USENIX Association

Proceedings of the 17th Large Installation Systems Administration Conference

San Diego, CA, USA
October 26–31, 2003

Distributed Tarpitting: Impeding Spam Across Multiple Servers

Tim Hunter, Paul Terry, and Alan Judge – eircom.net

ABSTRACT

This paper describes an Irish ISP's attempts to combat the abuse of resources caused by unsolicited commercial email. We describe the extension of a multicast system, used to implement POP-before-SMTP relaying, to share information about remote mail servers between multiple mail systems. The information may then be used to tarpit abusive servers – placing delays between SMTP protocol answers thus mitigating their impact on our systems. We then examine how effective this has been, and come up with some ideas for future development.

We also discuss building a policy around this and other measures we use to combat spam. An ISP is in the business of sending and receiving mail – this makes slowing or blocking mail a delicate subject.

Introduction & Problem Statement

Unsolicited commercial email (spam) is a problem that, by now, needs no introduction. If you know about email, you know about spam. There are whole books on how to stop it, and it's even on the nightly news [1, 2]. The spam problem from an ISP perspective is also reasonably well known – spam is expensive in terms of time spent receiving it, space spent storing it, and staff months spent dealing with complaints and the technical aspects of cleaning up after it.

Our company, eircom.net, is the ISP division of eircom, the largest telecommunications provider in Ireland. It is also the largest internet provider in Ireland, serving approximately 500,000 customers. To put this in context, a recent survey estimated that around 766,000 adults in the south of Ireland use the Internet at home [3]. As such, any impact on our services is very visible – and is often reported in the media. Our problems with spam are probably fairly average – both the consistent low-level spam that slowly fills up our filesystems, and high-volume assaults[1] that have at least once slowed even our recently retooled mail system solution [4]. However the impact of it, in terms of customer impression and our reputation, is relatively severe.

There are any number of popular solutions for individual and group spam blocking available. The simple ones are a good start – things like: don't run an open relay; don't allow multiple recipients for null sender; and verify that envelope sender contains a valid domain. Yet the spammers seem to have worked around them. Using a blocking list involves handing a fair amount of responsibility and control to a third party – something that would not make for a good response to a customer unhappy with missed mail. Content analysis tools bring up privacy issues [5], cost a lot in processing power, and tend to require tuning for each individual recipient.

Having rejected those options, we looked around for others and came upon tarpitting. The idea, when applied to SMTP, is that when a sender has triggered the tarpit, the SMTP server places an intentional delay between processing of the sender's "RCPT TO <address>" command, and it's "250 OK" response [6]. The tarpit delay is initially triggered by a particular rate of sending mail. The delay can be increased if the sender continues on sending at the maximum rate allowed.

This seemed to us to be a good middle ground – it would dampen the blow of a dictionary attack[2] or other high-volume mailshot from a single source, preventing server overload. If the sender turns out to be a legitimate source, we have not entirely blocked the flow of information to (or from) our customers. In either case, while the server's performance would not reveal anything amiss, any high tarpit delay can be used to trigger an alert to our operations group – enabling them to examine the situation and decide to block the sender entirely, or remove the delay. The general customer base doesn't see a problem, a mistakenly-captured legitimate sender is not entirely blocked, and we remain in control of the situation.

An additional advantage of this approach was that it fitted easily into our existing infrastructure.

Our Environment

We currently run several mail servers in parallel, each providing POP and SMTP services using qmail 1.03 and vpopmail 5.0 for local deliveries [8, 9]. The latter allows us to serve two ISPs (eircom.net and indigo.ie) using the same hardware and software. We

[1] Which seem to frequently come on a Friday evening or over the weekend. Ours is not always a polite adversary.

[2] A dictionary attack is the attempt by a spammer to guess user names on a server by using a large combination of common words, common names, and numbers [7].

use a number of qmail patches from the general community, as well as local modifications to both qmail and vpopmail. An SQL database is used to store provisioning and configuration information.

Our current servers have dual one gigahertz Pentium III CPUs, a gigabyte of memory, and run FreeBSD. Together they handle 1200 messages per minute on average, and many times that at peak times. Traffic is distributed among these systems via a pair of server load balancers. User and domain mail is stored on shared NetApp [10] filers.

One of the services we provide is POP-before-SMTP. This allows a user who is downloading their mail from outside of our IP range to relay mail through our servers for a certain amount of time after a successful POP authentication. As a customer's SMTP connection may reach a different server than the POP connection, potentially only a split second later, this information must be rapidly shared between the servers. So the information about who has POP'd, and when, is passed between servers via multicast,[3] and stored in a shared memory table on each individual server. This table is queried by a small application that runs in the tcpserver[4] exec chain and sets an environment variable if the outside IP is allowed to relay. This is the basis for our distributed tarpit information, and is covered in greater detail in the following sections.

Existing Solutions & Other Work

In October 2002, when we began looking into tarpitting in earnest, we did some searching about for existing solutions – but generally assumed that we'd have to do most of the work ourselves, and that the information to be shared would fit fairly well into the existing POP-before-SMTP infrastructure. We really only came up with one close match – Chris Johnson's tarpit.patch [13]. It's a good starting point, but doesn't implement, or have hooks for, the cross-session and cross-server functionality we need.

In going back to look for prior art for this paper, we came across the 2002 LISA paper on "Spam Blocking with a Dynamically Updated Firewall Ruleset" [14]. While this system wouldn't have worked at our site without changes, it certainly matches well with our desire for a non-permanent/partial blocking mechanism that is easily controlled. The modularity – in passing data in and out of a central agent responsible for determining what is blocked – also matches well with our solution (and the general qmail way of doing things). Elements of their design may well be incorporated into future updates of our system.

[3] Multicast [11] is a network protocol that allows one host to send a packet to a selected set of hosts. It's a middle ground between sending to just one host (unicast), or all hosts on the same LAN (broadcast). An application/host "signs in" to a multicast session by connecting to a designated multicast IP address.

[4] tcpserver is part of D. J. Bernstein's ucspi-tcp package of TCP client-server application tools [12]. We use it as a replacement for inetd, to monitor a port and invoke a program or series of programs for each connection.

POP-before-SMTP Implementation: auth-record, auth-monitord, and auth-lookup

qmail provides POP and SMTP service via two different daemons: qmail-pop3d and qmail-smtpd. In order to implement POP-before-SMTP, we need each instance of the POP daemon to record the IP address and timestamp, of a successful non-local authentication, into a table shared across our mail servers. The timestamp indicates when the entry should expire. When a subsequent connection is made to the SMTP daemon, this daemon must check for the connection IP address in the shared table. If the IP address is found, and the entry has not expired, then the SMTP service will allow relaying.

Like a number of D. J. Bernstein's tools, qmail separates functions into mutually untrusting programs. These often run each other in a chain by having each program do its bit and then exec any remaining command line arguments. Use of root is minimized and programs sometimes run as separate users. This approach keeps programs small, promotes security and modularity and also, incidentally, makes it easy for us to create new programs and add new features without making extensive (or sometimes any) changes to the existing code.

