Cc: or To: headers, for example, nor do they adjust **@Recipients**.

Example of Recipient Modification

```
sub filter_end {
   my($entity) = @_;

   # Put spammish messages in spamtrap
   if (looks_like_spam()) {
      my $recip;
      foreach $recip (@Recipients) {
         delete_recipient($recip);
      }
      add_recipient('spamtrap@mydomain.net');
   }

   # We archive everything!
   add_recipient('archive-bot@mydomain.net');
}
```


The **`looks_like_spam()`** function is an example; it's not part of MIMEDefang. We'll get to spam detection in a later section.

Notification Routines

- **`action_notify_administrator`** – send the MIMEDefang administrator a notification message.
- **`action_notify_sender`** – send the original sender a notification message.
- Use **`action_notify_sender`** with *great care*. In particular, do *not* notify senders of viruses, because modern viruses fake the sender address and you just end up irritating innocent third parties.

Example of Administrator Notification

```
sub filter_end {
   my($entity) = @_;

   if (message_contains_virus_clamd()) {
      my $qdir = get_quarantine_dir();
      action_quarantine_entire_message();
      action_bounce("Found the $VirusName virus");
      action_notify_administrator(
         "Found the $VirusName in message in $qdir");
}
```


Presumably, **`$FoundHTML`** is set elsewhere in the filter.

Example of Sender Notification

```
sub filter_end {
    my($entity) = @_;

    if ($FoundHTML &&
        grep { lc($_) eq '<md@lists.roaringpenguin.com>' } @Recipients){
        action_notify_sender('HTML not permitted on the list');
        action_discard();
    }
}
```

Message Modification in **`filter_end`**

- In **`filter_end`**, you can make structural modifications to the entire MIME message.
- In **`filter`**, you can only drop or replace individual MIME parts one part at a time.
- In **`filter_end`**, you can "look" at the entire MIME message all at once and modify its structure.
- **`filter_end`** is passed a MIME::Entity representing the *rebuilt*, possibly modified, message.

Message Modification in `filter_end` (2)

- **`action_add_part`** – add a MIME part to the message.
- **`append_html_boilerplate`** – append boilerplate to HTML MIME parts.
- **`append_text_boilerplate`** – append boilerplate to plain-text MIME parts.
- **`remove_redundant_html_parts`** – remove text/html parts that have corresponding text/plain parts
- **`replace_entire_message`** – replace entire message with your own MIME::Entity.

Example of Modification in `filter_end`

```
sub filter_end {
   my($entity) = @_;

   # If we have both plain-text and HTML, nuke HTML
   remove_redundant_html_parts($entity);

   # Sigh... lawyers insist on this
   if (message_is_outgoing()) {
      append_text_boilerplate($entity,
         'Silly legal boilerplate', 0);
      append_html_boilerplate($entity,
         '<em>Silly</em> legal boilerplate', 0);
   }
}
```


The **`message_is_outgoing()`** function is not built into MIMEDefang. Presumably, it's a function you would write that would look at **`$Sender`** and **`@Recipients`** and determine if mail is outgoing (or use the relay IP address to make its decision.)

The **`append_html_boilerplate`** function tries to be reasonably smart. It parses the HTML and puts your boilerplate just before the </body> tag if it can.

SpamAssassin™ Integration

- SpamAssassin™ is generally acknowledged as one of the best spam-detection systems.
- SpamAssassin uses many techniques to detect spam:
  - Header tests: E.g. invalid Message-IDs, fake MUA lines, fake Received: headers
  - Body tests: E.g. obfuscating HTML comments, spam phrases, body disguised with Base64 encoding.
  - Network tests: E.g. RBL lookups, SURBL lookups, Razor lookup and reporting, DCC lookup and reporting.
  - Bayesian analysis: Statistical analysis from a learned corpus of mail.

“SpamAssassin” is a trademark of Network Associates Inc.

SpamAssassin Integration

- All of the SpamAssassin tests are assigned weights, and if the weighted sum of rules that trigger exceeds a threshold, the message is likely spam.
- Weights are determined with a genetic algorithm run against a huge corpus of mail. The weights are adjusted to minimize error.
- SpamAssassin is written in Perl, and can be used as both a Perl module and as command-line tools. MIMEDefang integrates directly at the Perl module level.

SpamAssassin Integration

- MIMEDefang uses SpamAssassin only to *detect* spam. The SpamAssassin facilities for modifying messages are *not* used by MIMEDefang. If you want to modify a message (for example, tag the subject), you need to do it using MIMEDefang's functions (like **`action_change_header`**).
- Using Bayesian filtering with MIMEDefang is tricky, because MIMEDefang runs only as the “defang” user. However, it is possible to use a system-wide Bayes database with MIMEDefang and SpamAssassin.

SpamAssassin Integration

- Simplest function: **`spam_assassin_is_spam`** returns true if spam score is above configured threshold; false otherwise.
- Next up in complexity and detail: **`spam_assassin_check`** returns four value: hits (the actual score), required (the spam threshold), tests (a list of SpamAssassin tests that fired) and report (the SpamAssassin report as a string.)
- To get into nitty-gritty of SpamAssassin, call **`spam_assassin_init`**, which returns a Mail::SpamAssassin object for you to manipulate.

Other SpamAssassin-related functions are **`spam_assassin_mail`**, which returns a Mail::NoAudit object for feeding the the SpamAssassin perl module, and **`spam_assassin_status`**, which returns a Mail::SpamAssassin::PerMsgStatus object for the current message.

These functions hook into the low-level SpamAssassin library, giving you more access to and control over SpamAssassin.

