E-Mail-Adressenjägern auf Webseiten eine Falle stellen

Ernte – nein danke

Spammer ernten die Mailadressen ihrer Opfer am liebsten von Webseiten. Ziel sollte also sein, diese Harvester lahm zu legen. Das hier beschriebene PHP-Skript lässt die Spammer in eine Falle tappen und zeigt dabei, wie in PHP die Interprozesskommunikation über Semaphore funktioniert. Tobias Eggendorfer

Spam ist lästig, kostet Bandbreite und Zeit und stört die Kommunikation. Wer nicht erst auf den elektronischen Briefmüll warten will, um ihn dann mühsam auszusortieren, bekämpft das Übel schon bei der Entstehung. Zum Beispiel indem er Mailadressen auf seiner Homepage so tarnt [2], dass die Spammer mit ihren automatischen Sammelprogrammen [1] sie glatt übersehen. Website-Betreiber können diese Spider, Roboter oder Harvester genannten Programme zusätzlich ärgern und sie in Teergruben festsetzen (siehe Kasten „Teergruben und Honigtöpfe“).

Kostenkontrolle

Ziel ist es, die Sammelgeschwindigkeit zu reduzieren, um den Harvester zu bremsen. Es wäre auch möglich, die Kosten des Adressensammelns durch höheren Bandbreitebedarf zu steigern. Allerdings schädigt das den Falschen – den Betreiber der Website. Der E-Mail-Harvester sollte sich besser in einer Endlosschleife verfangen, die nur sehr langsam neue Daten ausliefert. Dabei muss die Webseite ständig neue Inhalte generieren, um den Spider zu täuschen und ihn möglichst lange zu binden.

Das lässt sich zum Beispiel per PHP-Skript erledigen. Es gibt eine HTML-Seite aus, die n-mal unter verschiedenen URLs auf sich selbst verweist und so eine Endlosschleife erzeugt. Dabei ist einiges zu beachten:

- Anständige Spider (etwa Googlebot) sollen nicht in die Falle tappen.
- Die generierten Links (die wieder zum Skript führen) dürfen sich nicht nur in ihren Parametern unterscheiden, sondern müssen unterschiedliche Dateinamen tragen. Viele Spider ignorieren Parameter, um sich nicht von Session-IDs in die Irre führen zu lassen.
- Der Server soll nicht unter der Last der gefangenen Harvester zusammenbrechen.

Der letzte Punkt ist schwierig, denn es gilt, die Zahl der parallel laufenden Prozesse zu begrenzen. Apache bietet zwar ein solches Limit, nur gilt es für alle Serverinstanzen. Das Skript braucht aber eine eigene Grenze. In der »ps«-Ausgabe kann es nicht erkennen, wie oft es schon gelaufen ist: Der Interpreter ist in Apache eingebettet. Lockfiles [3] verwenden wäre zwar möglich, aber durch die Zugriffe auf das Dateisystem wenig performant und zudem fehleranfällig. Besser geeignet sind Semaphore [4]. Sie verhindern, dass zu viele parallele Prozesse gleichzeitig einen kritischen Abschnitt betreten.

Ein kritischer Abschnitt ist etwa der Zugriff auf den Drucker: Drucken zwei Prozesse gleichzeitig, wäre das Ergebnis ein Werk für die Tonne. Beim Drucken setzt Linux zwar auf komplexere Mechanismen (unter anderem Warteschlangen), jedoch illustriert diese Anwendung die Aufgabe von Semaphoren: Prozesse dürfen sich bei bestimmten Aktionen nicht gegenseitig stören.

Bitte warten

Als Markierung für den kritischen Abschnitt dienen die Dijkstra-Operationen P(s) und V(s). P(s) kennzeichnet den Beginn des Abschnitts (Prüfen), V(s) verlässt ihn, s bezeichnet den Semaphor. P prüft, ob der Semaphor einen Wert größer null hat. Wenn ja, dekrementiert sie s und das Programm kann den kritischen Abschnitt betreten. Andernfalls wartet der Prozess, bis der Semaphor wieder einen Wert größer null annimmt. Analog inkrementiert V(s) den Semaphor um eins. Beide Operationen sind atomar, also nicht unterbrechbar.