We use this modular approach to split up the POP-before-SMTP tasks: sending out a multicast packet containing the IP address; writing the IP address and a timestamp into a shared memory table; and reading the shared table. The first task happens during the POP session, as a part of the exec chain, which looks like this:

tcpserver → vchkpw → auth-record → qmail-pop3d

- The tcpserver process listens on port 110 and forks a new exec chain for each incoming connection, setting $TCPREMOTEIP, the IP address of the remote system, among other environment variables.
- vchkpw reads the username and password from the network, and checks the entries against our user database. If the username and password are valid, vchkpw does a chdir() into the user's directory and execs auth-record. An invalid username and password, a timeout, or the QUIT command will result in the program exiting and the connection being closed.
- If the IP address is not on our local network, and the user is of a type allowed remote access, auth-record sends a multicast packet containing the IP address in $TCPREMOTEIP, and then execs qmail-pop3d.
- qmail-pop3d takes POP3 protocol transaction state commands (LIST, RETR, DELE, etc.) from the connection and returns the appropriate response.

The second task is performed by a daemon called auth-monitord. This program runs on each mail server

and listens to the multicast session that auth-record talks to. It reads the IP address contained in the multicast packet and writes it, along with a timestamp in the future, into a shared memory table. For example, if IP address 1.2.3.4 establishes an authenticated POP connection on server A at 14:45, auth-record on server A sends a multicast packet that is picked up by servers A, B, and C. Given a 15 minute timeout, the auth-monitord process on these servers will write an entry into their system's shared memory table that looks like this:

```
IP Address  SMTP relay allowed until
1.2.3.4     15:00
```

auth-monitord scans the table on a regular basis to clean out expired entries. The entry above will be removed during the first scan after 15:00. However, if another timestamp containing '1.2.3.4' arrives at 14:55, the timestamp will be updated to 15:10 and the entry will not be removed until after that time.

The use of multicast allows us to have the same information in a shared memory table on each mail server. It also removes any difficulty with race conditions – multiple simultaneous writes are placed into a (not necessarily ordered) queue by UDP/IP. auth-monitord needs no table locking code – it merely pulls entries one at a time from the queue.

Multicast is not a reliable protocol, so there is some chance that the tables will not be exactly the same across the mail servers. In general the chances of this are quite low. However, if a mail server is rebooted, or a new server is started, that server will begin with an empty shared memory table, which could be quite noticeable depending on the timeout period. To rectify this, the newly-started auth-monitord sends a multicast packet asking for a table dump from the other servers. The replies consist of multicast packets containing multiple entries with both IP addresses and their appropriate timeouts.

The final POP-before-SMTP task, looking for an entry in the shared memory table, is performed by auth-lookup. This program is invoked as part of the qmail-smtpd exec chain:

tcpserver → auth-lookup → qmail-smtpd

auth-lookup checks for the IP address, contained in environment variable $TCPREMOTEIP, in the shared memory table. If a table entry is present and has not expired, auth-lookup sets the environment variable $RELAYCLIENT with a null value. If this value is set, qmail-smtpd will allow the incoming connection to relay mail.

Figure 1 continues with our example data, and illustrates the flow of information between mail client and mail servers, and in between the mail servers themselves. In order to simplify the diagram, the only programs shown are auth-record, auth-monitord, and auth-lookup. The POP-before-SMTP process proceeds as follows:

1. At 14:45, the mail client on IP 1.2.3.4 establishes a POP3 connection with mail server A. The client gives a valid username and password to vchkpw, which then invokes auth-record.
2. auth-record sends a multicast packet that indicates that '1.2.3.4' has an authenticated POP3 connection. This packet is received by the auth-monitord process on mail servers A, B, and C.
3. On each server, the auth-monitord program writes an entry into the shared memory table for 1.2.3.4, with a relay expiration time of 15:00.
4. At 14:50, after the mail client has downloaded mail via POP3, it establishes an SMTP connection which is routed to server C.
5. auth-lookup is invoked, which checks for 1.2.3.4 in the shared memory table. As there is an entry, and it has not yet expired, auth-lookup then sets the $RELAYCLIENT environment

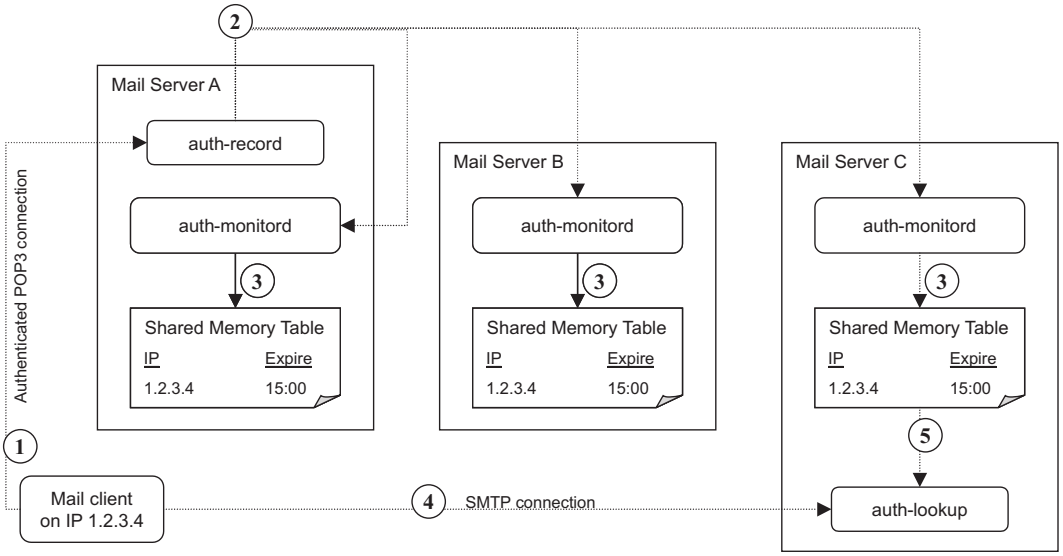

Figure 1: POP-before-SMTP example flow.

variable and invokes qmail-smtpd (not shown). The mail client on 1.2.3.4 will now be able to relay mail through server C.

Tarpit Implementation

Theory

An SMTP tarpit, as we have implemented it, starts off as a simple idea. If a particular remote system has sent us more than *r_max* mail messages, we insert a delay *D* between SMTP RCPT commands from the remote system and responses from our system. Then, as the number of messages continues above *r_max*, we increase delay *D* again once we have received *r_max* + *r_tarpit_inc* messages. We continue to increase *D* each time the total received messages[5] has increased by *r_tarpit_inc*. If we call the received message count *r_count*, we get:

$$\begin{aligned} &r\_count < r\_max \rightarrow D = 0 \\ &r\_count = r\_max \rightarrow D = 1 \\ &r\_count > r\_max \rightarrow D = 1 + Z \end{aligned} \qquad (1)$$

$$\text{where } Z = \text{int}\left(\frac{r\_count - r\_max}{r\_tarpit\_inc}\right)$$

(and the *int*() function returns the number given with any fractional portion stripped away, i.e., int(*x*) is 1 if ($x \geq 1$ and $x < 2$)).

We also set a maximum value for *D*, so that the SMTP conversation does not violate the relevant standards.[6]

Figures 2 and 3 illustrate how this should work for a single SMTP session. For these examples we've set *r_max* to 1000 messages, and *r_tarpit_inc* to 100. The delay value is zero for the first 1000 messages, and then increases in a linear fashion as more messages are sent in. The impact on how long it takes to send messages to more than a thousand recipients is shown in Figure 3. For this example, we assume that the sender is sending to 4000 recipients, and can send to five recipients per second. As shown, it takes less than four minutes to inject the first 1000 recipients. However, after this the sender is limited to one recipient per second (60 per minute) – and after another 100 recipients, 30 per minute, and so on. As the delay increases, the time needed to insert another 1000 recipients increases exponentially.