Example of SpamAssassin Integration

```
sub filter_end {
   my($entity) = @_;
   my($hits, $req, $names, $report) = spam_assassin_check();
   my $stars = ($hits < 15) ? ("*" x int($hits)) : ("*" x 15);

   # Bounce anything scoring twice the spam threshold
   if ($hits >= 2 * $req) {
      action_bounce("No spam wanted here.");
      return;
   }

   # Add spam-score header
   action_change_header("X-Spam-Score", "$hits ($stars) $names");

   # Tag subject if over threshold
   if ($hits >= $req) {
      action_change_header("Subject", "[Spam: $hits] $Subject");
   }
}
```

Checking Addresses for Existence

- Many e-mail systems use a filtering server that relays to actual mail server.
- Filtering server may not know all valid internal e-mail addresses. It must relay for anything@domain.net
- Problem: Mail for invalid recipients is accepted by filtering server, but rejected by internal machine. Filtering server is now responsible for generating and mailing a DSN.
- Most of these DSNs sit in the queue or bounce.

Checking Existence - 2

- MIMEDefang has a function called **`md_check_against_smtp_server`** that runs a "mini-SMTP" dialog for each RCPT command.
- If the internal server would reject a recipient, then so does MIMEDefang. This lets you reject unknown recipients at RCPT TO: time on the perimeter.
- Does not work with MS Exchange before 2003, and even 2003 requires hoops: http://hellomate.typepad.com/exchange/2003/09/exchange_2003_r.html

Checking Existence - 3

- Alternatively, you can validate addresses on the perimeter by:
  - Writing Perl code to connect to an LDAP or SQL database.
  - Playing Sendmail virtusertable tricks.

Miscellaneous Functions

- **message_rejected()** returns true if message has been rejected (eg, by **action_discard**, etc.) Use it to save computation:

```
sub filter {
   my($entity, $fname, $ext, $type) = @_;

   # No point in doing work if message already rejected
   return if message_rejected();

   # Do heavy computations now...
   # ... etc ...
}
```

Miscellaneous Functions

- **`read_commands_file()`** makes some global variables normally available only after **`filter_begin`** available to **`filter_relay`**, **`filter_sender`** and **`filter_recipient`**.
- Obviously, only those variables whose values are known will be available – you can't use **`$Subject`** until **`filter_begin`**, for example.
- Consult mimedefang-filter man page for details.

Advanced Techniques

- You can use all of Perl's power to write sophisticated filters.
- I will show three examples of more sophisticated filters:
  - A filter that preserves relay information across secondary MX hosts.
  - A partial implementation of "Greylisting".
  - A simple-minded per-class-C throttle.

Preserving Relay Information

- Most organizations use (or should use) secondary MX hosts in case the primary MX host is down.
- If the secondary MX host relays to the MIMEDefang machine, then the relay address is not directly useful.
- However, we can detect mail from a secondary MX host and extract the “real” relay IP address from the Received: headers.
- We just defer host processing until **`filter_begin`**.

Preserving Relay Information

```
1  @Relayers = ("127.0.0.1", "192.168.2.10", "192.168.3.15");

2  sub from_secondary_mx ($) {
3      my($ip) = @_;
4      if (grep { $ip eq $_ } @Relayers) {
5          return 1;
6      }
7      return 0;
8  }
```


Line 1: Declare a list of our secondary MX hosts. These are IP addresses *as seen by the MIMEDefang machine*.

Lines 2-3: Subroutine entry point. We'll return 1 if argument is a secondary MX host; 0 if it is not.

Lines 4-6: Return 1 if argument is found in the list of secondary MX hosts.

Line 7: Return 0 if argument was not found in list of secondary MX hosts.

Preserving Relay Information

```
 1  sub get_real_relay_from_headers () {
 2      open(HDRS, "<./HEADERS") or return undef;
 3      while(<HDRS>) {
 4          chomp;
 5          next unless /^Received:\s*from/i;
 6          if (/\(\[(\d+\.\d+\.\d+\.\d+)\]\)/  or
 7              /\s+\[(\d+\.\d+\.\d+\.\d+)\]\)/ or
 8              /\[(\d+\.\d+\.\d+\.\d+)\]/      or
 9              /\((\d+\.\d+\.\d+\.\d+)\)/      or
10              /\@(\d+\.\d+\.\d+\.\d+)/        or
11              /\s+(\d+\.\d+\.\d+\.\d+)\s+/    or
12              /(\d+\.\d+\.\d+\.\d+)/) {
13              my $addr = $1;
14              if (!from_secondary_mx($addr)) {
15                  close(HDRS);  return $addr;
16              }
17          } else {
18              close(HDRS);  return undef;
19          }
20      }
21      close(HDRS);
22      return undef;
23  }
```


Line 1: Subroutine entry point.

Line 2: Open the ./HEADERS file, which contains one header per line.

Line 3: Iterate over all the headers.

Line 5: Ignore any header that doesn't begin with “Received: from”

Lines 6-12: Pick out a “likely-looking” relay IP address. If an IP address is of the form ***a.b.c.d***, we look for these patterns in order of preference. A **<space>** indicates a whitespace character; other characters are literal.

- **([*a.b.c.d*])**
- **<space>[*a.b.c.d*])**
- **[*a.b.c.d*]**
- **@*a.b.c.d***
- **<space>*a.b.c.d*<space>**
- ***a.b.c.d***

Lines 14-16: If we found an IP address that is *not* one of our secondary MX hosts, return it. This allows for a chain of secondary MX hosts.

Lines 17-19: If we failed to find any IP address at all in the Received: header, then it is too strange for us to parse and we give up. It is important to give up; if we do not, we could end up parsing the next Received: header down and returning incorrect information.

Lines 20-23: Give up if no Received: header found or no non-secondary-MX IP address found.