Ein binärer Semaphor nimmt nur die Werte 0 und 1 an. Es gibt jedoch auch Semaphore mit einem beliebigen, positiv-ganzzahligen Ausgangswert n. Es können dann n Prozesse gleichzeitig den kritischen Abschnitt betreten – die Harvester-Falle setzt sich eine Obergrenze der parallel laufenden Instanzen. Das System kümmert sich um die Limits sowie um die Aufräumarbeiten: Sollte der Prozess, der den Semaphor hält, abstürzen, gibt Linux die Sperre wieder frei. Das ist ein großer Vorteil gegenüber Lockfiles, die stehen bleiben.

Semaphore in PHP

PHP-Skripte können Semaphore verwenden, wenn der Interpreter mit Unterstützung für System-V-Semaphore kompiliert wurde. Dazu musste die Configure-Option »--enable-sysvsem« gesetzt sein. Unter Umständen ist PHP neu zu übersetzen. Dann stehen die Funktionen »ftok()«, »sem_get()«, »sem_acquire()«, »sem_release()« und »sem_remove()« zur Verfügung.

Die Funktion »ftok()« erzeugt einen für Semaphore geeigneten eindeutigen Schlüssel (Listing 1, Zeile 31). Sie erwartet zwei Parameter: den Dateinamen des Skripts und einen Projektnamen. Der Dateiname ist am einfachsten über das Makro »__FILE__« zu erhalten.

Der dann folgende Aufruf von »sem_get()« erzeugt in Zeile 33 den Semaphor. Die Funktion erhält als Parameter den per »ftok()« generierten Schlüssel und den Ausgangswert des Semaphors, zusätzlich optional die Zugriffsrechte.

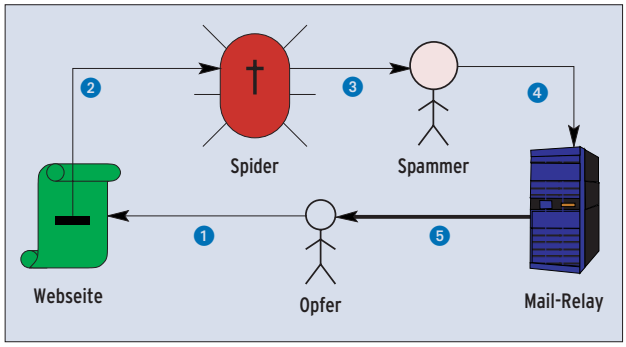

Abbildung 1: Diesen Spam-Kreislauf gilt es zu stören: Das Opfer veröffentlicht seine Mailadresse auf einer Webseite (❶). Der Spider sammelt die Adressen ein (❷) und teilt sie dem Spammer mit (❸). Der sendet seinen elektronischen Müll meist über Mail-Relays (❹) an die Opfer (❺).

Zugriffsrechte setzen

Diese Rechte sind denen im Unix-Filesystem ähnlich. Meist genügt es, wenn der Owner-Prozess das Schreib- und Leserecht für den Semaphor hat (Listing 1, Zeile 5). Die Standardeinstellung ist weniger restriktiv und erlaubt allen zu lesen sowie dem Owner und seiner Gruppe zu schreiben. Schreib- und Leserecht für den Owner muss sein, da sonst die P- und V-Operationen scheitern.

