PREVENTING SPAM BY DYNAMICALLY OBFUSCATING EMAIL-ADDRESSES

Tobias Eggendorfer
Fernuniversität in Hagen
LG Parallelität und VLSI
Informatikzentrum
D-58097 Hagen
Germany
tobias.eggendorfer@fernuni-hagen.de

Jörg Keller
Fernuniversität in Hagen
LG Parallelität und VLSI
Informatikzentrum
D-58097 Hagen
Germany
joerg.keller@fernuni-hagen.de

ABSTRACT

Filtering unsolicited commercial email has proved its untrustworthiness by not keeping spam out of the inbox and marking solicited mails as unsolicited. As filters only cure a symptom of the spam-epidemic, it seems more promising to resolve the problem at its roots. According to different studies, spammers rely on email addresses published in a machine readable format on the world wide web.

We have tested different approaches to obfuscate email addresses in the www and present experimental results that indicate their usefulness. However, modifying existing webpages only to obfuscate mail addresses on them means a lot of work and forces web designers to understand the techniques and concepts used to conceal addresses.

Instead we provide an output filter for the common Apache webserver, which allows to leave even dynamically generated web pages unchanged.

KEYWORDS:

- Unsolicited Commercial Email
- Prevention
- Dynamic Obfuscation
- Webserver

SPAM FILTERS

The large amounts of unsolicited commercial email (UCE), also known as spam, can have mail servers collapse [1]. It keeps mailboxes overflowing, blocking thus important mails by an over-quota-condition. Even if a ham-mail, the opposite of spam, finds its way into user's mail client, chances are, it is moved to a spam folder or is failed to be noticed among hundred of spam mails. Doing this, UCE turns email into an unusable, unreliable means of communication.

Most users and providers have implemented filtering technologies [2] to reduce the amount of spam received. Although there are many concepts in filtering spam, none of them is reliable: Content-filtering tries to determine the spam-status by looking for "bad" words. It might work on some recipients, but not on all: A bank official cannot put "mortgage", a word often found in spam, on a blacklist. Relay-Black-Lists (RBL) will not work either, as more and more people provide their own mail server connected to the internet by a DSL-line using a dynamic IP address. In Germany this often happens to circumvent German legislation (Telekommunikationsüberwachungsverordnung, TKÜV) that forces providers with more than 1000 customers to give law enforcement agencies access to email communication at mail server level at any time. Most dynamic IP-blocks are listed in RBLs to prevent internet worms from spreading as they usually originate from infected home computers. Any other filtering solution has its own, specific limitations. So most filters, including those combining several technologies and building an overall spam-score, have spam recognition rates of less than 99%. Even worse, their false positives rates, i.e. ham considered to be spam, are up to some percent [3], [4].

Considering this, spam filtering is only a symptomatic cure with heavy side-effects. To really get rid of spam, the root of the evil needs to be eradicated.

HOW SPAMMERS COLLECT MAIL ADDRESSES

According to different studies [5], [6], spammers still use the world wide web to find mail addresses to spam to. To achieve this, they use so called harvesters; programmes working generally the same as friendly spiders used by search engines: They follow each link shown on a web page they visit and extract email addresses from each page found.

In [6] a two-line unix shell code harvester is shown to demonstrate how primitive those programmes are. More elaborated harvesters are found on the world wide web [6]. Most of them run on windows systems and have graphical user interfaces. Some were even advertised in T-Online's download area[1], one of Germany's biggest ISPs.

1 See: http://groups.google.de/groups?selm=42343BF3.1040005%40schloachdi.de

If spammers' only source for mail addresses is harvesting them on webpages, the obvious solution is to conceal mail addresses on the web. The above studies confirm this hypothesis, so the solution to conceal email addresses on web pages should work.