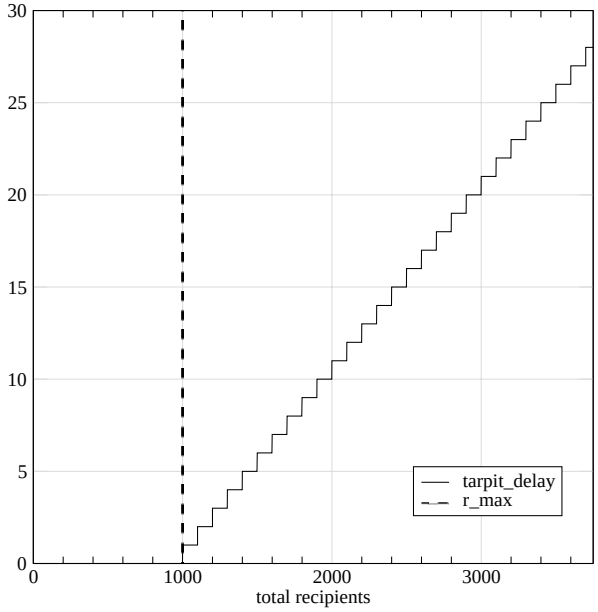

Figure 2: Behavior of Delay D as *r_count* increases.

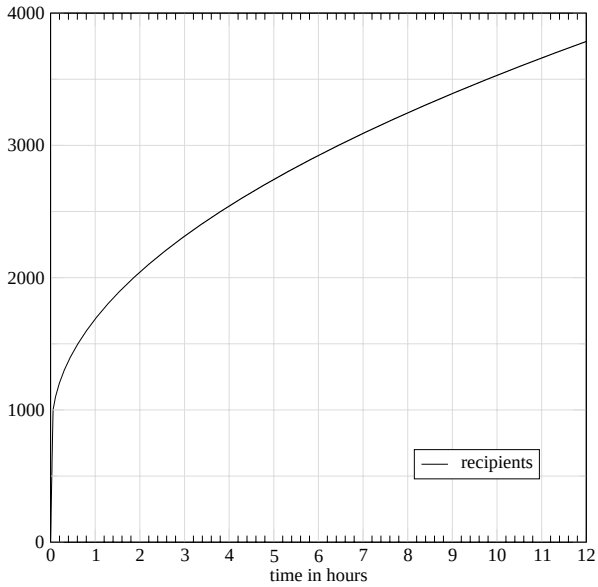

Figure 3: Cumulative recipients accepted over time.

Now, if we extend this idea beyond a single SMTP session, we need to store *r_count* and an IP address across sessions. We'll get into the detail of how this is done below – but you've already got the basics from the discussion of auth-monitord and friends. However, this brings up another problem. Any remote mail server will eventually build up an *r_count* larger than *r_max*, given enough time. We don't want to end up tarpitting every mail server that ever talks to us. It's impolite. Thus we need some way to periodically decrement the *r_count* values – slow enough that we'll catch real spammers, but fast enough that we don't tarpit benign mail servers.

[5] Unless specifically noted otherwise, when we're talking about number of messages in this paper, we're referring to the number of deliveries that the mail system is being asked to make. Someone opening an SMTP connection to our server can insert 100 messages with 1,000 recipients each – and the server will attempt 100,000 deliveries as a result. This is why the delay is inserted at the RCPT command, instead of any other.

[6] Section 4.5.3.2 of RFC 2821 [6] indicates that an SMTP client must wait five minutes for the response to an RCPT command. It goes on to say "A longer timeout is required if processing of mailing lists and aliases is not deferred until after the message was accepted." This could be interpreted to mean that a longer delay would not directly violate the standard.

Here's where things could get quite complicated – an accurate view of how many messages were received during a given time period would require multiple counters and at least one additional time counter. We don't want the shared memory table to be too large, so instead we will store a single count value that we'll periodically modify using two more variables, *r_divide_int* and *r_subtract_int*, in the following equation:

$$r\_count = \text{int}\left(\frac{r\_count}{r\_divide\_int}\right) - r\_subtract\_int \quad (2)$$

Now, we want to be polite and make sure that we don't tarpit every system that ever sends mail to us. However, should a system trigger a tarpit delay, we don't want the reduction operation to let them off too easily. Thus we add another threshold – *r_untarpit*. If a delay is in place, *r_count* must drop below this value in order for *D* to go back to zero. This changes Equation (1) a bit – we now need to talk about D_c (current delay value) and D_n (the next value after we decrement *r_count*).

$$\begin{aligned} D_c = 0 \;\&\; r\_count < r\_max &\rightarrow D_n = 0 \\ D_c > 0 \;\&\; r\_count > r\_untarpit \;\&\; & \\ r\_count < r\_max &\rightarrow D_n = D_c \\ r\_count = r\_max &\rightarrow D_n = 1 \\ r\_count > r\_max &\rightarrow D_n = 1 + Z \end{aligned} \quad (3)$$

$$where \;\; Z = \text{int}\left(\frac{r\_count - r\_max}{r\_tarpit\_inc}\right)$$

By this time we've added a bit of complexity to the simple idea of inserting a delay after a certain number of messages are received. However, we should note that the behavior, within an individual SMTP session, is still that of Equation (1), as seen in Figures 2 and 3. At the beginning of the session the values of *r_count* and *D* are the same as the shared values – the session will start with no delay if there is no shared entry, and will start with a delay if both shared values are greater than zero. After this point in the session the delay *D* can only go up – there is no reduction operation, and thus no need for an *r_untarpit* lower threshold. When the session ends, the count of RCPT commands received during that session is added to the shared *r_count* value. Only the shared *r_count* is subject to Equation (2), and only the shared delay D_n is calculated using Equation (3).

We'll see how this looks when we put everything together in the next section.

Data

The theory above has accumulated a few variables that need to be stored in the shared memory tables (the latter having been described in POP-before-SMTP section). Listing 1 is the C structure description of that table.

The *client_address* and *pop_expire_time* variables are simple – the former is the key field and indicates what IP we're describing. If the latter is set and the current time is less than the value, then SMTP relay is allowed from the IP address.

Next we have five variables related to receipt counts. *r_count* and *r_max* describe the current and first threshold recipient count value, as introduced in Equation (1) from the last section. The *r_count* is described as a rough guess based on the inexact nature of our reduction operation. *r_rop_time* specifies the time at which we will next perform the reduction operation detailed in Equation (2). Each time this operation is run, this value is set to the current time plus *r_rop_interval*, defined below. *r_untarpit* is the lower threshold introduced in Equation (3). Finally, *r_last_entry* is a timestamp indicating the last time that the *r_count* value was incremented.