Preserving Relay Information

```
 1  sub filter_relay ($$$) {
 2      my($hostip, $hostname, $helo) = @_;
 3      if (from_secondary_mx($hostip)) {
 4          # From a secondary MX host - defer processing
 5          return('CONTINUE', 'OK');
 6      }
 7      # Do real relay-checking here...
 8  }
 9  sub filter_begin () {
10      if (from_secondary_mx($RelayAddr)) {
11          my $real_relay = get_real_relay_from_headers();
12          if (defined($real_relay)) {
13              $RelayAddr = $real_relay;
14              # Do real relay-checking here...
15          }
16      }
17  }
```


In this example, **`filter_relay`** does not do any host-IP-based filtering if mail is coming from a secondary MX host. Instead, filter_begin checks if mail is coming from a secondary MX host. If it is, then host-IP-based filtering is done in **`filter_begin`**. This defers the IP-based filtering until after the DATA phase.

Greylisting

- Some spammers use special *ratware* to send out spam. The goal of ratware is to send massive amounts of e-mail out quickly, not to ensure reliable delivery of any particular message.
- Some ratware *ignores* SMTP error codes. If a message is failed, the ratware never retries.
- Legitimate mail servers always retry a message that has experienced a temporary failure (4xx SMTP reply code.)

Greylisting (2)

- By tempfailing mail from an unknown sender the very first time, we can detect ratware (and prevent delivery.) A legitimate mail server will almost always retry, leading to practically no false-positives. We call this "hit-and-run detection."
- Hit-and-run detection was introduced in our CanIt product in January 2003, and expanded upon by Evan Harris at http://projects.puremagic.com/greylisting

Greylisting (3)

- Full greylisting takes into account the Sender/Recipient/Relay-Address tuple, and imposes minimum and maximum bounds on retransmission time.
- We show a simplified version that only takes into account sender and recipient. The code:

```
sub filter_recipient ($$$$$$$$$) {
    my($recip, $sender, $rest_of_the_junk) = @_;
    if (should_greylist($sender, $recip)) {
        return("TEMPFAIL",
        "Tempfailed as anti-spam measure.  Please try again.");
    }
    return ("CONTINUE", "");
}

$DBFile = "/var/spool/MIMEDefang/greylist.db";
1;
```

should_greylist

```
 1  sub should_greylist ($$) {
 2      my($sender, $recip) = @_;
 3      my %hash;
 4      $sender = canonicalize_email($sender);
 5      $recip  = canonicalize_email($recip);
 6      my $key = "<$sender><$recip>";
 7      lock_db();
 8      tie %hash, 'DB_File', $DBFilename;
 9      my $ret = ++$hash{$key};
10      untie %hash;
11      unlock_db();
12      return ($ret == 1);
13  }
```

98

Lines 1-2: Subroutine entry point. Arguments are sender and recipient.

Line 3: Declare a local variable that we'll tie to the Berkeley DB

Lines 4-5: Strip angle brackets off e-mail addresses and make lower-case.

Line 6: Construct a key for lookup in the Berkeley DB.

Line 7: Lock the DB (other slaves might try to access it concurrently.)

Line 8: Tie our hash to the Berkeley DB. Now, accessing %hash accesses the disk database.

Line 9: Increment the number of times we've seen this sender/recipient combination. Thanks to Perl, if the key isn't in the database, it automagically gets inserted with a value of 1.

Linues 10-11: Disconnect from the database and unlock it.

Line 12: Return true if this is the *first* time we've seen this sender/recipient pair.

canonicalize_email

```
1 sub canonicalize_email ($) {
2     my($email) = @_;
3     # Remove angle-brackets; convert to lower-case
4     $email =~ s/^<//;
5     $email =~ s/>$//;
6     $email = lc($email);
7     return $email;
8 }
```


Lines 1-2: Subroutine entry point. Argument is an e-mail address.

Line 4: Remove leading angle bracket, if any.

Line 5: Remove trailing angle bracket, if any.

Line 6: Convert to lower-case.

Line 7: Return canonical value.

lock_db and unlock_db

```
 1 sub lock_db () {
 2     open(LOCKFILE, ">>$DBFilename.lock") or return 0;
 3     flock(LOCKFILE, LOCK_EX);
 4     return 1;
 5 }

 6 sub unlock_db () {
 7     flock(LOCKFILE, LOCK_UN);
 8     close(LOCKFILE);
 9     unlink("$DBFilename.lock");
10     return 1;
11 }
```


Line 2: Open a lock file.

Line 3: Acquire an exclusive lock on the file.

Line 7: Relinquish lock on lock file.

Line 8: Close LOCKFILE descriptor.

Line 9: Remove lock file.

NOTE: No error checking!

Obvious enhancements

- Implement full greylisting with minimum/maximum allowed retry times.
- Add Relay-IP to the key.
- Make exceptions (no point in greylisting mail from 127.0.0.1 or friendly hosts.)
- Add error-checking.
- Use a SQL database rather than Berkeley DB. This makes it easier to run multiple MX hosts sharing the same greylisting database.

Per-Class-C Throttle

- This simple filter limits a given class C network to sending 50 mails in 10 minutes. We use a sliding window updated once a minute to keep track of attempts per class C network.
- This filter will illustrate how to extend the MIMEDefang protocol, and will serve as an introduction to the **`md-mx-ctrl`** program. **`md-mx-ctrl`** allows you to inject commands into a running multiplexor.

Per-Class-C Throttle - filter_sender

```
 1  use DB_File;
 2  use Fcntl ':flock';

 3  $DBFilename = "/var/spool/MIMEDefang/throttle.db";
 4  sub filter_sender ($$$$) {
 5      my($sender, $hostip, $junk) = @_;

 6      # Chop off last byte -- only care about class C network
 7      $hostip =~ s/\.\d+$//;
 8      if (count_attempts($hostip, 1) > 50) {
 9          return('TEMPFAIL',
10          "Rate-limiting in effect for network $hostip");
11      }
12      return('CONTINUE', "OK");
13  }
```

103

Lines 1-2: Load Berkeley DB and file locking Perl modules.

Line 3: Set name of Berkeley DB file.

Line 7: Keep only the first three bytes of IP address of relay host.

Lines 8-11: If we've had more that 50 attempts from class C network in the last 10 minutes, tempfail the MAIL FROM: command.