METHODS TO HIDE MAIL ADDRESSES

In [6] different methods to conceal addresses were shown. Some of them rely on JavaScript-enabled Browsers, a feature harvesters do not implement as of now. Anyway, using Mozilla's publicly available JavaScript-engine, anyone could build a new harvester able to execute JavaScript and thus revealing carefully hidden addresses. Therefore [6] introduces a human-machine-detection [7] in JavaScript by asking someone to enter a key shown on the page. A JavaScript enabled harvester will fail because it does not understand the semantics. Spammers solve this by creating online games or sweepstakes and ask humans to solve the puzzle. Their solution is then used by the harvester to crack the code. JavaScript has another disadvantage: Most people disable it in their browser for security reasons. Those are then kept from communicating with the site's owner because they will not see his email address on the web page.

Another approach is to display an image containing the address. Most harvesters ignore pictures and even if they would not, they needed an OCR-programme to find the mail address. By slicing the image into parts, OCR-programmes will also fail. But the picture is also unreadable to visually handicapped page visitors relying on a braille-line to read.

As a consequence, to accomplish the typical requirement “barrier free”, the email address needs to be concealed by using only means of standard HTML. The solution most frequently provided is to use HTML-entities: Each character is replaced by an ampersand, followed by its ASCII-Code and a semicolon. An “A” is represented by “&65;”. This encoding is compatible to any platform as long as email addresses are restricted to 7-bit-ASCII-standard. Currently this is to be expected, internationalized domain names (IDN) are converted in their puny-code representation for email and so brought back to 7 bit.

Although this solution seems very appealing, it only takes one line of code to exchange HTML-entities with their character representation. By using lynx, a command line text browser, with the --dump option, there is no coding required at all. At least one harvester is able to do the transformation already, as both [5] and [6] report on having received mails on addresses obfuscated that way.

Almost the same problem arises with the other common solution: The address is URL-encoded, i.e. each letter is replaced by a percent-sign and its hexadecimal ASCII-value. An “A” would become “%41”. Most scripting-languages provide a ready-to-use function to do the retransformation. But for some reasons, hiding an address by URL-encoding is still efficient. In neither study was a spam mail received that way.

A better solution is to obfuscate an email address up with spaces: user@example.com becomes the human readable u s e r @ e x a m p l e . c o m. Every visitor of a web page should be able to capture the mail address, but a harvester is confused: Removing whitespaces all over the page is not a solution, as the mail address would unite with the words left and right to the address. To increase the difficulty level, spaces could be inserted only after a random amount of letters. By doing this, computers should be definitely unable to recognise the address.

To make the displayed address look nicer on the page, the spaces could be hidden in a HTML-DIV-tag set to invisibility by CSS. A human visitor using a CSS-capable browser would not even see that hiding took place, current harvesters ignore CSS, so they will see the spaces in between. If evolving Harvesters understand CSS, the space width within those DIV-tags could be reduced by reducing the font size, which, in turn makes the space again virtually invisible for a human visitor, but again existing for the harvester.

TESTING THE METHODS

To confirm the results from [5] and test the proposals from the preceding section, a test page has been set up under a previously unused domain [6]. On it, in a DIV-tag set to be invisible by CSS, different email addresses using different methods of obfuscation were listed. To human visitors using a CSS-capable browser, those methods and addresses were invisible. But harvesters ignore CSS. To them, the addresses were readable.

Harvesters were sent there by hidden links on thousands of web pages, thanks to the support of many webmasters world-wide. Both, using hidden links and not displaying the addresses was done to exclude as many human visitors as possible. The idea was: the fewer people know about the page's existence and the fewer people see the addresses listed there, the smaller the chance anyone could give those addresses to a spammer, thereby reducing significance of the test's results.

The mailserver for the domain has been configured to count incoming mails and their time of arrival in a database. By doing so, mails did not see any spam filter and the risk of overlooking a spam mail has been eliminated that way, too.