The values *conn_count*, *conn_max*, *conn_rop_time*, *conn_untarpit*, and *conn_last_entry* are used to tarpit based on number of SMTP connections. If *conn_count*

```
struct table_entry {
  struct in_addr client_address;     /* Clients address */
  time_t pop_expire_time;            /* Expire time for POP check */
  int r_count;                       /* RCPT TO's issued (rough guess) */
  time_t r_rop_time;                 /* Time for next reduction op */
  int r_max;                         /* Max value for rcpt count before tarpit */
  int r_untarpit;                    /* Must drop below this to get out of tarpit */
  time_t r_last_entry;               /* For reporting */
  int conn_count;                    /* SMTP connections */
  time_t conn_rop_time;              /* Time for next reduction op */
  int conn_max;                      /* Max value for conn count before tarpit */
  int conn_untarpit;                 /* Must drop below this to get out of tarpit */
  time_t conn_last_entry;            /* For reporting */
  int tarpit_delay;                  /* Determines if and how severely to tarpit */
  int tarpit_max_delay;              /* Max delay for tarpit */
  int config_db_rev;                 /* If the IP is in the config DB.
                                        Values are -1 : error
                                                    0 : ip not in db
                                                  1,2 : alternating db versions */
}
```

Listing 1: Shared memory table entry structure.

goes above *conn_max* during the *conn_rop_interval* (defined below), *tarpit_delay* will be calculated in the same fashion as for excessive recipient counts. If a delay is warranted based on both excessive recipient and connection counts, *tarpit_delay* will be the sum of the two delay calculations.

As you may have guessed, *tarpit_delay* is the shared delay value. It is the starting delay value used in qmail-smtpd, and is recalculated each time that the *r_count* or *conn_count* values are incremented or decremented. However, it will not be set higher than *tarpit_max_delay*, a value customizable per IP address.

The final value stored, per IP, in the shared memory table, is the *config_db_rev*. If greater than zero, the IP address in this table entry is also in the configuration database. The configuration database gives us the ability to set custom values for *r_max*, *r_untarpit*, *conn_max*, *conn_untarpit*, and *tarpit_max_delay*. This allows us to disable tarpitting (or at least set very high thresholds) for systems like our own internal mail servers or other ISP's mail servers.[7] It also allows us to set lower than normal thresholds.

```
CREATE TABLE tarpit_authmonitor_config
(
    r_subtract_int MEDIUMINT NOT NULL,
    r_divide_int MEDIUMINT NOT NULL,
    r_rop_interval MEDIUMINT NOT NULL,
    r_max_default MEDIUMINT NOT NULL,
    r_untarpit_default MEDIUMINT NOT NULL,
    r_tarpit_inc MEDIUMINT NOT NULL,
    conn_subtract_int MEDIUMINT NOT NULL,
    conn_divide_int MEDIUMINT NOT NULL,
    conn_rop_interval MEDIUMINT NOT NULL,
    conn_max_default MEDIUMINT NOT NULL,
    conn_untarpit_default MEDIUMINT NOT NULL,
    conn_tarpit_inc MEDIUMINT NOT NULL,
    pop_timeout MEDIUMINT NOT NULL,
    tarpit_max_delay_default MEDIUMINT NOT NULL,
    notarpit MEDIUMINT NOT NULL,
    shm_table_entries MEDIUMINT NOT NULL,
    last_modified TIMESTAMP NOT NULL,
    created TIMESTAMP NOT NULL
);
```

Listing 2: SQL table definition for global tarpit configuration.

The two legitimate values for this variable are 1 and 2. The differing values are used to update the shared memory tables when a change is made to the configuration database. If all non-zero *config_db_rev* values in shared memory are '1', and the configuration database is updated, its revision will become '2', and the shared memory table will be updated accordingly. The next update will reuse the revision value '1'. While somewhat simplistic, this is sufficient for our needs, as database configuration changes are rare. If the value is less than zero, there was an error while reading the configuration from the database.

[7] We hereafter refer to the practice of setting very high thresholds for a particular IP as whitelisting that IP. Whitelisting can also refer to disabling tarpitting (setting *tarpit_max_delay* to zero) for a particular IP.

Our configuration database contains global settings, in addition to the per-IP custom settings mentioned above. Listing 2 shows the SQL definition for the table that holds these settings. The default values for the customizable settings are stored here (*r_max_default*, *r_untarpit_default*, *conn_max_default*, *conn_untarpit_default*, and *tarpit_max_delay_default*).

The database also stores values for the count reduction operation described in Equation (2) – *r_divide_int* and *r_subtract_int*. How often the reduction operation is performed is determined by *r_rop_interval*. The number of additional recipients needed to increment the tarpit delay, after *r_max*, is *r_tarpit_inc*. These values also have connection count counterparts.

The remaining miscellaneous settings are used to set the POP-before-SMTP timeout (*pop_timeout*), size of shared memory table (*shm_table_entries*), and whether or not tarpitting is in use (*notarpit*). This last value was used initially to give us an idea of what the code would do without actually delaying any SMTP sessions.

Figure 4 shows a series of four graphs that illustrate the theoretical performance of the tarpit code. These graphs were plotted using data from a perl program that simulates input to the mail system and the tarpit response. The following spammer and tarpit values were used:

```
Spam parameters
  Max simultaneous connections     100
  Messages/connection             1000
  Messages/second, per connection    5

Tarpit configuration
  Reduction interval (minutes)      15
  conn_max              (not simulated)
  r_subtract_int                     5
  r_divide_int                       2
  r_max                           1000
  r_untarpit                       100
  r_tarpit_inc                     100
  tarpit_max_delay (seconds)        30
```

Concentrating on the cumulative messages graph first, we see that tarpitting generally works, but perhaps not immediately as we suspect. During the first hour, the overall insertion rate is slightly less than 29/sec (1700/min) – 17 per minute per connection. After the first hour the injection rate settles to just under 3.4/sec. Still not ideal, but enough to keep the load on our servers reasonable. Were tarpitting disabled, the injection rate from this example would be about 250/sec. Over 24 hours the sender manages to inject nearly 400,000 recipients – this would have taken less than an hour without tarpitting.

Before examining some of the other interesting aspects of the system's behavior as a whole, let's look at the other variables graphed, and how they demonstrate various aspects of the system. *r_count* would exactly track the message total graph were it not for the

reduction operation, which we can see happening every 15 minutes. After hour one, when the overall injection rate becomes negligible, we can see the shape of Equation (2) on the *r_count* graph. The *conn_count* graph is quite similar in shape to that of *r_count* – the *tarpit_delay* graph would be as well, except for the limit imposed by *tarpit_max_delay*. We can see the effect of *r_untarpit* in the area where *tarpit_delay* stays at 8 between hours two and three – while the directly calculated *tarpit_delay* would have gone to zero almost immediately, the *r_untarpit* functionality keeps it at the last value calculated before *r_count* went below *r_max*. There's some argument to be made that the *tarpit_delay* should stay at the maximum value achieved until the count goes below *r_untarpit*.

