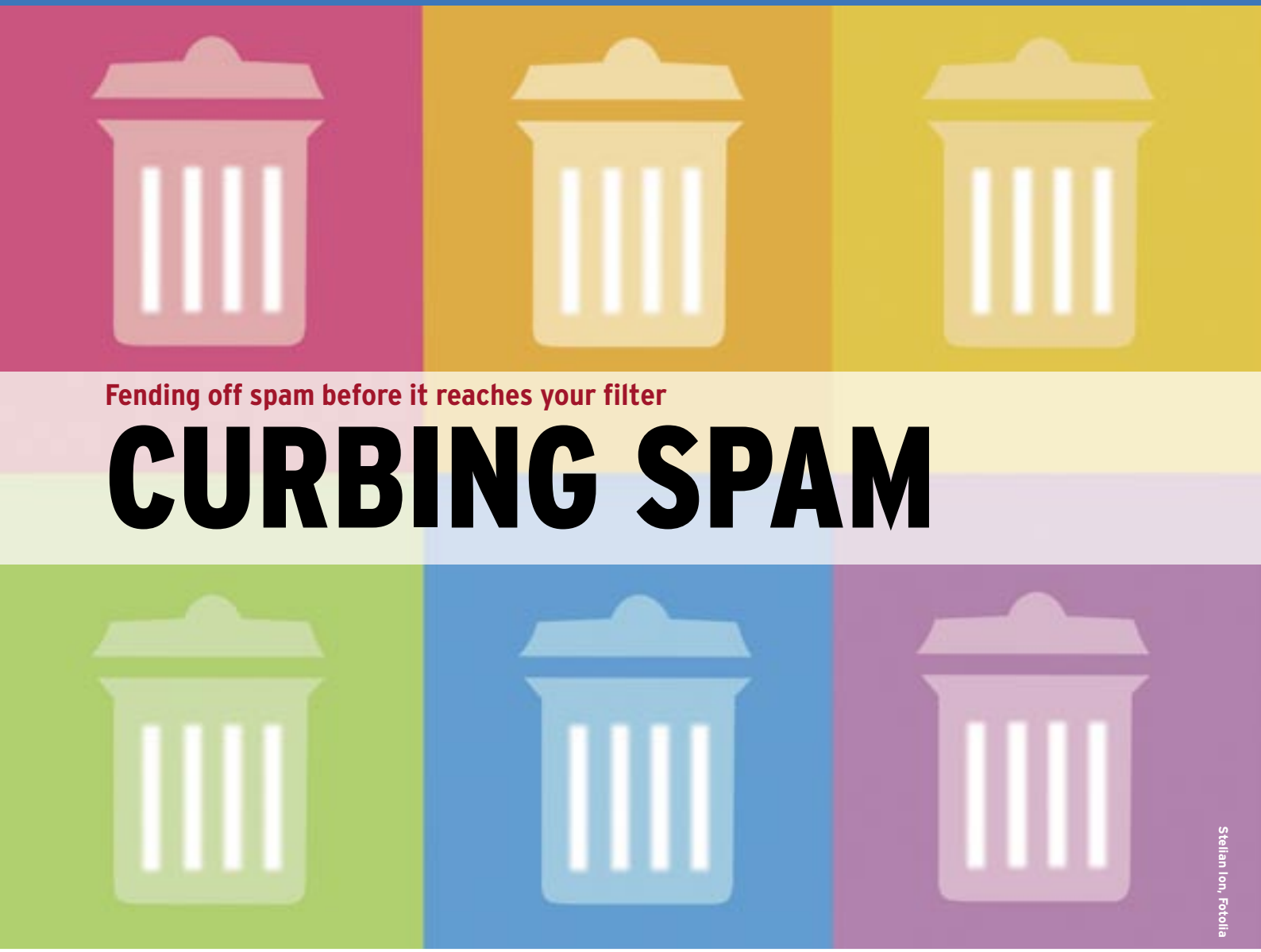

Fending off spam before it reaches your filter

CURBING SPAM

Stellan Ion, Fotolia

Sometimes the best way to keep spam out of your mailbox is to keep the spammers from getting your address.

BY TOBIAS EGGENDORFER

If you must publish your email address online, spammers will quickly flood your inbox with junk (Figure 1). Filtering the incoming mess requires CPU power, and you risk losing legitimate messages. Most of the work on spam protection has concentrated on recognizing and filtering spam messages as they arrive. Another approach, however, is to convince spammers to leave your address alone in the first place.