The test setup and the results are described in detail in [6]. The most important results can be summarized as follows: Addresses shown in plain text received lots of spam, whether they were linked with an A HREF-tag or not. All but one obfuscated addresses were ignored by harvesters. Only the “mailto:”-linked address hidden using HTML-entities got a few spam mails. All other addresses, including a not linked HTML-entities hidden address, did not

receive even one spam-mail.

Table 1 gives an overview on the amount of spam received on each address between 2004-12-19 and 2005-08-25. It has been normalised to the spam-counter on a linked, plain text address as 100%, i.e. the amount of spam received by an email address published the usual way.

Obfuscation Method	Normalised Amount of Spam
HTML-Entities	0,35%
URL-Encoding	0,35%
HTML comments	0,35%
Linked HTML-Entites	36,68%
JavaScript plain text	130,45%
plain text	98,62%
linked plain text	100,00%

Table 1 Amount of spam received on addresses obfuscated different ways

Those results were verified using several harvesters freely available on the internet pointed to the test page. They did not find any address other than those spammed – although most of them are advertised with the promise to find even obfuscated addresses.

The harvesters we tested did not understand the encoding of the HTML-entity masked addresses, but listed them in the hidden format as found mail addresses. They also only found linked ones.

This is in concordance to the tests using the website: The amount of spam received on a not linked, but obfuscated address is a bare minimum.

As some harvesters seem to recognise mail-addresses in HTML pages only if they linked, we recommend to replace “mailto:”-Links with HTTP-links to contact-forms and to obfuscate unlinked email addresses in a way they do not match typical regular expressions.

HAVE THE SERVER OBFUSCATE THE ADDRESSES

Although the above methods are efficient and easy to implement on an new webpage, it means a lot of work to modify existing pages and, if more than one person is in charge of keeping a website up to date, there is a risk that someone forgets to hide an address.

To mask an address manually also requires to understand both: the methods and the goals. It is very likely that home users and small businesses will not have the necessary knowledge. So their webpages would still offer email addresses to spammers.

Therefore we strived for a solution providing website-providers with an easy configurable, one-click-to-the-customer solution enabling or disabling concealed mail addresses on any page delivered by a server to enable a major step in fighting spam. If the big players in the provider market would implement the described method to dynamically obfuscate email-addresses, most harvesters will start starving.

A programme doing the obfuscation should work on any delivered text file – be its MIME-Type text/plain or text/html. To really hide all addresses displayed on a webpage in plain text, the programme needs to work on dynamically generated pages as well, whether they have been made up by JSP, PHP, Perl, Shell or even a compiled programme on the server.

CONCEPT

To achieve this, the programme would need to catch the output stream of the webserver right before data is delivered over the network. The modern Apache 2.0 webserver offers the possibility to do so without the need to modify its source code and recompile the whole server [8]. It is sufficient to write a programme in Perl and call it via Apache's configfile with the following directives:

```
SetHandler modperl
PerlModule MyApache::ObMail

<Directory /var/www/html>
  PerlOutputFilterHandler MyApache::ObMail
</Directory>
```

Apache will then send any output through this filter. The Perl-Code for this filter is shown in the appendix.

The filter script needs to be installed in a subdirectory named “MyApache” in a file ObMail.pm in any Perl-Module directory.

HOW THE FILTER WORKS

The Filter uses an input buffer to store data received from the webserver. Doing so is necessary: The amount of bytes transmitted might exceed any provided buffer – no matter how big it is. By using this buffer however, data is divided into buffer-sized fragments.

An email address now might be distributed among two fragments, so a regular expression would not match it. To resolve this, the script first divides data found in the buffer into two parts: The last part contains any characters found behind the last white spaces in the buffer and is then prepended to the next buffer-fragment. This is sufficient,

because email addresses do not contain white spaces.

SECURITY CONSIDERATIONS

In theory, any text-file, be it HTML or plain text, will contain white spaces. From cryptanalysis it is known that white spaces are the most frequently used character in almost any language. Source code also contain lots of white spaces.