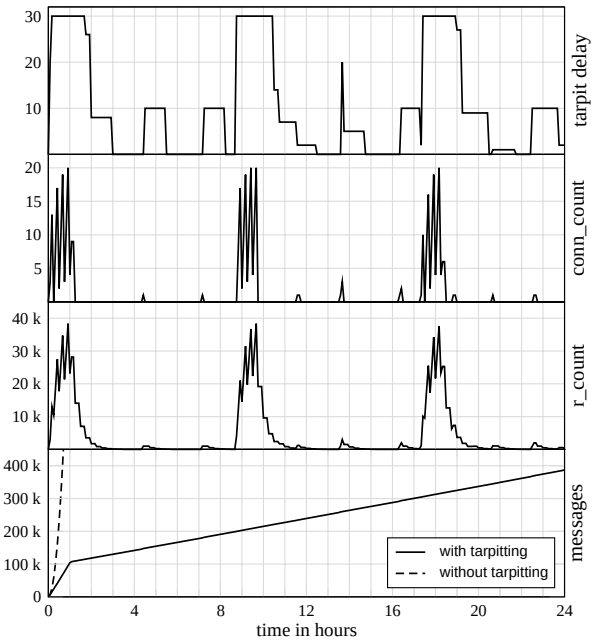

Figure 4: Theoretical example of cumulative messages, *r_count*, *conn_count*, and *tarpit_delay* values over one day.

Finally, why does the injection rate stay low, even when *r_count* and *tarpit_delay* go to zero at hour three? The key to this is the simple tarpit behavior shown in Figure 3. The initial connections that start up in hour zero have no initial tarpit value, and are never tarpitted because they deliver exactly *r_max* messages. However, after ten minutes, any new connections are started up with a *tarpit_delay* value of 30 seconds. While the shared *tarpit_delay* value is reduced every 15 minutes, the individual *tarpit_delay* value in each qmail-smtpd process is not. So each connection started with a *tarpit_delay* of 30 will take just over eight hours to deliver its 1,000 messages!

This is reflected in the *r_count* and *tarpit_delay* graphs – the bulk of the connections time out at eight hour intervals. At these points there are a few sender connections that get a zero *tarpit_delay*, complete quickly, and drive up the shared *tarpit_delay* again for the rest of the restarting connections. There are likely a few connections in each cycle that start when the shared *tarpit_delay* is between zero and 30 – this would account for the intermittent small bursts of activity, spaced two to four hours apart. These connections can be considered to be starting at various points on the Figure 3 graph, and taking as much time as needed to deliver 1,000 messages from that point. All of this serves to stagger out the restarting connections, so that we avoid seeing the 30 messages/sec slope at eight-hour intervals.

We didn't necessarily plan for it to work out this smoothly – the straight lines and periodic behavior are probably more a feature of the simulator than the system itself. Still, somewhat confident that it works well in theory, we proceed to setting it up to work in practice.

Mail System Modifications

As mentioned in the POP-before-SMTP section, our pre-tarpit mail system calls qmail-smtpd as part of an exec chain:

tcpserver → auth-lookup → qmail-smtpd

To implement tarpitting, we need auth-lookup to pass tarpit details to qmail-smtpd, and we also need to be able to write additional details into the shared memory table. So with tarpitting, our exec chain looks like this:

tcpserver → auth-lookup →
qmail-smtpd → auth-record

auth-lookup

auth-lookup checks for an entry in the shared memory table corresponding to the IP address passed by the tcpserver process. It also pulls generic configuration information out of a second shared memory table, copied there from the configuration database by auth-monitord.

The four values that it passes to qmail-smtpd (via the environment) are $TARPIT_RCPT_REMAINING, $TARPIT_DELAY, $TARPIT_MAX_DELAY, and $TARPIT_INCREMENT (*r_tarpit_inc*). The first represents the remaining recipients allowed before *tarpit_delay* is incremented – this is calculated in auth-lookup to minimize the amount of code we're adding to the already weighty qmail-smtpd.

The values passed are based on the values stored in shared memory for the source IP address of the connection, or default values if the IP is not in the main shared memory table. All values are passed as zero if the global configuration variable notarpit is set to 1.

qmail-smtpd

We only make three, rather simple, modifications to qmail-smtpd. The first is pulling the values set by auth-lookup out of the environment. At this point, if *tarpit_r_remaining* and *tarpit_delay* are zero, tarpitting is disabled in this instance of the process. As zero is the default value for these two variables, tarpitting

will also be disabled if the environment variables $TARPIT_RCPT_REMAINING and $TARPIT_DELAY are not set.

The second modification is to the smtp_rcpt subroutine. This is the subroutine that is called to parse the RCPT command. At the end of the subroutine, after all the regular checks and parsing, we add a simple few lines, shown in Listing 3, that actually implement the tarpit.

```
/* tarpitting */
if (r_remaining == 0 &&
       (tarpit_delay == 0 ||
        tarpit_increment > 0)) {
    if (tarpit_delay < tarpit_max_delay)
        tarpit_delay++;
    r_remaining = tarpit_inc;
}
if (notarpit == 0 && tarpit_delay > 0)
    sleep(tarpit_delay);
if (r_remaining > 0)
    r_remaining--;
r_count++;
```

Listing 3: C code in qmail-smtpd's smtp_rcpt subroutine to implement tarpitting.

The first if clause checks to see if the internal *tarpit_delay* count needs to be increased. The increase will happen if *r_remaining* is zero and either *tarpit_delay* is zero (the initial case where *r_count* has reached *r_max*), or *tarpit_increment* is greater than zero. The *tarpit_increment* check is to verify that the configuration is in a sane state. Even if all conditions are set for an increase, *tarpit_delay* is not increased above the *tarpit_max_delay* threshold.

We then simply sleep() for *tarpit_delay* seconds, as long as *tarpit_delay* is non-zero and tarpitting is enabled. Finally, we decrement *r_remaining* and increment *r_count*. Even though one is more or less a reflection of the other, their relationship is complicated enough (see Equations (1) and (2)) to merit recording them separately.

The last modification to qmail-smtpd allows it to send the per-session recipient count information on to the other systems. The main() subroutine is modified to accept a command-line argument (the next program in the exec chain) and store it in the global variable "argl". Then various relevant exit points are modified to write qmail-smtpd's internal *r_count* into the environment variable $TARPIT_RCPT_COUNT, after which the next program in the command chain is invoked with execvp(). As mentioned above, this next program is auth-record.

auth-record

The modifications to auth-record are fairly short. An initial check is made to see if the $TARPIT_RCPT_COUNT environment variable exists and is non-null. If so, a multicast packet is sent containing the recipient count value and a singular increment to the connection count.

auth-monitord

auth-monitord is the daemon that establishes the shared memory table used to store information about POP-before-SMTP. It also records updates, regularly scans the table to expire old entries, and will send out the contents of its table to populate another server's table.

Updated for tarpitting, auth-monitord has three new features: an additional shared memory table to store global configuration settings; global and per-ip settings are read from a database and written to shared memory at regular intervals; and the main shared memory table now stores tarpitting information. Individual *tarpit_delay* values are recalculated after per-IP updates and during each table scan.

Pseudocode for the startup and main loop of auth-monitord is shown in Listing 4. Most of it should be fairly self explanatory – the *tarpit_delay* calculations and reduction operations are as in the Theory section. There are, however, a few details worth noting. The first is that the shared memory table is hashed for faster access, as our current table size is about 16,000 entries (average number of valid entries on a weekday is about 7,600). This uses nearly a megabyte of memory.

auth-monitord is started at boot time, so it's almost certainly the first program on the system to open the multicast session. When opening the socket to be used for multicast communication, we set SO_REUSEPORT on the socket so that the same port may be used by multiple programs. In addition to reads and writes from auth-monitord, one or more copies of auth-record may be writing at the same time.

```
struct auth_message {
  int op_code;
  int table_len;
  int version;
  struct table_entry entry[0];
};
```

Listing 5: Multicast message structure.