Die »sem_get()«-Routine erzeugt einen Semaphor nur dann, wenn er noch nicht existiert. Andernfalls gibt sie ein Handle auf den bereits bestehenden Semaphor zurück. Die zusätzlichen Parameter wertet sie nur beim Erzeugen aus; bei weiteren Aufrufen stören sie aber nicht. Es ist sogar sinnvoll, stets alle Parameter zu übergeben, da die beteiligten Instanzen

Teergruben und Honigtöpfe

Honigtöpfe und Teergruben sind das Internet-Pendant zu mit Bier gefüllten Schalen in zahllosen Schrebergärten: Sie dienen als Falle für unerwünschte Gäste. Spammer, Cracker und ähnliches Gesindel bleiben kleben und sind aus dem Verkehr gezogen. Beide Begriffe werden heute meist synonym verwendet. Ursprünglich sollten Honeypots Angreifer anlocken, um sie zu beobachten und zu erforschen. In Teergruben soll das virtuelle Ungeziefer dagegen stecken bleiben, ähnlich den Schnecken in den Bierschalen der Hobbygärtner.

Viele Anwendungen für das Funktionsprinzip

Die Begriffe bezeichnen nur ein Funktionsprinzip und keine konkrete Umsetzung. Beispielsweise untersucht Richard Stevens [7] Honeypots als Ergänzung für Netzwerk-basierte Intrusion Detection Systeme (NIDS), etwa Snort. Lutz Donnerhacke [8] verwendet ähnlich wie Daniel Rehbein [9] eine Teergrube, um den Versand von Spam zu erschweren. Der vorliegende Artikel zeigt, wie eine Teerfalle die Harvester von Spammern in die Irre führt.

Ein Honeypot wirkt auf Angreifer einladend, wenn viele Ports offen stehen und ältere Softwareversionen installiert sind. Damit stehen zahlreiche Sicherheitslöcher mit bekannten Exploits zur Auswahl. Dies simuliert der Honeypot nur und stellt sicher, dass er nicht als solcher erkennbar ist. Der Übeltäter weiß nicht, dass er einen Honigtopf vor sich hat, und versucht das Zielsystem zu kompromittieren. Aus seinen Aktionen lernen die Admins für die Abwehr künftiger Attacken [10].

Honeyd simuliert 65000 Systeme

Honeyd [11] kann verschiedene Systeme simulieren, indem er die passenden Serverantworten gibt und die TCP-Fingerprints imitiert. Honeyd muss in einem Netz mit öffentlichen IP-Adressen laufen, nur so kann der Daemon mehrere Systeme realistisch simulieren. Er erhält meist alle im Netzbereich freien IP-Adressen. Das System soll in der Lage sein, mehr als 65000 verschiedene Computer auf einer Hardware zu simulieren. Legitime Clients haben mit den unbenutzten IP-Adressen nichts zu schaffen, daher ist jeder Zugriff auf den Honeyd verdächtig. Er könnte der Beginn eines bislang unbekannten Angriffs sein.

Einer Gefahr setzt sich der Admin aber aus: Der Honeypot ist eventuell nicht völlig sicher, der Rechner könnte kompromittiert werden. Er darf daher nicht auf einem Produktivsystem laufen, sondern besser auf einem dedizierten Rechner. Forschungs-Honeypots stehen meist außerhalb der Firewall-Grenzen.

Neben den Forschungssystemen etablieren sich auch Produktiv-Teerfallen. Sie verzögern bekannte Angriffe und schützen damit Dritte. Zum Beispiel bremst Labrea Tarpit [12] den Code-Red-Wurm. Labrea öffnet auf freien IP-Adressen den Port 80. Es setzt die maximale Größe der Datenpakete auf ein Minimum und antwortet nur verzögert, wenn überhaupt. So blockiert jede Teerfalleninstanz beim Gegenüber einen Port und bremst die Ausbreitungsgeschwindigkeit des Wurms.

Produktiv-Honeypots als Schutzsystem

Besonders trickreich geht Labrea vor, um freie IP-Adressen zu ermitteln: Es protokolliert alle ARP-Requests im Netzsegment (siehe Titelthema dieses Hefts). Bleibt ein ARP-Request unbeantwortet, nimmt Labrea an, dass keine Maschine die IP verwendet, und meldet sich selbst als Besitzer der Adresse.