Binary data should however not be transmitted with any text-MIME-type. But in reality, the script works on untrusted input data. An attacker might put a HTML-file on the server containing more characters without white spaces in between than the buffer will store. Due to the while-loop prepending the leftover buffer from the previous loop-run to the current buffer, a buffer overflow condition might be abused to execute arbitrary code on the server [9], [10].

The attacker might also do this by abusing any web form, where data he entered is shown on the next dynamically generated page.

To avoid this danger, the filter will output the buffer without caring for word boundaries if the buffer size exceeds twice the given maximum buffer size (variable BUFF_LEN in the script). This solution works fine with Perl, as memory is allocated dynamically. If the filter is written in any other language, one should allocate three times BUFF_LEN of memory for the buffer.

PERFORMANCE

The choice of the buffer size is not only important for security reasons, but it is also the parameter determining the performance of the script. Tests with Apache Bench, a benchmark for web servers included in Apache 2.0 simulating heavy load and concurrent requests, showed, that if the buffer is too small, performance might go down severly. In tests, after reducing the buffer from 10 KBytes to 1 KByte, delivery of the pages was five times slower.

But a too big buffer is also a performance killer: Doubling BUFF_LEN to 20 KBytes brought also performance down again by a factor of ten, because of a lack of memory and the resulting need for swapping. This is due to the parallel HTTP-requests: Each instance of the script needs to allocate memory.

So the buffer should be carefully adjusted to its optimum depending on the available free memory on the system.

With a view to performance, at first glance Perl is not optimal: It needs an interpreter to run, which makes it usually slower than a compiled language like C. But by using mod_perl, this problem is also solved: mod_perl compiles this Perl-module at Apache's startup.

Overall, for small files, performance decrease compared to a system without the filter, is minimal: The delivery time for files approximately 250 Bytes increased only by a factor of 1.2 – under heavy load with 1000 concurrent requests . The bigger files get, the slower the performance: For 5 KBytes, it took already 1.7 times longer than without the filter. With a 2 MByte HTML-file, it took 30 times longer to download than without the filter. However, such a size is quite unusal for a HTML-file.

This is expected behaviour: The while-loop for the input fragments means a performance at least within O(n) (n = length of document) without taking into consideration the implementation of the regular expression parser in Perl.

SUMMARY

The Perl module introduced herein offers on the fly email obfuscation for web pages to prevent spammers' harvesters to find addresses. It has been designed with performance and security in mind and might easily be setup on any web server running Apache 2.0 and mod_perl; both are often available out-of-the box.

The concept of concealing mail addresses on web pages has been proven to be effective by tests on the web. [5] even claims a long term decrease of spam if addresses are removed from the web.

REFERENCES

[1] S. Frei, I. Silvestri. G. Ollmann, *Mail Non-Delivery Notice Attacks*, http://www.techzoom.net/paper-mailbomb.asp, 2004

[2] A. Schwartz, S. Garfinkel, *Stopping Spam*, (Sebastopol, O'Reilly, 2002)

[3] J. Bager, *AOLs Spamfilter übertreibt (in German: AOLS's Spam Filter overacts)*, http://www.heise.de/newsticker/data/jo-21.10.03-000/, 2003

[4] U. Mansmann, *Spamcop sperrt GMX (in German: Spamcop blocks GMX)*, http://www.heise.de/newsticker/data/uma-10.09.03-000/, 2003

[5] Center for Democracy and Technology, *Why am I getting all this spam?*, http://www.cdt.org/speech/spam/030319spamreport.pdf, 2003

[6] T. Eggendorfer, *Methoden der präventiven Spambekämpfung im Internet (in German: Methods of Preventive Spam Abatement)*, (Munich / Hagen, Master thesis, Fernuniversität in Hagen, 2005)

[7] L. von Ahn, M. Blum, N. Hopper, J. Langford, *The Captcha-Project. Telling Humans and Computer apart (Automatically)*, http://www.captcha.net, 2004