Two techniques that make spammers ignore your address are:

- making the spammers think the address is invalid;
- disguising the address to avoid it being discovered by harvesters.

This article examines both techniques. We'll start with a study of the Sponts box, a hardware-based tool that sends a *user unknown* message back to the spammer, and we'll end with an automated technique for altering the appearance of an email address so a harvester bot won't recognize it.

Sponts Effect

The manufacturers of the Sponts Box, which we investigated during this spam slam, refer to a phenomenon called the *Sponts effect*. The theory is that if spammers fail to deliver an email message, and receive a user unknown message instead, they will remove the address from their list to keep the list clean.

Spammers use programs known as address verifiers to validate addresses on their lists. The verifier takes an email list, reads the domain part, checks whether the domain has an MX (Mail Exchange) DNS entry, and then connects to the machine. Then spammers typically go through the SMTP dialog up to *RCPT TO:* or use the SMTP *VRFY* command to check the address.

Both of these commands query the target MTA (Mail Transfer Agent) to see whether it knows the address and is responsible for it. If so, the MTA will return an *OK* in the SMTP dialog, and if not, it will return an error message. The verifier evaluates these results and tags entries that produce an error message as invalid entries.

Table 1: Message Response and Messages Received

	Accepted	User unknown
Start	29958	29820
1 Month later	33424	36582
2 Months later	34839	36768
3 Months later	36738	35287

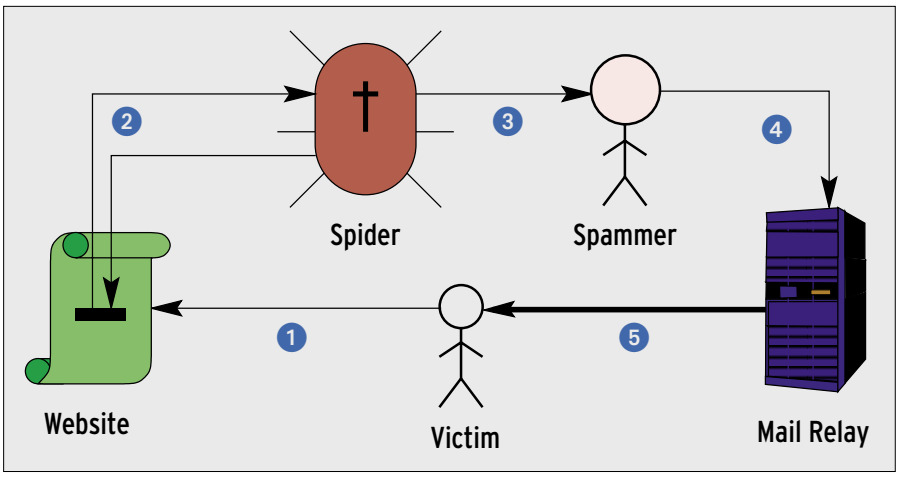

Figure 1: The vicious cycle of spam starts with the victim publishing an email address on a website. Spiders harvest the address and send it to the spammers, who in turn send electronic junk via mail relays to the victims.

This is where the Sponts effect could be useful, but the box does something more complex. It first tries to detect spam during the SMTP *HELO* command and then by investigating the *MAIL FROM:* line. These commands occur prior to *RCPT TO:* in the SMTP dialog (Figure 2). If the software is not sure whether the sender is a spammer, it accepts the message, like graylisting, and analyzes the message by running rules against it, returning a temporary SMTP error in case of spam. When the spammer tries to connect, the Sponts Box recognizes the IP address, the *EHLO* domain name, the *MAIL FROM:*, and the *RCPT TO:* properties, and returns a *User unknown* message. Unfortunately, verifiers only connect once. As they view the first attempt as a positive reaction, they will tag the address as good. Thus the Sponts block will only be effective if the verifier has already been blacklisted.