Listing 5 shows the one simple message structure used by auth-record and auth-monitord for passing information. The op code is either AUTH_ADD, for passing one table entry (table_entry is defined in Listing 1), AUTH_DUP, for passing multiple table entries, or AUTH_GET, for requesting table entries from other auth-monitords. The routine that sends the contents of a shared memory table does so by sending multiple small packets – each less than the size of the ethernet MTU. The packets are sent with a 1 to 10 millisecond delay between each one, in an attempt to avoid flooding the network. An average update only takes a few seconds.

When an AUTH_GET request is issued, every mail server sends out an update except the one that requested the update. This results in multiple table dumps going over the network at the same time – and all auth-monitords process every update packet. While not terribly efficient, this mechanism is simple to implement, only happens when a mail system is

rebooted, and ensures that all systems are in sync. When a table entry is received, via AUTH_DUP, that already exists in the table, the newer of the *last_entry* times are used, and the entry with the larger (connection/recipient) count prevails, determining both the count and the next reduction op time.

The other source of shared memory table updates, aside from auth-record messages, is the configuration database. The database is re-read about every five minutes, and the *config_db_rev* variable mentioned earlier is used to implement any changes needed to individual table entries. The update routine sets *config_db_rev* to 1 if the previous revision was 2 or an error (zero or less), and to 2 otherwise. It then writes all per-IP configuration settings from the database into the shared memory table, marked with the current *config_db_rev*.[8] When the subsequent database scan is performed, entries with a revision of less than one have values like *r_max* checked against the database default, and updated if the default has changed. If an entry with custom settings is not marked with the current *config_db_rev*, it must have been removed from the database, and is therefore removed from the table.

Finally, it's worth noting that auth-monitord also uses two sanity check routines quite frequently: table_check verifies that shared memory data structure pointers are valid, and is run every time the table is searched or scanned; table_check_entry verifies that a particular entry has expected values, and is run each time an entry is added or updated. If these checks were not performed, a corrupted entry or table could either shut down the SMTP service or turn the system into an open relay – either option is highly undesirable.

[8]This means that the shared memory table always contains an entry for an IP that has custom thresholds in our database. Given that we currently only have 100 custom entries, this isn't yet a problem.

User Interface

A user interface is needed for tarpitting for three reasons – as a debugging tool, to allow changes to the configuration, and to tell us who is being tarpitted so that they may be either whitelisted or blocked from sending mail entirely (blacklisted). Almost coincidentally, we have a tool for each reason: auth-dump, a web interface, and auth-watch.

```
Load initial configuration from database (including shared memory table size);
Establish shared memory table and table data structure;
                /* entries are hashed by last octet in IP address */
Copy configuration settings from database into shared memory;
Build multicast network socket;
Load per-IP configuration information from the database and place it in shared memory;
Send message with op code AUTH_GET to multicast port;
while (TRUE) {
    Sleep until a message arrives or 5 seconds passes;
    if (A message has arrived) {
        Read message from network;
        Discard message if source is a unicast address we don't trust;
        if (message_op_code == AUTH_ADD) { /* single entry from auth-record? */
            Sanity-check entry, add it to the table;
            Recalculate its tarpit_delay ;
        } elseif (message_op_code == AUTH_DUP) { /* multiple entries from another
                                        auth-monitord, in response to an AUTH_GET */
            Sanity check (multiple entries); add them to the table;
            Recalculate tarpit_delay values;
        } elseif (message_op_code == AUTH_GET) {
            Send contents of our shared memory table;
        }
    } /* end if a message has arrived */
    if (It has been five minutes or more since last table scan) {
        Load global and per-IP configuration information from the database;
        Update copy in shared memory;
        /* Scan table: */
        Expire POP times if appropriate;
        Perform connection and receipt-count reduction operations that are due;
        Remove empty entries;
        Insert new default values if appropriate;
        Recalculate tarpit_delay
    }
}
```

Listing 4: Pseudocode for auth-monitord.

- auth-dump does what it says on the tin. For each entry in the shared memory table, a line is printed with the entry values separated by colons. Useful, but not exactly user friendly.
- Our web interface, however, is user friendly. It provides the ability to change global configuration settings, list per-IP entries, as well as add, modify, or delete per-IP entries. It also provides a snapshot view of the current shared memory table, allowing us to see which currently active systems have custom thresholds.
- The final tool, auth-watch, is a perl script that provides a running/historical view of the shared memory table. As seen in Figure 5, it shows the current *r_count* and *tarpit_delay* values and the highest values seen during the run of the program. The 'diff' value shown is the difference between the current *r_count* and the one five minutes ago – indicating how active the sender is.

Tarpitting in Practice

Policy

Much of the problem in detecting and preventing spam is finding the line between what is and is not spam. Similarly, tarpitting must be tuned to find the balance between not catching spammers and unduly impeding legitimate customers. Even if we achieve this goal successfully, what do we do if we find a customer system in the tarpit? Here we must decide between two more extremes: do we immediately set custom parameters, so that the customer is not tarpitted again, or do we assume there's a problem on the customer's system and block their ability to send until they resolve it? If we take the position that the customer is always right – and in turn fail to detect a customer open relay sending mail through our servers, this may result in the ISP's mail servers being added to one of the many distributed block lists. This in turn harms all customers.

Finding these balances is a bit of a thorny problem. It's also clearly not something that technology can solve. The solution, or at least the agreed way to go in looking for it, should involve the people who will have to clean up if things get messy – customer support, sales, and management. To this aim, we developed a policy document before we brought the system live.

A policy document that deals with tarpitting may also want to deal with general questions about the use of SMTP services. Terms like SMTP and blacklisting, and concepts like relaying need to be clearly defined. Some of the questions that such a document might address:

- Who will we relay for?
- Who will we accept mail for?
- Who will be tarpitted?
- What happens when we find someone in the tarpit?
- Adding to and removing from an (internal) whitelist or blacklist.
- Monitoring and reporting.

Having such a policy, agreed upon and signed off, is unlikely to protect you from the thorns – but it should provide something of a cushion.

Performance

The tarpit software has been installed on our production systems since November of 2002. However, it was in measurement-only mode (*notarpit*=1) until mid-February of 2003. At that point the ability to delay mail was enabled, but the thresholds were still set quite high. A few weeks later, we were reasonably sure that the system was working as expected, without any side effects. At this point we lowered the thresholds to realistic values.

Prior to this point we'd begun to see intense bursts of incoming mail on a regular basis. These incidents would drive our delivery attempts up to five or ten times the normal rate, resulting in some combination of: load spikes, mail delays, queues overflowing with undeliverable bounce messages, and/or bursts of mail from our systems to one or more external ISPs. The latter resulted from spam sent to invalid addresses bouncing to a fake return address at the ISP or ISPs in question.

```
Top auth-dump entries [rcpts>50], by rcpt count [log 59870: 1384.7]
Sat Jul 19 02:01:52 2003 - Mon Aug  4 02:32:34 2003

   IP Address               r_count    diff   conn_count   tarpit_delay
 1 17.16.128.142            1675/2943  529       214           7/022
 2 17.18.233.246             365/0782  0         6             0/000
   host.domain.com
 3 17.17.54.155              256/0488  92        256           0/000
   yetanother.net.
 4 192.68.62.241             170/0170  0         3             0/000
 5 17.21.82.39               140/0520  39        5             0/000
 6 192.168.171.156           122/0122  35        122           0/000
 7 17.31.38.230              112/0112  15        3             0/000
 8 192.168.52.10             110/0110  0         1             0/000
   obvious.spammerdomain.name
 9 192.168.48.10             108/0108  12        4             0/000
10 17.17.122.2               100/1168  41        327           0/000
11 17.22.191.99               90/0369  -3        4             0/000
12 17.18.234.10               90/0140  -9        24            0/000
```

Figure 5: auth-watch output.