Line 12: Otherwise, allow SMTP transaction to continue.

Per-Class-C Throttle - count_attempts

```
 1  sub count_attempts ($$) {
 2      my($ip, $do_incr) = @_;
 3      my $attempts = 0;
 4      my %hash;
 5      my $time;
 6      my $i;
 7      $time = time_in_minutes();

 8      lock_db();
 9      tie %hash, 'DB_File', $DBFilename;
10      $hash{"$ip:" . $time}++ if $do_incr;
11      for ($i=0; $i<10; $i++) {
12          my $x = $hash{"$ip:" . ($time-$i)};
13          $attempts += $x if defined($x);
14      }
15      print STDERR "DEBUG: attempts($ip) = $attempts\n";
16      unlock_db();
17      untie %hash;
18      return $attempts;
19  }
```


Lines 1-2: Subroutine entry. We are given two arguments: $ip is the class-C network, and $do_incr is a flag indicating whether or not to increment the number of attempts for this network.

Line 7: Get an integer representing current time in minutes since midnight, January 1st, 1970.

Lines 8-10: Usual Berkeley DB connection sequence.

Lines 11-14: Sum the attempts in the last 10 minutes.

Line 15: Log some debugging output.

Lines 16-19: Disconnect from Berkeley DB and return answer.

The Berkeley DB holds buckets indexed by {Class-C-Addr, Time-In-Minutes}. We increment the current bucket, and sum the most recent 10 buckets, for each class C address.

Per-Class-C Throttle - time_in_minutes

```
1  sub time_in_minutes () {
2      return int(time()/60);
3  }
```


Time_in_minutes is dead simple: Get time in seconds and divide by 60.

Per-Class-C Throttle - clean_database

```
 1  sub clean_database () {
 2      my %hash;
 3      my $thing;
 4      my $n = 0;
 5      my $now = time_in_minutes();
 6      lock_db();
 7      tie %hash, 'DB_File', $DBFilename;
 8      foreach $thing (keys %hash) {
 9          my($ip, $time) = split(/:/, $thing);
10          if ($time < $now - 9) {
11              delete($hash{$thing});
12              print STDERR "Cleaning $thing\n";
13              $n++;
14          }
15      }
16      unlock_db();
17      untie %hash;
18      return $n;
19  }
```


If we don't clean the database periodically, it will grow without bound.

Lines 1-7: Connect the Berkeley DB

Lines 8-15: Iterate over every key in the database. If it is for a bucket 10 minutes old or older, delete the key.

Lines 16-19: Disconnect from Berkeley DB and return a count of the keys cleaned up from the database.

Per-Class-C Throttle - filter_unknown_cmd

```
 1  sub filter_unknown_cmd ($) {
 2      my($cmd) = @_;
 3      if ($cmd eq "cleandb") {
 4          my $ncleaned = clean_database();
 5          return ("ok", $ncleaned);
 6      } elsif ($cmd =~ /count\s+(\S+)/) {
 7          my $attempts = count_attempts($1, 0);
 8          return ("ok", $attempts);
 9      }
10      return("error:", "Unknown command");
11  }
```


If you define a function called filter_unknown_cmd, then if the slave receives a request it doesn't understand, it passes the request on to filter_unknown_cmd. Here, we define two requests:

Lines 3-5: If the “cleandb” request comes in, we expire old data from the database.

Lines 6-9: If a “count *class_c_addr*” request comes in, we return the count for the specified Class C address.

Sendmail's SOCKETMAP

- Many parts of Sendmail's processing involve *map lookups*. These are database lookups that look up a value given a key. Examples:
  - Access database lookups.
  - Alias lookups.
  - Virtusertable lookups.
- Map lookups are commonly implemented as Berkeley DB lookups or LDAP lookups.
- Sendmail 8.13 has a new map type called SOCKETMAP that uses a simple protocol over a TCP or UNIX-domain socket to talk to a mapping daemon.

Communication between Sendmail and the SOCKETMAP server does not use the Milter protocol. Sendmail documents (yet another) wire protocol for this lookup. It just turns out to be quite convenient to re-use the multiplexor to handle SOCKETMAP lookups.

Sendmail's SOCKETMAP - 2

- MIMEDefang lets you implement a socketmap lookup by defining the **`filter_map`** function and invoking the multiplexor with the **`-N`** option (to specify the socket.)
- This lets you implement a map in Perl! You can do cool stuff like SQL database lookups, computational lookups, etc...
- **`filter_map`** runs outside the context of an SMTP session, so most mail-related global variables are unavailable.

SOCKETMAP Example

- Sendmail config file example:

```
V10/Berkeley
Kmysock socket unix:/var/spool/MIMEDefang/map.sock
Ksock2 socket unix:/var/spool/MIMEDefang/map.sock
```

- MIMEDefang filter example (silly one...)

```
sub filter_map ($$) {
  my($mapname, $key) = @_;
  if ($mapname eq "mysock") {
    my $ans = reverse($key);
    return ("OK", $ans);
  }
  return("PERM", "Unknown map $mapname");
}
```