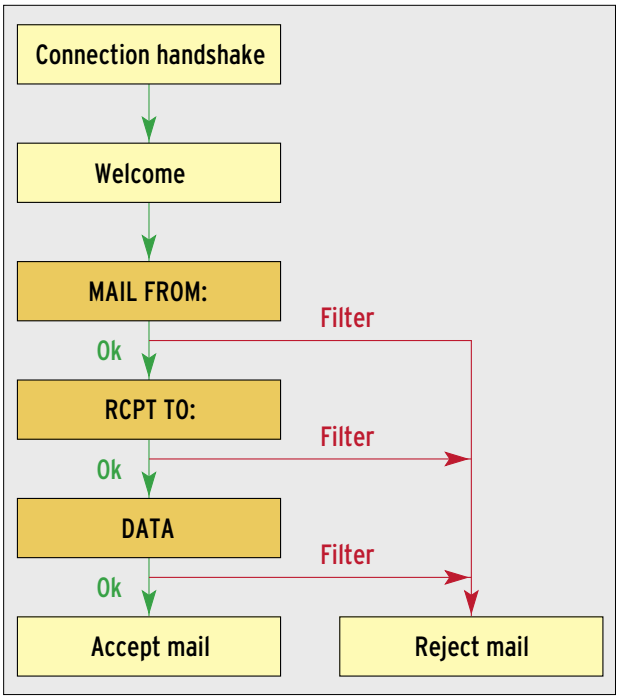

Figure 2: The Sponts Box checks various parts of the SMTP handshake to see if the message is spam or ham. If the box detects junk mail, it returns a user unknown message.

Communication Meltdown

These considerations were suspicious, so we took two heavily spammed domains, both of which received 30,000 spam messages a month, and programmed a SMtarPit-based [1], miniature mail server in Perl, which appeared to accept every incoming message for one domain. However, the mail server created a tarpit and rejected every message for the other domain, returning a *user unknown* and logging the results.

We let the mail server accept messages for the two test domains (Figure 3) for three months. There was no increase in the average number of messages per day, and no relative changes between the two domains (Table 1). The domain that accepted every message received more mail at the start and end of the test phase, but the domain that rejected everything overtook the other domain for the rest of the test.

The only explanation is the arbitrary nature of spam. We did not see any noticeable changes; both mail servers had the same amount of work to do, and there was no sign of a Sponts effect.

Obfuscation

We still had hope that we could combat the spam by obfuscating the addresses. We published a couple of addresses on web pages for 11 months, and we removed them in October 2005. To avoid confusing spammers, we gave them a new address to spam. Figure 4 shows the volume of junk mail that reached a control address that was online the whole time, the address we removed, and the address we added in October.

There are two obvious effects. First, the volume of spam increases rapidly for the new address. After about four months, the address receives as much spam as the addresses in the control group, and overtakes them toward the end of the test. Second, the spam curve for the hidden address is inversely proportional to this. After about four months, the spam volume dropped to half that of the control group. You would expect this to drop again in couple of months when the spammers get address CDs that do not have this address.

Removing an address from a web page is the same as using effective obfuscation. Some users insert nospam in the local or domain part of an email address. To test this, we published the email addresses *max.sample.nospam@example.com* and *betty.sample@nospam.example.com*, but suspiciously, we configured the mail server to accept and count mails sent to *max.sample@example.com* and *betty.sample@example.com*. Most spammers got Betty's address by removing the nospam subdomain, but they did not do so on Max's in most cases. However, as some spammers were able to spam Max, this obfuscation method is not as safe as it could be.

Out of Sight

Our tests have shown that simply adding blanks to an email address can be an ef-

fective obfuscation method, and it is theoretically secure. It is difficult to hide the email addresses you need to publish on your website. It would be more practical to replace the contact data by an obfuscated display when serving up the data. The idea is that the address is entered and stored naturally, and when it is displayed, a filter pumps the address up with spaces, so that:

```
user@example.com
```

becomes:

```
u s e r @ e x a m p l e . c o m
```

The address will be recognizable to a site visitor, but not to a harverster bot. For a large website, or for a web provider supporting user-built web pages, this technique for uniformly obfuscating all addresses reduces overhead and eliminates the uncertainty associated with obfuscating each address individually.

From the user's point of view, it would be ideal to have a one-click solution to hide addresses on virtual hosts run by major providers; on your own servers, you would just need a single line per host in *httpd.conf*. The following sections describe a solution that implements this form of address obfuscation through an Apache output filter.

Apache Filter

The Apache *httpd* has hooks that give admins the ability to deploy filters more or less anywhere in the website-handling process – from input filters in the input data stream, to output filters that directly influence the output data stream. The approach is simple: you choose a suitable program language, accept the data passed by the Apache API, use a regular expression to search for email addresses, replace the addresses with obfuscated ones, and hand the modified website back to Apache.