Auch Honeyd lässt sich zur Wurmkur nutzen. Laurent Oudot beschreibt [13], wie der Daemon den Msblast-Schädling bekämpft. Dass man Spammer auch beim Ausliefern ihres Mülls behindern kann, zeigt Daniel Rehbein [9]: Er nutzt einen präparierten Mailserver.

meist nicht wissen, ob der Semaphor bereits vorhanden ist. Die Funktion »sem_acquire()« führt in Zeile 35 die P-Operation aus. Sie wartet, bis der Prozess den kritischen Abschnitt betreten darf.
In Zeile 45 ist mit »sem_release()« das Gegenstück zu sehen, die V-Operation. Mittels »sem_remove()« könnte das Skript den Semaphor auch löschen. Dabei müsste es aber beachten, dass eventuell parallel laufende Prozesse diesen Semaphor bereits nutzen und später freigeben wollen. Sie erzeugen dann bei ihrer V-Operation eine Fehlermeldung, die das Skript in Zeile 47 vorsorglich abfängt und an den Admin meldet. Das System löscht automatisch Semaphore, die kein Prozess mehr verwendet, »sem_remove()« ist daher nicht nötig.

Praxis des Fallenstellens

Listing 1 zeigt ein PHP-Skript, das eine tückische HTML-Datei ausgibt (Abbildung 2). Sie enthält mehrere Links mit unterschiedlichen Beschreibungen und unterschiedlichen Zieladressen. Beide erzeugt das Skript mit Hilfe der PHP-Funktionen »uniqid()« und »md5()« (Zeilen 38 und 39). Die Wahrscheinlichkeit, dass zwei unterschiedliche IDs zu dem gleichen MD5-Hash führen, ist sehr gering. Die Anwendung lässt sich auch durch einen doppelten Dateinamen nicht stören – es gibt genügend weitere Links, in denen sich der Adressensammler verfängt.
Der Anfangswert des Semaphors liegt in der Konstanten »MAX_CONCURRENT_USERS« (Zeile 4). Sie definiert die Obergrenze der parallelen Zugriffe auf das Skript. Ein »sleep()«-Aufruf in Zeile 41 sorgt für die Wartezeit; wie lange das dauert, definiert die »TIME_WAIT«- Konstante sekundengenau (Zeile 3). Die Verzögerung ließe sich mit »usleep()« sogar in Mikrosekunden einstellen. Die gesamte Laufzeit des Skripts ist das Produkt aus der Anzahl der generierten Links und der Wartezeit, also »LINKS_GENERATED * TIME_WAIT«.
Die generierten Dateinamen sind nicht sonderlich kreativ. Als Verbesserung wäre es sinnvoll, beliebige Wörter aus dem Alphabet (A-Z, a-z, 0-9) zusammenzusetzen und für den Linktext ebenfalls Zeichenkombinationen zu verwenden, die der natürlichen Sprache ähnlich sind. Sie sollten auch Leerzeichen enthalten und in der Länge variieren. Zudem könnte die Dateinamen-Erweiterung aus einem Vorrat sinnvoller Extensionen zufällig gewählt werden. Mit allen Techniken zusammen dürfte es den Harvestern schwer fallen, solche Skripte zuverlässig zu erkennen.

Ein Skript mit vielen Namen

Die nächste Aufgabe lautet: das Skript bei jeder generierten Ziel-URL aufrufen, ohne dass der Besucher der geskripteten Seiten den Trick bemerkt. Hierfür bieten sich zwei Varianten an. Einmal kann der Webmaster in der Apache-Konfiguration einen Alias setzen. Listing 2 zeigt die Konfiguration unter der Annahme, dass das Skript im Verzeichnis »/spamfight« steht und »index.php« heißt. Die »AliasMatch«-Direktive unterstützt im Unterschied zum einfacheren »Alias« auch reguläre Ausdrücke.
Wer keinen Zugriff auf die »httpd.conf« hat, dem bleibt dieser elegante Weg verschlossen. Per ».htaccess« klappt es aber auch, der User muss nur den »errorDocument«-Handler zweckentfremden. Be-