Since early March we have not been as drastically affected by spam. This does not mean that we have eliminated it entirely. We've even experienced a few more burst incidents, but tarpitting has kept the volume of the bursts down to only 2-3 times our normal traffic.

Figure 6 shows data gathered from an actual spam sender. This set of data conveniently shows what works, and what doesn't work, about tarpitting. In the first two hours of the data set, recipient addresses are sent in at a fairly rapid pace, and tarpitting kicks in after 15 minutes – on average the rate is 7 messages/sec, and more than 40,000 messages are sent in total. The fluctuations in *tarpit_delay* are directly related to the reduction operations on *r_count*.

The sender appears to have noticed that tarpitting is taking place, and then tries to work his way around it. Over three hours, 3000 messages are sent at a rate of about 20/min. Entirely reasonable. Finally, at 6.5 hours, things start to pick up again. Over the next 4.5 hours, mail is sent at a more rapid pace, but tarpitting isn't triggered. The sender doesn't come to our attention except during the last burst at the end. Still, in order to stay under our radar they had to keep their send rate at two messages/sec.

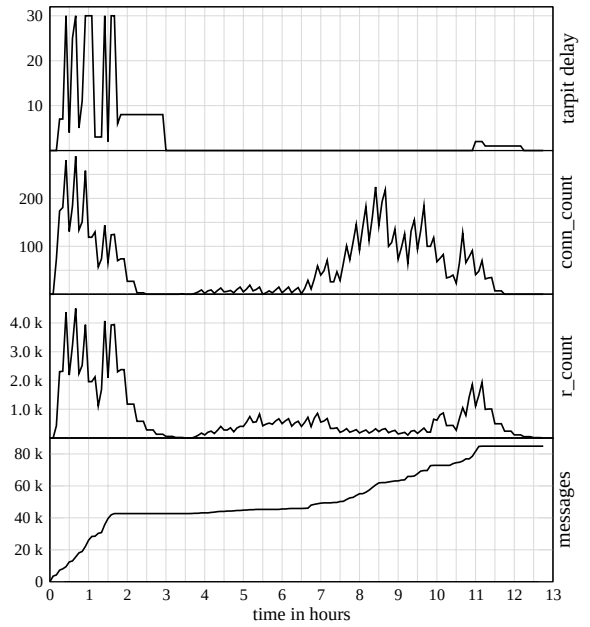

Figure 6: Actual example of cumulative messages and tarpit values for sender seen on July 25.

Overall, the tarpit software is doing a good job of preventing denial-of-service levels of spam. Yet we'd prefer that anybody sending 40,000 messages in a four or five hour period was brought to our attention, and either whitelisted or blacklisted.

Future Work

The first focus area for improvement is the 'grace period' – this is the time period where a spammer opening multiple connections can get a large number of messages in before tarpitting starts. Multiple connections opened at the same time, when the shared *tarpit_delay* is zero, result in (*r_max* × *connections*) messages getting in without being delayed. This also accounts for the initial steep slope in the Figure 4 messages graph, which we have seen in practice. So our first idea for improvement was to ask qmail-smtpd to send incremental updates – instead of updating the shared memory table at the end of a session, update it every time 100 RCPT commands have been issued.

In an "ideal world" situation with lots of connections opening in a short period of time, and messages being sent as quickly as possible for as long as possible, this doesn't make a lot of difference. Under these conditions, data in the shared memory table only benefits new connections – and the bulk of the new connections happen all at the same time as seen in Figure 4. However, if new connections are established in a more random fashion, incremental updates would help ensure that more accurate estimates are available to the connections. In either the ideal or real world cases, incremental updates also give an administrator a more accurate view of the state of the tarpit system.

Another accuracy issue is shared table synchronization across systems – as is, tables are never going to be entirely in sync: each system does its own reduction operations, which won't happen at the same time, and multicast is not a reliable protocol, so there is the potential for lost updates. We've discussed a few ideas for improvement, but generally we've tried to avoid depending too much on strict synchronization – distributed systems are an entire field of study in themselves, and not what we want to concentrate on.

As an alternative solution, we could limit the number of recipients per connection. A related idea would be limiting (across all servers) the number of connections that a remote system is allowed to open at one time.[9] To do this accurately we would need to send a connection-count update at the beginning of the qmail-smtpd session, changing our exec chain:

tcpserver → auth-lookup → auth-record →
qmail-smtpd → auth-record

The first auth-record would be configured to only send a connection count update, and the second to only send a recipient count update, in order to keep the connection counter accurate. The auth-lookup would be configured to exit directly, dropping the connection if the connection count read was at or higher than the maximum. Going back to our theoretical example, we see the following results:

[9] The connection count values in the shared memory table were originally intended to implement connection tarpitting, trapping one-message-per-connection spammers. Observation has shown that our current setup would likely only catch customers with misconfigured mail servers that open multiple SMTP connections without sending anything.

	Messages/sec	
	First hour	Thereafter
Figure 4	30	3.4
Limit recipients per connection by half	16	3.5
Limit connections by half	15	1.8
Both of the above	8.1	2.0

While primarily intended to reduce the initial input rate, these ideas also seem to help with the steady-state rate. Unfortunately our simulator doesn't quite seem to emulate the somewhat less steady traffic as seen in the real world example of Figure 6. One of the ideas that this graph suggests to us, but which would likely show no effect on our simulation, is a less-drastic reduction operation formula. A less-drastic drop (perhaps achieved merely by changing *r_divide_int* to be a floating point value) should remove the rapid fluctuations seen in shared *tarpit_delay* value during the first wave of messages. This also has its downside – the reduction operation is designed to be fairly rapid, in order to avoid penalizing our dynamic IP customers. If one customer were to send mail at a rate high enough to trigger a tarpit delay, and then disconnect, a delay that remained in place for several hours would end up affecting any other customer that was allocated that same IP address. This is clearly something we want to avoid.

We could, however, implement slower reduction operations if we were able to add per-network entries to the database and shared table (as opposed to per-IP entries). We would also make *r_divide_float* one of the customizable values. These changes would enable us to make the default reduction curve longer than that used for our own dynamic-IP networks. This would also enable us to change other defaults for our customers in general – giving us the ability to tighten controls with less risk of reducing service to the customers. A similar idea would be to not only set defaults by network, but tarpit by network – helping us catch spammers who establish multiple connections from multiple contiguous addresses.

A further obvious way to reduce the steady-state input rate, without the need for any code changes, is to increase the *tarpit_max_delay*. Applied to our theoretical example this has no effect on the initial rate, and drops the long-term rate to 1.9 messages/sec. The relevant RFC seems to allow us up to a five minute delay value. These delays not only reduce the incoming rate of spam, but extract a cost – in system resources – from the sender. However, they also extract a cost from us.[10] It might therefore be a good idea to drop all connections from the sender's IP address after they have reached the maximum tarpit delay level for a certain period of time. An extension to this might be a separate timeout (longer than the reduction operation would dictate) for those IP addresses that end up in a drop-all-connections state – making our system a bit more like Deny-Spammers [14].