SOCKETMAP Example

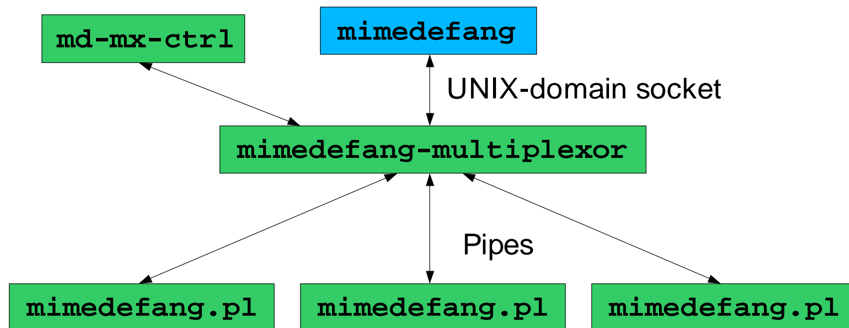
- And here is a Sendmail test session:

```
$ sendmail -C ./socketmap.cf -bt
No local mailer defined
ADDRESS TEST MODE (ruleset 3 NOT automatically invoked)
Enter <ruleset> <address>
> /map mysock Testing...
map_lookup: mysock (Testing...) returns ...gnitseT (0)
> /map sock2 foo
map_lookup: sock2 (foo) no match (69)
```

md-mx-ctrl

- The program **md-mx-ctrl** lets you "manually" send a request to the multiplexor.
- Some special requests are handled directly by the multiplexor rather than being passed to a Perl slave.
- All other requests are passed to a free Perl slave.
- **md-mx-ctrl** lets you "pretend" to be **mimedefang**. The multiplexor treats a request from **md-mx-ctrl** just like any other request.

Augmented Architecture



- **`md-mx-ctrl`** lets you inject requests and read responses from the multiplexor.

`md-mx-ctrl` special requests

- **`free`**: Print the number of free slaves.
- **`status`**: Print the status of all slaves.
- **`rawstatus`**: Print the status of all slaves in machine-readable form (used by watch-mimedefang.)
- **`reread`**: Force slaves to reread filter rules at earliest convenient opportunity. Idle slaves are killed; busy slaves are killed as soon as they become idle.
- **`msgs`**: Print number of messages processed since multiplexor started.

`md-mx-ctrl` special requests - 2

- **`load`**: Print many useful statistics.
- **`slaves`**: List slaves and their PIDs
- **`rawload`**: Same as **`load`** but in a machine-parseable format.
- **`barstatus`**: Print busy slaves in a “bargraph” format.

`md-mx-ctrl` special request examples

```
md-mx-ctrl free
3

md-mx-ctrl status
Max slaves: 4
Slave 0: idle
Slave 1: busy
Slave 2: idle
Slave 3: stopped

md-mx-ctrl rawstatus
IBIS 1 3 30 0
```


The format of the **`rawstatus`** output is documented in the **`md-mx-ctrl (8)`** man page.

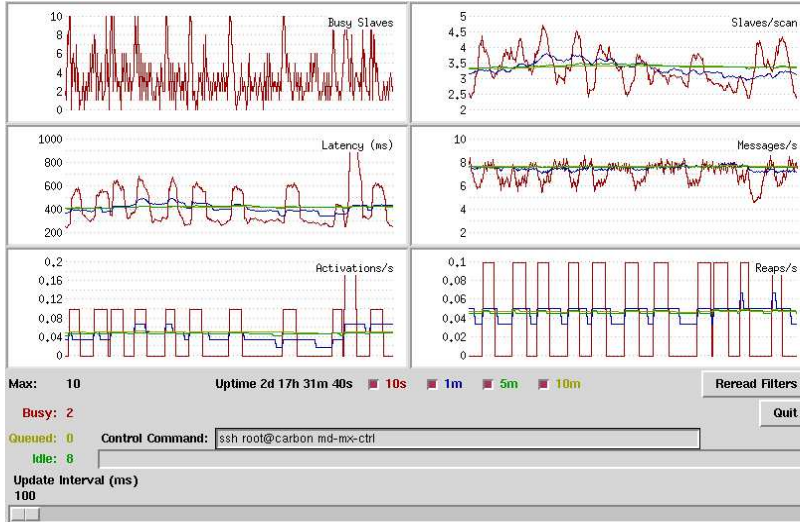
`md-mx-ctrl` special request examples - 2

```
md-mx-ctrl load
Load:       Msgs:   Msgs/Sec:  Avg ms/scan: Avg Busy Slaves:
10 Sec        52        5.20        703.6        4.40
 1 Min       428        7.13        430.9        3.30
 5 Min      2224        7.41        396.7        3.25
10 Min      4483        7.47        407.0        3.29

for i in `seq 1 5` ; do md-mx-ctrl barstatus; sleep 1; done
  3/10 BBB.......    0/30 .................... 7141
  2/10 BB........    0/30 .................... 7149
  5/10 BBBBB.....    0/30 .................... 7156
  4/10 BBBB......    0/30 .................... 7166
  3/10 BBB.......    0/30 .................... 7175
```

`md-mx-ctrl` and **`watch-mimedefang`**

- A Tcl/Tk program called **`watch-mimedefang`** uses **`md-mx-ctrl`** to monitor MIMEDefang graphically.
- Can monitor remote machines over a low-bandwidth SSH connection. Can even run this on Windows with a Windows SSH program like putty.

`md-mx-ctrl` and `watch-mimedefang`



`md-mx-ctrl` extended requests

- Our sample Class C throttle filter added the **`cleandb`** and **`count`** requests:

```
md-mx-ctrl cleandb
ok 17

md-mx-ctrl count 192.168.5
ok 22
```

- Caveat: You have no control over which Perl slave runs the request. There is no way to broadcast a request to *all* Perl slaves.

Different Strokes for Different Folks

- One of the top-5 FAQs: “How do I get MIMEDefang to use different rules for different recipients?”
- In general, very difficult. SMTP has no way to indicate selective success/failure after the DATA phase. There's no way (after DATA) to say “Message delivered to X, Y and Z. Message bounced for U, V and W. Message tempfailed for P and Q.”