Listing 1: Harvester-Bremse

```
01 <?
02   define("LINKS_GENERATED",20); // Links pro Seite
03   define("TIME_WAIT",1);        // Wartezeit [s] zwischen zwei Links
04   define("MAX_CONCURRENT_USERS",20); // Obergrenze paralleler Aufrufe
05   define("SEM_PERMS",0600);     // Rechte des Semaphors
06
07   /* HTTP-Header ausgeben.
08      Die Anfrage liefert immer OK, falls Einsatz über Error-Handler.
09      Die Seite ist immer neu, um eventuelle Proxys zu umgehen.
10      Content-Encoding wird wegen W3C-Konformität gesandt.
11    */
12   header("HTTP/1.1 200 OK");    // Für ErrorDocument-Variante
13   header("Status: 200 OK");     // Für PHP3-Kompatibilität
14   header("Expires: ".gmdate("D, d M Y H:i:s")." GMT");
15   header("Last-Modified: ".gmdate("D, d M Y H:i:s")." GMT");
16   header("Cache-Control: no-store, no-cache, must-revalidate");
17   header("Cache-Control: post-check=0, pre-check=0", false);
18   header("Pragma: no-cache");
19   header("Content-Encoding: iso-8859-1");
20
21   /* HTML-Kopf ausgeben */
22 ?><!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
23 <html>
24 <head>
25   <title>My very special links</title>
26   <META name="ROBOTS" content="NOINDEX, NOFOLLOW">
27 </head>
28 <body>
29 <h1>Welcome to my very special links.</h1>
30 <?
31   $sem_key = ftok (__FILE__,'T'); // Key für den Semaphor
32   if ($sem_key != -1) {            // Key korrekt erzeugt
33     $sem_id = sem_get($sem_key,MAX_CONCURRENT_USERS,SEM_PERMS);
34     if ($sem_id !== FALSE) {      // Semaphor erzeugt oder gefunden
35       if (sem_acquire($sem_id)) {
36         /* Kritischer Abschnitt. P(s) war erfolgreich. */
37         for ($i=0; $i<LINKS_GENERATED; $i++) {
38           print "<a href=\"". md5(uniqid(rand(), true)).
39                 ".php\">".    md5(uniqid(rand(), true)).
40                 "</a><br><br>";
41           sleep(TIME_WAIT);
42         }
43
44         /* Kritischen Abschnitt verlassen: V(s) */
45         if (!sem_release($sem_id)) {
46           /* Fehler bei V(s), Webmaster informieren */
47           die ('Internal Error');
48         }
49       } // end if sem_acquire
50     } // end if sem_id !== FALSE
51   } // end if semkey != -1
52
53   /* HTML-Footer */
54 ?>
55 </body>
56 </html>
```

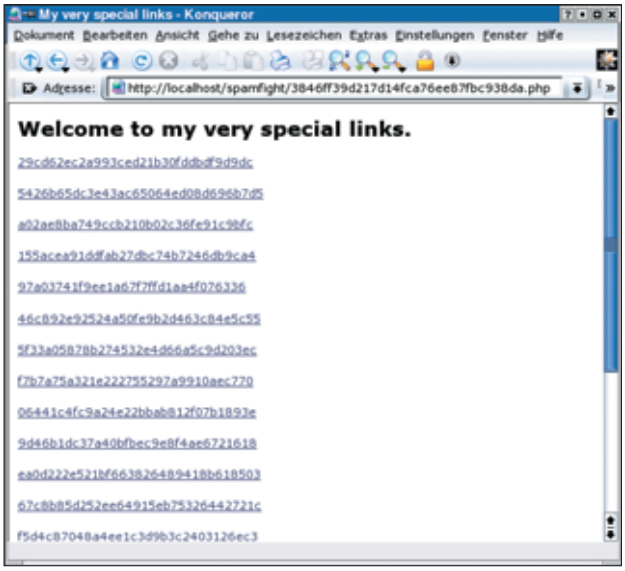

Abbildung 2: Die Harvester-Falle erzeugt unter beliebigem Namen (siehe Adresszeile) 20 zufällige Links, die wieder zum Skript führen. Beim Ausliefern jeder Seite lässt sich das PHP-Skript 20 Sekunden Zeit – eine Sekunde pro Zeile.