The final idea we've had on how to improve tarpitting would be to 'seed' the configuration database with information from a network blocking list. Entries could automatically be inserted with thresholds that would cause connections from the listed addresses to be given an immediate delay. Thus mail would still be allowed in, at very slow rates, from senders who may have been listed incorrectly, or who have fixed their spam problem but not yet been removed from the list. It would also give us the local control we desire – an entry in the configuration database with higher-than-normal settings, or a 'do not blacklist' flag, would not be overwritten by a program loading entries from a blocking list.

Conclusion

At the end of 2002 we were faced with an unexpectedly high volume of incoming mail, and with extremely high volume bursts of mail from individual sources. Both problems were largely attributable to a rising tide of unsolicited commercial email. The implementation of tarpitting was intended to dampen the burst attacks, and greatly slow the flow of mail from point sources sending tens or hundreds of thousands of messages. It was also intended to give us a fine degree of control over its behavior, have minimal impact on the customers, and require little change to our existing mail systems.

Tarpitting has definitely helped with the burst attack problem – in the five months since it was turned on, we have not seen any concentrated attacks of the magnitude that we were seeing on a regular basis before that time. Unfortunately, in its current state, it has not solved the general incoming spam problem. As our real world example demonstrated, it's still possible to inject 40,000 messages into our system, in less than six hours, without triggering a tarpit delay. However, we've got many ideas for improving it, and we've even theoretically demonstrated that some will work. Overall it's a useful addition to our spam-fighting toolbox.

There may be other tools out there that would suit us – even tools designed for ISPs much larger than ourselves. One problem with being an ISP is that you are a large target for those sending spam. One can easily see how this would make an organization reluctant to publish any but the vaguest details about how they are fighting spam – much less actual tools. It's clear that those who distribute spam spend as much time finding new ways to send it as we do finding ways to stop it. Thus a pessimistic viewpoint would say that writing this paper just makes their job

[10] One suggestion for fighting the spam problem is to find a method of charging the sender for each message sent. D. J. Bernstein proposes that all mail be stored on the sender's system, or at the sender's ISP [15]. Adam Back's Hashcash system would charge the sender in CPU cycles, by requiring them to produce a token that is difficult to compute but easy to verify [16]. Microsoft's Penny Black research project is investigating several sender-pays techniques [17].

easier. In writing this paper we hope to find a middle ground between hiding everything and full disclosure. Better coordination and information sharing between ISPs (and other large mail volume sites) would lower the volume of spam for everyone.

Availability

How and what code to make publicly available is still under review. If the code is made available, a pointer to it will be available from http://www.qmail.org.

Acknowledgments

The authors would like to thank Erin Hunter and Aoife Cox, as well as Jerry Connolly, Dave Ryan, and Frank Slyne for their assistance in reviewing this paper. We would also like to thank Deeann Mikula for her assistance and shepherding, as well as AEleen Frisch and Rob Kolstad for their incredible patience.

Finally, we would like to thank our management at eircom.net for allowing us to pursue interesting (and productive) ideas, and to share the results.

Author Information

All of the authors have worked at eircom.net for past several years.

Tim Hunter is a systems administrator and manager of a sysadmin team at eircom.net He holds a Bachelor's degree in Electrical and Computer Engineering from the University of Colorado at Boulder, USA. He has worked in systems administration from internal support to on-site customer support in places far away, and currently sits somewhere in the middle. After hours he can occasionally be found running, salsa dancing, hiking, or climbing. He can be reached at tim.hunter@eircom.net .

Paul Terry is a sysadmin at eircom.net. He graduated with a BSc in Mathematics and Mathematical Physics from NUI Maynooth, Kildare, Ireland and has since worked in various roles from operating Macs to coding things for various ISP systems. When not at work, Paul can be found clinging desperately to some cliff shouting at the poor soul who has volunteered to belay him, or running up some mountain, then wondering what the hell he is doing up there and running back down again. Paul can be contacted at paul.terry@eircom.net .

Alan Judge is Head of Research and Development at eircom.net. Most recently concentrating on product development and ISP strategy, he has worked in operations, systems and network administration, campus and ISP architecture, data centre design, and object-oriented distributed systems. He started out as a student admin on the first Unix machine in Ireland and got his first job working for the first ISP in Ireland. He holds Ph.D. and B.A. (Mod) degrees in Computer Science from Trinity College, Dublin, Ireland. He can frequently be found carrying one of the excessive number of cameras he owns or in hiking boots, or both. He can be reached at Alan.Judge@eircom.net .

Disclaimer

References

[1] Schwartz, Alan and Simson Garfinkel, *Stopping Spam*, O'Reilly & Associates, http://www.oreilly.com/catalog/spam/ , October, 1998.

[2] CBS Evening News, *No End In Sight For Spammers*, http://www.cbsnews.com/stories/2003/06/18/eveningnews/main559272.shtml , June 18, 2003.

[3] Commission for Communications Regulation, *Quarterly Market Report – Internet Survey*, Ref 03/67c, http://www.comreg.ie/publications/default.asp?ctype=5&nid=101030 , May, 2003.

[4] Clark, Matthew, "Spammers Clog Eircom Mail Server," *electricnews.net*, http://www.enn.ie/news.html?code=8678739 , November 1, 2002.

[5] Details on Ireland's implementation of the EU data protection act can be found at http://www.dataprivacy.ie . We're honestly not sure that this would apply to direct processing of customer email without the customer's explicit consent – but we wouldn't want to get into an argument about it either.

[6] Klensin, J., Editor, *RFC 2821: Simple Mail Transfer Protocol*, http://www.ietf.org/rfc/rfc2821.txt , April, 2001.

[7] Geller, Tom, "SpamCon Foundation newsletter #013," http://www.spamcon.org/about/news/newsletters/013/opinion.shtml , 11 September, 2001.

[8] Bernstein, D. J., *qmail web page,* http://cr.yp.to/qmail.html .

[9] Inter7 Internet Technologies, *vpopmail web page*, http://www.inter7.com/vpopmail.html .

[10] NetApp, http://www.netapp.com .

[11] de Goyeneche, Juan-Mariano, *Multicast over TCP/IP HOWTO*, v1.0, http://www.tldp.org/HOWTO/Multicast-HOWTO.html , 20 March 1998.

[12] Bernstein, D. J., *ucspi-tcp web page*, http://cr.yp.to/ucspi-tcp.html .

[13] Johnson, Chris, *tarpit.patch*, http://www.palomine.net/qmail/tarpit.patch .

[14] Mikula, Deeann M. M., Chris Tracy, and Mike Holling, "Spam Blocking with a Dynamically Updated Firewall Ruleset," *LISA 2002*.

[15] Bernstein, D. J., *Internet Mail 2000 web page*, http://cr.yp.to/im2000.html .

[16] Back, Adam, *Hashcash – A Denial of Service Counter-Measure*, http://www.hashcash.org/hashcash , 1 August, 2002.

[17] Microsoft Research, *The Penny Black Project web site*, http://www.research.microsoft.com/research/sv/PennyBlack/ .