Streaming

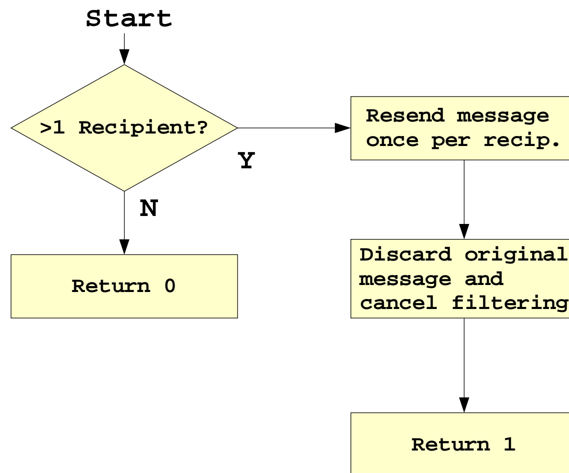
- MIMEDefang implements a *hack* to allow per-recipient (or per-domain) customizations. First, the code and then the explanation:

```
sub filter_begin () {
    if (stream_by_recipient()) {
        return;
    }
    # At this point, @Recipients
    # contains a single entry
}
```

How Streaming Works

- If **`@Recipients`** contains *one* entry, then **`stream_by_recipient()`** does nothing, and returns 0.
- If **`@Recipients`** contains *more than one* entry, then **`stream_by_recipient()`** re-mails copies of the original e-mail individually to each recipient. It then calls **`action_discard()`** and returns 1. It also prevents *any further* filtering of original e-mail – there are no further calls to **`filter`** or **`filter_end`** for the original e-mail.
- The key to making this work is that Sendmail 8.12 in its default configuration *re-submits* the re-sent messages via SMTP.

How Streaming Works - Diagram



Streaming Example

- Do not filter mail for **`abuse@mydomain.net`**

```
sub filter_begin () {
    return if (stream_by_recipient());
}

sub filter ($$$$) {
   my($entity, $fname, $ext, $type) = @_;
   # Do not filter for abuse
   if (canonicalize_email($Recipients[0])
        eq 'abuse@mydomain.net') {
      return;
   }

   # Normal filtering rules here...
}
```

- Even if mail comes in for two recipients, abuse's mail will *not* be filtered, but the other recipient's *will* be filtered.

Streaming Downsides

- Mail must be re-sent, so it's harder to determine original relay IP (resent mail appears to come from 127.0.0.1). MIMEDefang has another hack to work around this; consult the man pages. I present an alternate approach to get the real relay IP later on in this talk.
- If a re-sent mail is rejected, your machine is responsible for generating a bounce message. Can no longer get away with just an SMTP failure code.
- Mail is re-sent in deferred mode (Sendmail's **`-odd`** switch – queue the mail, send during the next queue run.) This delays streamed mail.

Exercise for the reader: Consider what happens if we re-send mail in interactive or background mode rather than deferred mode. Explain why the decision was made to resend mail in deferred mode.

Per-User SpamAssassin Settings

- Streaming allows for per-user SpamAssassin settings.
- You can obtain the SpamAssassin object used by MIMEDefang and adjust its settings based on recipient address.
- This is tricky and is an exercise left for the audience. See the mimedefang-filter(5) and Mail::SpamAssassin(3) man pages.

Streaming by Domain

- Another common scenario is different settings for different domains, but the same setting within a domain. **`stream_by_domain`** is similar to **`stream_by_recipient`**, but groups recipients by domain.
- Example: If mail comes in for **`a@a.net, b@a.net`**, **`c@b.net`** and **`d@c.net`**, then three copies are sent out: One to **`a`** and **`b`**, one to **`c`**, and one to **`d`**.
- Global variable **`$Domain`** is set if all recipients are in the same domain.

Streaming by Domain Example

```
sub filter_begin () {
    if (stream_by_domain()) {
        return;
    }
    if ($Domain eq 'a.net') {
        # Rules for a.net
    } elsif ($Domain eq 'b.net') {
        # Rules for b.net
    } else {
        # Rules for everyone else
    }
}
```

The Notification Facility

- *Notification* here means letting a program know when something “interesting” has happened. It doesn't refer to notifying a person by e-mail.
- The multiplexor defines “interesting” events as:
  - A slave has been killed because of a busy timeout
  - The number of free slaves has changed
  - Someone has requested a filter reread
  - A slave has died unexpectedly
  - The number of free slaves has dropped to zero
  - The number of free slaves was zero, but is now non-zero.

The Notification Facility - 2

- The notification facility sends simple one-line ASCII messages over the socket.
- Programs connecting to the notifier can specify which messages they are interested in. The notifier will only send those messages.
- The messages are:
  - **`B`** – There was a busy timeout
  - **`F`** ***n*** – There are *n* free slaves
  - **`R`** – Someone has requested a filter re-read
  - **`U`** – A slave has died unexpectedly
  - **`Y`** – the number of free slaves went from 0 to 1
  - **`Z`** – the number of free slaves went from 1 to 0

The Notification Facility - 3

- Examples:
  - If a slave dies unexpectedly or a busy timeout occurs, your program could page you.
  - You could log statistics to have a complete history of the number of busy slaves over time.
  - You can reject SMTP connections when there are no free slaves, and accept them again when there are free slaves.
- Documented in mimedefang-notify(7) man page.

The Notification Facility – Linux-specific example - 1

```
#!/usr/bin/perl -w
#
# On Linux, prepare to use this script like this:
#  /sbin/iptables -N smtp_conn
#  /sbin/iptables -A INPUT --proto tcp --dport 25 --syn -j smtp_conn
# Then run the script as root.

use IO::Socket::INET;

sub no_free_slaves {
    print STDERR "No free slaves!\n";
    # Reject new connections on port 25 (reject SYN packets)
    system("/sbin/iptables -A smtp_conn -j REJECT");
}

sub some_free_slaves {
    print STDERR "Some free slaves.\n";
    # Accept new connections again on port 25
    system("/sbin/iptables -F smtp_conn");
}
```

The Notification Facility – Linux-specific example - 2

```
sub main {
    my $sock;

    $sock = IO::Socket::INET->new(PeerAddr => '127.0.0.1',
                                  PeerPort => '4567',
                                  Proto => 'tcp');
    # We are only interested in Y and Z messages
    print $sock "?YZ\n";
    $sock->flush();
    while(<$sock>) {
        if (/^Z/) {
            no_free_slaves();
        }
        if (/^Y/) {
            some_free_slaves();
        }
    }
    # EOF from multiplexor?? Better undo firewalling
    system("/sbin/iptables -F smtp_conn");
}

main();
```

The Tick Facility

- Sometimes, you would like to run a task periodically. For example, you might want a database maintenance or cleanup job to be run every so often.
- The multiplexor can issue a *tick* request every so often. This runs a function you define called **`filter_tick`**.
- The multiplexor ensures that at most *one* tick request is outstanding at any given time – you'll never have two simultaneously-executing **`filter_tick`**s.