Apache filters are typically programmed in C and C++, and APIs are available for both. Perl has a convenient alternative in the form of mod_perl. As mod_perl compiles the script when you launch Apache, there is no danger of performance hits at runtime. Unfortunately, mod_perl is a memory hog, and Apache loads it into memory for each instance you run. A C module would consume far less memory, but better documentation, and the neat string handling functions were what made up our minds to opt for mod_perl for this project.

Obfuscation

Everybody has preferences when selecting a suitable obfuscation technique. If you prefer a different technique, no problem. The filter we will be investigating uses a separate obfuscation function, *obfuscate()* (Listing 1, line 17).

The next issue occurs during website parsing. Apache does not give the filter the whole file, but splits the file into segments the handler processes sequentially (lines 39 through 85). This makes sense because an output filter might be applied to DVD images, besides simple *text/html* or *text/plain* files. If Apache passed in the whole image, the system might run out of memory and crash.

This said, splitting files up into blocks has a few major disadvantages for obfuscation. If an email address straddles a block boundary, the filter would probably overlook at least part of the address. To prevent this, the filter will need to store some information between block boundaries to identify addresses reliably. From a technical point of view, it is easy to pass data over block boundaries, however, it is more difficult to decide which data to pass. If the filter stores too much information, it will waste memory space, and if it doesn't store enough, it may not work as intended.

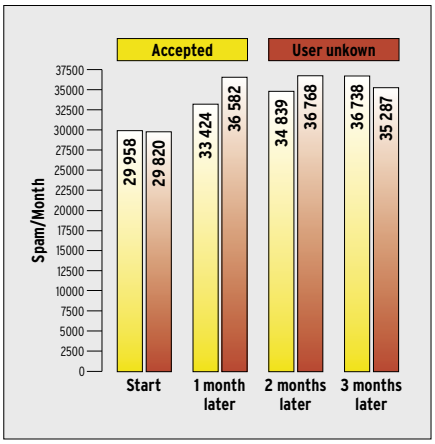

Figure 3: The incoming volume of spam remained constant over a period of three months, regardless of whether the mail server accepted every single message or rejected each message with a User unknown.

Standard Support

RFC 822 and the current 2822 variant [2] specify the makeup of an email address, which is not permitted to contain blanks under any circumstances. Fortunately, the blank is the most frequently used character in normal texts in any European language. Character distribution will probably follow the same patterns in

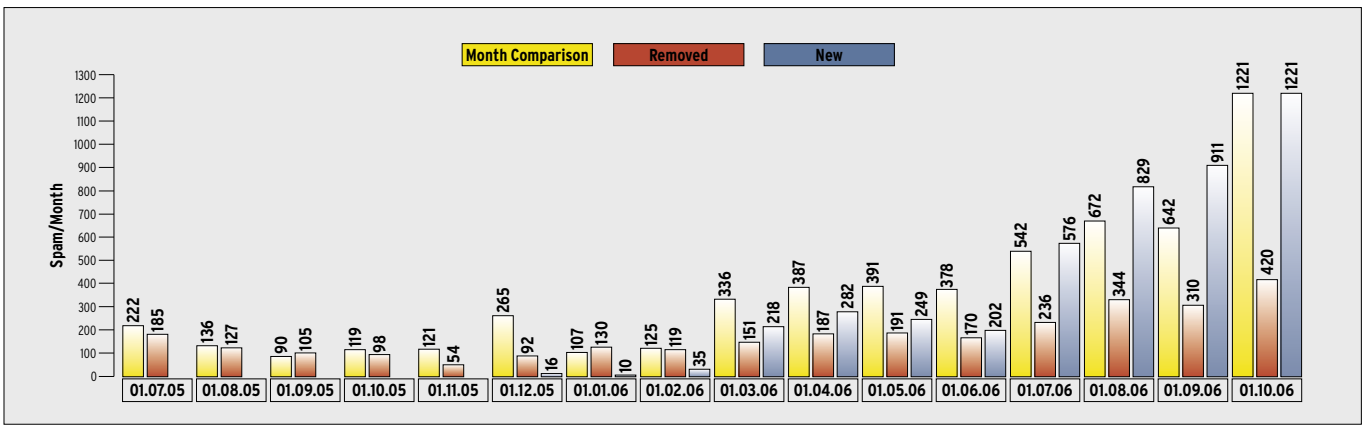

Figure 4: The longer you publish an address on a website, the more spam the address receives (yellow). The address with the red bars was taken off the web in October 2005 and replaced by the gray address. The volume of spam does not increase noticeably when the entry is removed from the web.