[8] L. D. Stein, D. MacEachern, *Writing Apache Module with Perl and C*, (Sebastopol, O'Reilly, 1999)

[9] G. Hoglund, G. McGraw, *Exploiting Software. How to break code*, (Boston, Addison Wesley, 2004)

[10] C. Peikari, A. Chuvakin, *Security Warrior. Know Your Enemy*, (Sebastopol, O'Reilly, 2004)

APPENDIX: PERL OUTPUT FILTER (SOURCE CODE)

```
# Perl-Output-Filter for Apache 2.0
#
# Automatically conceals email addresses to
# prevent harvesters from fetching them.
#

package MyApache::ObMail;

use strict;
use warnings;

use Apache::Filter ();
use Apache::RequestRec ();
use APR::Table ();

use Apache::Const -compile =>
    qw(OK DECLINED);

use constant BUFF_LEN => 10240;
    # The buffer stores output data from
    # Apache. Its size is the key to
    # performance: If it is to small, it
    # slows the system down. If it is to
    # big swapping becomes necessary – and
    # again, the system slows down. An
    # optimal setting might be found using
    # ApacheBench (included in Apache 2.0).

sub obfuscate
  {
    # Invocation: obfuscate(data)
    # Here, mail addresses within in the
    # data block given as only parameter
    # are obfuscated

    my $line = shift;

    my $mail_regexp = '[A-Za-z_0-9.-]+@'.
                      '([A-Za-z_0-9-]+\'.
                      '.)+[A-Za-z]{2,6}';
    my $adr = undef;

    while ( $line =~ /($mail_regexp)/g )
      {
        # Split each address found in its
        # characters and put it together
        # again with spaces in between.
        # The address is given in $1 – so
        # another hiding method might be
        # implemented without trouble.

        $adr = join(' ',split(//,$1));

        # Now, replace any upcoming
        # occurences of this address.
        # If the hiding method has been
        # changed, take care: $1 might have
        # been reset by another regular
        # expression.

        $line =~ s/(.*)$1(.*)/$1$adr$2/gi;

      } # end while mail_regexp
    return $line;
  }

sub handler
  {
    # This function is called by Apache. It
    # takes care of working on the blocks
    # of data delivered by the httpd.

    my $f = shift;

    unless ($f->ctx)
      {
        # Test the content-type only on
        # first invocation
        unless ($f->r->content_type =~
                    m!text/(html|plain)!i )
          {
            # Works only on text/html or
            # text/plain
            return Apache::DECLINED;
          }
        # Reset Content-Length calculated
        # by the server. We'll obviously
        # change the amount of data sent.

        $f->r->headers_out->unset(
                        'Content-Length');
      }

    my $leftover = $f->ctx;

    while ($f->read(my $buffer, BUFF_LEN))
      {
        $buffer = $leftover . $buffer
                  if defined $leftover;
        if (length($buffer) > (2*BUFF_LEN))
          {
            # Did not found any whitespaces
            # for too long. To prevent a
            # buffer overflow, data is
            # sent. Note: Mail-addresses
            # half on this block and half
            # on the next might not be
            # fully obfuscated!

            $f->print(obfuscate($buffer));
            $buffer = $leftover = "";
          }
        else
          {
            # Keep the last beginning of a
            # word in leftover to work only
            # on full addresses and not on
            # fragments.

            $buffer =~ /(.*)(\s\S*)\z/gs;
            $leftover = $2;
            $f->print(obfuscate($1));
          } # end if bufferoverflow
      } # end while read


    if ($f->seen_eos)
      {
        # End of data-stream in sight.
        if (defined $leftover)
          {
            $leftover=obfuscate($leftover);
            $f->print(scalar $leftover);
          }
      }
    else
      {
```

```
        # There is more data to be
        # processed. Pass them to the next
        # invocation.

        $f->ctx($leftover) if defined
                                  $leftover;
      } # end if

    return Apache::OK;
}
1;
```