findet sich das Skript in »/spamfight«, ist folgender Eintrag zu wählen:

```
ErrorDocument 404 /spamfight/index.php
```

Das PHP-Skript muss vor dem Robot verbergen, dass in Wirklichkeit der HTTP-Fehler 404 aufgetreten ist. Daher sendet der Code in Listing 1 (Zeile 12) einen »HTTP/1.1 200 OK«-Header. Den würde der Webserver einfügen, wäre kein Fehler aufgetreten. Der zweite Header – »Status: 200 OK« – kann entfallen, wenn das Skript unter PHP 4 läuft. Wer auf »php.ini« Zugriff hat, schaltet zusätzlich das Flag »expose_php« auf »off«, um das Skript nicht durch einen PHP-Header zu verraten. Auch ist zu empfehlen, in der Zeile 39 von Listing 1 statt ».php« die Extension ».html« einzutragen.

Die Harvester-Falle soll zielsicher wirken und daher erwünschte Robots warnen, aber nicht fangen. Die beste Technik dafür ist eine »robots.txt«-Datei ([5], [6]), an die sich anständige Programme halten. Sie muss im Hauptverzeichnis des Webservers stehen. Wenn das Skript in »/spamfight« steht, lautet die »robots.txt«:

```
User-agent: *
Disallow: /spamfight/
```

Als zusätzliche Sicherheit gibt das PHP-Skript in Zeile 26 ein entsprechendes HTML-Tag aus.

Damit sind alle zu Beginn aufgestellten Anforderungen erfüllt, das Skript kann in Aktion treten. Bei 20 generierten Links ist die HTML-Datei 2054 Byte groß. Da das Skript maximal 20fach parallel läuft (Zeile 4) und pro Seite 20 Sekunden benötigt, können die Harvester im Schnitt maximal eine Seite pro Sekunde laden.

Berechenbare Last

Greifen die Harvester gierig zu, würde im Schnitt pro Sekunde ein Prozess terminieren. Damit ergäbe sich ein Datentransfervolumen von knapp 170 MByte pro Tag: 2054 Byte/Sekunde mal 60 Sekunden/Minute mal 60 Minuten/Stunde mal 24 Stunden/Tag durch 1024 und noch einmal durch 1024. Im Monat summiert sich das auf rund 5 GByte Traffic. Diese Grenze lässt sich durch eine andere Konfiguration des Skripts beliebig verschieben – längere Pausen und weniger Prozesse führen zu geringerem Datenverkehr.

Das Skript nutzt gezielt aus, dass zahlreiche Harvester die »robots.txt« ignorie-

Listing 2: Alias in »httpd.conf«

```
01 <Directory /spamfight>
02   AliasMatch ^/spamfight/[A-Za-z0-9]+\.[A-Za-z0-9]+$
   /spamfight/index.php
03 </Directory>
```

Honeypots gegen Spam

Um der zunehmenden Spamflut Einhalt zu gebieten, schlagen zahlreiche Autoren den Einsatz von Teerfallen und Honigtöpfen vor. Meist bremsen sie den SMTP-Dialog [9], [14]. Sie blockieren im Idealfall alle Ports des Spammers. Das leisten sie aber nur bedingt, da eine einzelne SMTP-Verbindung zahlreiche Mails übertragen kann. Dem Spammer genügt pro Server eine Verbindung, jede Honeypot-Instanz blockiert also nur einen Port.