(X)HTML files, if we use any whitespace variant (blank, tabulator, newline) as separators.

This gives us an approach to block boundary handling. The filter will process each block up to the last whitespace, and append the next block passed to it to the results.

The only block the filter processes is the last one, so we can be sure that the filter will not split email addresses. The regular expression in line 68 searches for the last word boundary and splits the buffer. Anything occurring after the last word is passed by the handler to the next call in *$leftover*.

Lack of Memory

This approach might not work if the script stumbles over data without whitespaces. If the content is provided by a user (web form entries, for example), an attacker could enter data without whitespaces, causing the filter's memory consumption to skyrocket. If the script fails to find a blank within two buffers, it will output the whole buffer as a workaround. Although you may think the output could include a clear text mail address, the 10 KB buffer size would make this almost impossible.

RFC 2821 [3] restricts SMTP command lines to a maximum of 1000 characters, including CR + LF (end of line). *MAIL FROM:* and the greater than and less than characters surrounding the mail ad-

Listing 1: Output Filter

```
01 # Perl Output Filter for Apache 2.0:
02 # Automatically conceals email addresses to
03 # prevent harvesters from fetching them.
04
05 package MyApache::ObMail;
06
07 use strict;
08 use warnings;
09 use Apache::Filter ();
10 use Apache::RequestRec ();
11 use APR::Table ();
12 use Apache::Const -compile => qw(OK DECLINED);
13
14 use constant BUFF_LEN => 10240;
15   # Store apache output data
16
17 sub obfuscate {
18   # Invocation: obfuscate(data)
19   # Conceal all mail addresses
20
21   my $line = shift;
22   my $mail_regexp = '[A-Za-z_0-9.-]+@'.
23                     '([A-Za-z_0-9-]+\'.
24 '.)+[A-Za-z]{2,6}';
25   my $adr = undef;
26
27   while ($line =~ /($mail_regexp)/g) {
28     # Split address into single characters
29     # and reassemble with spaces in between
30     $mail = $1;
31     $obfus = join(' ',split(//,$mail));
32
33     # Replace all occurences
34     $line =~ s/$mail/$obfus/gi;
35   }
36   return $line;
37 }
38
39 sub handler {
40   # Called by Apache. Works through the blocks
41   # of data delivered by the httpd.
42
43   my $f = shift;
44   unless ($f->ctx) {
45     # Test content-type on first invocation
46     unless ($f->r->content_type =~
47                 m!text/(html|plain)!i ) {
48       # Only modify text/html and text/plain
49       return Apache::DECLINED;
50     }
51     # Reset Content-Length calculated by the
52     # server. We'll change the amount of data
53     $f->r->headers_out->unset('Content-Length');
54   }
55
56   my $leftover = $f->ctx;
57   while ($f->read(my $buffer, BUFF_LEN)) {
58     $buffer = $leftover.$buffer
59               if defined $leftover;
60     if (length($buffer) > (2*BUFF_LEN)) {
61       # Don't wait forever for whitespace
62       $f->print(obfuscate($buffer));
63       $buffer = $leftover = "";
64     } else {
65       # Keep the last beginning of a word
66       # in leftover to work only on full
67       # addresses and not on fragments.
68       $buffer =~ /(.*)(\s\S*)\z/gs;
69       $leftover = $2;
70       $f->print(obfuscate($1));
71     }
72   }
73
74   if ($f->seen_eos) {
75     # End of data-stream in sight.
76     if (defined $leftover) {
77       $leftover=obfuscate($leftover);
78       $f->print(scalar $leftover);
79     }
80   } else {
81     # Pass remaining data to next invocation
82     $f->ctx($leftover) if defined $leftover;
83   }
84   return Apache::OK;
85 }
86 1;
```

dress reduce this to a maximum of 985 characters for the address. The RFC also restricts the local part to 64 characters, and the domain name to 255. The @ character separates the two. If the buffer is longer than 64 + 1 + 255 = 320 characters, and does not include any blanks, it can't contain a valid mail address.

Mail addresses can be longer, under certain circumstances. The RFC says "Objects larger than these sizes should be avoided when possible," but it quotes text with converted X.400 addresses as possible source of oversized addresses.

The RFC recommends not to implement length restrictions. If it were not for this recommendation, the filter could use a value of 320 bytes for *$leftover*. You may think about implementing this alternative, or you set a limit of 985 characters.