The Tick Facility - 2

- You cannot predict which slave will run a tick request.
- If all slaves are busy when it is time to run a tick, the request is simply *skipped*. The multiplexor tries again at the next regularly-scheduled tick time.
- Tick requests run outside the context of an SMTP connection, so most mail-related global variables are not available.

Common Spammer Tactics 1

- Fake the HELO string to be from your domain, or even your IP address. Defense:

```
# Assume my IP address is 192.168.1.2
sub filter_sender {
    my ($from, $hostip, $hostname, $helo) eq @_;
    if ($helo =~ /mydomain\.net$/ or
        $helo eq '192.168.1.2') {
        return('REJECT', 'Faked HELO');
    }
    return('CONTINUE', 'OK');
}
```

- Do not use on server used to send internal mail out! In this case, HELO can legitimately be from your domain. (Or check **`$hostip`**.)

Common Spammer Tactics 2

- Fake sender to be from a big-name ISP. We can do *selective and judicious* sender-relay mismatch tests:

```
sub filter_sender {
    my ($from, $hostip, $hostname, $helo) eq @_;
    $from = canonicalize_email($from);
    if ($from =~ /\@hotmail\.com$/ and
        !($hostname =~ /hotmail\.com$/)) {
        return('REJECT',
    "Mail from hotmail.com not accepted from $hostname");
    }
    return('CONTINUE', 'OK');
}
```

- Used sparingly, this is highly effective. It may reject valid e-mail, but you should educate users to relay via Hotmail for their Hotmail accounts.

The mismatch test is effective for large providers like Hotmail and Yahoo that have good reverse-DNS setups. In general, however, there is no good reason to assume that the sending relay's hostname should end in the same domain as the envelope from address, so don't overdo this rule.

Sender Policy Framework (SPF - http://spf.pobox.com) is designed to allow domains to publish their list of outgoing relays; it is a more accurate test than a “mismatch” test. Unfortunately, SPF is still not widely deployed, and it has its share of critics.

Common Spammer Tactics 3

- Fake sender address from *your* domain. How cheeky! Unfortunately, you end up dealing with bounces.
- We suggest publishing *receive-only* addresses on your Web site. For example, **`sales@canit.ca`** and **`info@canit.ca`** are used only to receive mail, never to send it. (We send from personal addresses.)
- Receive-only addresses should *never* receive bounces.

Rejecting Spurious Bounces

```
sub filter_recipient {
    my($recipient, $sender, $junk) = @_;
    $recipient = canonicalize_email($recipient);
    if ($sender eq '<>') {
        if ($recipient eq 'info@canit.ca' or
            $recipient eq 'sales@canit.ca') {
        return('REJECT',
        "$recipient is a receive-only address");
    }
    return('CONTINUE', 'OK');
}
```

Sendmail Macros

- Several Sendmail macros are available during the **scan** request in the **%SendmailMacros** hash. Example:

```
# Do not filter mail from authenticated users
sub filter_end {
    my($entity) = @_;
    if (defined($SendmailMacros{'auth_authen'})) {
        return;
    }
    # Sender is not authenticated
    # Run SpamAssassin, etc...
}
```

- Use **mimedefang**'s **-a** option to pass additional macros.

Performance

- A poorly-tuned MIMEDefang setup will be *very slow*.
- A well-tuned setup will be slower than plain-vanilla Sendmail, but still process a respectable volume of mail.
- I'll present tips for tuning your MIMEDefang setup.

Use Embedded Perl

- When the multiplexor starts a new slave, it does this by default:
  - set up pipes for the slave
  - **`fork`**
  - **`execve`** perl mimedefang.pl
- The overhead of creating a slave is quite high. In addition, slaves cannot share any data memory.

Use Embedded Perl

- With the multiplexor's -E option, we embed a Perl interpreter right into the multiplexor. On startup, it reads and initializes the filter. Then when it needs a new slave:
  - set up pipes
  - **`fork`**
- There is no **`execve`** overhead.
- Perl initialization done *once* instead of once per slave-activation.
- Perl slaves can share some data memory.

Embedded Perl Caveats

- Data-page sharing is not that helpful because of Perl's reference-counting implementation. Accessing even “read-only” data changes the memory and results in page copies. :-(
- Although I followed the perlembed man page religiously, the code fails on some platforms. MIMEDefang's configure script will detect whether or not it's safe to use embedded Perl. Help from Perl gurus???
- On platforms that don't support embedded Perl, the -E option is ignored.

Use Good Hardware

- Obvious first step: Use fast hardware with lots of memory.
- In our lab, we benchmarked a dual-Xeon at 2.6GHz with 1GB of RAM running Linux. It processed about 13 messages/second using SpamAssassin and Clam AntiVirus. This is about 1.1 million messages/day.
- However, this was under ideal conditions: Sender was on the same fast LAN as receiver. In real world, expect 35-60% of this performance.

Disks

- MIMEDefang spools messages into **`/var/spool/MIMEDefang`**. This data does *not* need to survive a crash; ***use a RAM disk***, especially on Solaris.
- Do *not* be tempted to use a RAM disk for **`/var/spool/mqueue`**! Your machine will be fast, but dangerous.
- Use fast disks. Consider a separate disk for the Sendmail queue. Consider a separate disk for **`/var/log`** (or log to a remote loghost.)