Tuning

The buffer size is decisive for filter performance. We don't want it to be too small or too big. The best thing is to discover the optimum value by running the Apache 2.0 *ab* benchmark. The benchmark accesses a website a set number of times with a specific number of parallel connects to discover the average access time for a website.

The more parallel access you have to allow for, the smaller the buffer should be, as Apache will create a new buffer for each instance of the filter. If you have a large buffer, the system will start swapping sooner than you would want it to. A smaller buffer means more calls to the filter. Fine tuning the buffer size would reduce the inevitable delay with smaller HTML files to just a few percent, although having to restart Apache when you change the source code is a pain.

It makes sense to have the filter check to see if it really is receiving text data each time it is called; this prevents the filter from accidentally mangling non-text files. To do this, line 47 checks a regular expression of the content type detected by Apache.

The filter is triggered for *text/plain*, or *text/html*, otherwise it will just tell Apache that it is not interested in the data (line 49). Depending on your requirements, you could add more MIME types [4] here, to cover web servers that serve up WML pages to WAP-capable mobile phones, for example.

The Next Step

If the handler decides it is responsible, the first thing it does is reset the *Content Length* set in the HTTP header (line 53). Mail address obfuscation changes the length of the transferred document. The *obfuscate()* function referred to previously adds camouflage in lines 17 through 37.

The function expects a block of the right size, and uses the regular expression stored in *$mail_regexp* (lines 22 through 24) to extract mail addresses (line 27).

The regular expression is stored in a variable because the script uses it multiple times; it represents a practical implementation of the set of all valid email addresses as described in the context-free grammar used by RFC 2822.

The *split()* function chops up a string into an array of characters; *join()* adds blanks and glues the string back together (line 31). Thus, *a@bc.de* becomes *a @ b c . d e*. If you prefer some other obfuscation technique, this is where you need to make your changes.

To install and activate the Perl module, you need to store it in mod_perl's module search path, which are simply the normal Perl module paths. You could call the directory *MyApache* for your own Apache modules, and the module file could be *ObMail.pm* (for Obfuscate Mail). The directory and file names must be identical to the names in the *package* line of the source code (line 5), and they are case sensitive. The changes in the Apache configuration file, *httpd.conf*, are shown in Listing 2. After restarting the web server, the module should modify any web pages that it serves.

Conclusions

We did not notice a Sponts effect in our lab, and the obfuscation technique preferred by Usenet and forum users, adding *nospam*, has its limitations. One thing is for sure, however – it does make sense to protect your own email address, and if you have to reveal your address on a web page, make sure that you obfuscate it. ■

Authenticated SMTP

One of the reasons for today's spam epidemic is that SMTP does not use authentication. Although there have been many suggestions on improving SMTP security, all of them fail because they restrict or disable legacy SMTP functions, including such trivial things as mail forwarding, a function that is just as indispensable as call forwarding on phone systems. At the same time, the suggestions use competing standards, some of which are patent protected.

Apart from the business and political wrangling over standards, markets, power, and influence, as well as the ineffectiveness of the current proposals, there is another reason why these approaches are unlikely to succeed. Ten years ago, open relays were banned and blacklisted. A quick visit to *ordb.org* reveals that there are still 250,000 open relays today. Changing SMTP would affect at least 25 million mail servers worldwide, and far more clients, all of which would need updating.

In addition to this, the approaches are fairly ineffective. Eighty percent of all domains that have an SPF (Sender Policy Framework) entry in their DNS records belong to spammers. Thus, it would make sense to use the SPF entry as an additional criteria for detecting spam, in contrast to what the SPF entry was originally intended to do. Microsoft's call to use SPF was designed to exclude spammers and make SMTP more secure.

Listing 2: Loading the Module

```
01 SetHandler modperl
02 PerlModule MyApache::ObMail
03
04 <Directory /var/www/html>
05   PerlOutputFilterHandler
   MyApache::ObMail
06 </Directory>
```

INFO

[1] Paul Grosse, SMtarPit 0.6.0: *http://www.fresh.files2.serveftp.net/smtarpit/*

[2] RFC 2822, "Internet Message Format": *http://www.ietf.org/rfc/rfc2822.txt*

[3] RFC 2821, "Simple Mail Transfer Protocol": *http://www.ietf.org/rfc/rfc2821.txt*

[4] MIME-Type-Listen: *http://www.iana.org/asignments/media-types/*