Miscellaneous

- On systems that support it, have **`syslog`** log asynchronously. This cuts disk I/O tremendously.
- If you can avoid calling SpamAssassin, avoid it! Write your filter to call SpamAssassin only as a last resort if message hasn't been rejected for some other reason.
- Avoid network tests (RBL, DCC, etc.) if possible. Network latencies can cause processes to pile up and kill the machine. Consider local mirrors if you must use network tests.

Storing State between Callbacks

- Successive callbacks for a given message can occur in different Perl slaves.
- However, in all cases, the *current working directory* is the same for a given message. You can store state in files in the current directory.
- If your MIMEDefang spool is on a RAM disk, this is relatively cheap.

Invoking MIMEDefang

- Consult the man pages for all the gory details. We'll cover some command-line options here.
- To invoke MIMEDefang, you first start the **`mimedefang`** program, and then **`mimedefang-multiplexor`**.
- Each program has its own command-line options.

Some `mimedefang` Options

- **`-U`** ***`user`*** – Run as *user*, not root. Mandatory.
- **`-p`** ***`connection`*** – Specify milter socket.
- **`-m`** ***`socket`*** – Specify multiplexor socket.
- **`-P`** ***`pidfile`*** – Write PID to *pidfile* after becoming daemon.
- **`-C`** – Conserve file descriptors by not holding descriptors open across Milter callbacks. On busy Solaris servers, this might be required. Also, on busy Solaris servers, use **`ulimit -n`** to increase number of file descriptors per process to 1024 from 256. (The Milter library uses **`select()`** internally; it won't handle more than 1024 descriptors on 32-bit Solaris systems.)
- **`-T`** – Log the filter times.

Some multiplexor Options

- **-U** ***user*** – Run as *user*, not root. Mandatory.
- **-s** ***socket*** – Specify multiplexor socket.
- **-a** ***socket*** – specify an “unprivileged” socket. The mutliplexor will only accept safe requests on this socket (for example, **load**, **status**, etc.) and not unsafe ones (**reread**, **scan**, etc.) This lets unprivileged users monitor performance.
- **-N** ***socket*** – listen for Sendmail SOCKETMAP requests.
- **-O** ***socket*** – socket for notification messages.
- **-p** ***pidfile*** – Write PID to *pidfile* after becoming daemon.
- **-X** ***secs*** – run a 'tick' request every *secs* seconds.

A socket may be specified using one of three forms:

/path/to/socket	A UNIX-domain socket.
inet:*portnum*	A TCP socket bound to ***portnum*** on 127.0.0.1.
inet_any:*portnum*	A TCP socket bound to ***portnum*** on the wildcard address.

For the **-s** option, only the UNIX-domain type is allowed.

Multiplexor Tuning Options

- **`-m`** ***`minSlaves`*** – Try to keep *minSlaves* Perl slaves running at all times.
- **`-x`** ***`maxSlaves`*** – Never allow more than *maxSlaves* to be running at the same time.
- **`-r`** ***`maxRequests`*** – Kill a slave after it has processed *maxRequests* requests.
- **`-i`** ***`idleTime`*** – Kill off excess slaves that have been idle for *idleTime* seconds.
- **`-b`** ***`busyTime`*** – Kill off a slave (and tempfail the mail) if it takes longer than *busyTime* seconds to process a request.
- **`-w`** ***`waitTime`*** – Wait *waitTime* seconds between starting slaves unless mail volume asks for slaves faster.
- **`-W`** ***`waitTime`*** – Wait *waitTime* seconds between starting slaves no matter what.
- **`-E`** – Use embedded Perl interpreter. Always use this option!

Multiplexor Tuning Options

- **`-R`** ***`maxRSS`*** – Limit resident-set size of slave filter processes to *maxRSS* kB. (Not supported on all operating systems.)
- **`-M`** ***`maxMem`*** – Limit total memory space of slave filter processes to *maxMem* kB. Suppported on Linux and most modern UNIX systems.
- **`-q`** ***`queueSize`*** – If all Perl slaves are busy, allow up to *queueSize* requests to queue up until a slave becomes free. By default, no queueing is done (*queueSize* is zero.)
- **`-Q`** ***`queueTimeout`*** – If a request is queued for more than *queueTimeout* seconds before a slave becomes free, fail the request.
- Normally, if all slaves are busy, a request fails. Queuing requests allows those SMTP transactions that are underway to have a better chance at completing. New SMTP transactions are tempfailed because **`mimedefang`** always issues a **`free`** request at connection time.

Multiplexor Logging Options

- **`-l`** – Log anything Perl slaves put on STDERR to syslog. Highly recommended.
- **`-t`** ***`filename`*** – Log statistical information to *filename*.
- **`-T`** – Log statistical information using syslog(3).
- **`-Y`** ***`label`*** – Set syslog label to *label*.
- **`-S`** ***`facility`*** – Set syslog facility to *facility*.
- **`-L`** ***`interval`*** – Log slave status every *interval* seconds.

Policy Suggestions

- Don't notify senders if you find a virus. Most viruses fake sender addresses. We suggest discarding viruses.
- Don't add silly disclaimers to outgoing mail. Their legal status is doubtful, but their annoyance factor is undisputed.
- Train users not to send HTML mail (or even enforce it.) Spammers use many silly HTML tricks to evade content-filters; non-HTML mail is much more likely to go through filters unmolested.

Policy Suggestions (2)

- Don't attempt tarpitting with MIMEDefang. Special ratware can run thousands of concurrent SMTP sessions, and you'll hurt your own server more than you hurt spammers.
- Be wary of DNS RBL's. They can be effective, but can cause tremendous collateral damage.

Conclusions

- MIMEDefang gives you the power of Perl to filter, slice and dice your e-mail.
- Use the power wisely!
- Participate in the MIMEDefang community: http://www.mimedefang.org
- Visit Roaring Penguin Software at Booth #21.
- And have fun.