Niels Provos [15] schlägt vor, komplette Class-C-Netze mit Honeyd auszustatten. Dort simuliert das System offene Relays und Proxies, um Spam zu sammeln und in kollaborative Filter einzutragen. Zugleich lassen sich so Erkenntnisse über die Methoden der Spammer gewinnen. In diese Richtung geht auch Laurent Oudot [16], der eine aufwändige Konstruktion aus offenen Proxies wählt. Diese simulieren die Weiterleitung der Spam-Mail nur, lassen aber Testmails der Spammer durch. Die Filterfunktion bleibt daher unbemerkt.

Der vorliegende Artikel geht einen anderen Weg. Lange vor dem Versand hindert eine Teergrube die Spammer daran, E-Mail-Adressen von Opfern zu sammeln. Das PHP-Skript blockiert nur den Robot und füttert ihn nicht – wie Daniel Rehbein [9] vorschlägt – mit präparierten E-Mail-Adressen.

Kritisches zu Teergruben und Honeypots

Bei aller Verlockung haften den Honeypots auch Nachteile an. Sie beanspruchen Ressourcen auf der Seite jener Anwender, die sich schützen wollen. Bei einigen Techniken ist es außerdem auch fraglich, wie sie juristisch zu bewerten sind. Ein modifizierter SMTP-Daemon ist durchaus geeignet, den Server eines legitimen Absenders zu blockieren und dort Schaden anzurichten. Ist eine Teergrube aber ausschließlich für Spammer erreichbar und warnt sie reguläre User ausdrücklich, dürfte es keine Probleme geben. Ein Spammer wird kaum den Klageweg beschreiten, dazu müsste er sich identifizieren. Das scheuen diese dunklen Gestalten in aller Regel.

Allerdings könnten sie auf ihre Art zurückschlagen und den Honeypot angreifen. Dieses Risiko einzuschätzen ist in der Praxis schwierig. Der Betrieb einer Teerfalle ist auch eine Kostenfrage. Je nach Konzept sind größere IP-Netzbereiche notwendig, die nur schwer zu bekommen sind. Zahlreiche Techniken fordern zudem eine große Verbreitung, um die Spammer effektiv zu bremsen. Ob diese kritische Masse jemals erreicht wird, ist fraglich.

Andererseits sind Teerfallen eines der wenigen Erfolg versprechenden Mittel im Kampf gegen Spam. Der Rechtsweg bleibt trotz des Begehungsort-Prinzips und der daraus folgenden Zuständigkeit der deutschen Gerichtsbarkeit gegen ausländische Spammer meist verschlossen – der Kläger müsste sicher wissen, woher der Spam stammt. Spamfilter sind unzuverlässig und bleiben stets einen Schritt hinter den neuesten Spammer-Techniken zurück.

ren, sich also entgegen der Konvention verhalten. Redliche Spider werden nicht geschädigt. Auch menschliche Besucher können kaum darüber stolpern, wenn der Einstiegslink auf das Skript gut versteckt ist. Ein 1 mal 1 Pixel großes Bild in Hintergrundfarbe mit entsprechendem »alt«-Text bleibt den meisten sicherlich verborgen.

Keine Sabotage

Computersabotage im Sinne der deutschen Gesetze dürfte kaum vorliegen, denn »robots.txt« weist normgerecht darauf hin, dass der Inhalt nicht für Spider geeignet ist. Wer das ignoriert, handelt auf eigene Gefahr. *(fjl)* ■

Infos

[1] Center for Democracy and Technology, „Why am I getting all this spam?“: [http://www.cdt.org/speech/spam/030319spamreport.pdf]
[2] Tobias Eggendorfer, „Privatadresse – Homepages spamsicher gestalten“: Linux-User 05/04, S. 42
[3] Marc André Selig, „Dienstgespräche – Interprozesskommunikation jenseits der Signale“: Linux-Magazin 05/04, S. 76
[4] Andrew S. Tanenbaum, „Moderne Betriebssysteme“: Pearson Studium 2002
[5] W3C Recommendations, Appendix B, „Performance, Implementation and Design Notes“: [http://w3.org/TR/REC-html40/appendix/notes.html]
[6] Martijn Koster, „A Standard for Robot Exclusion“: [http://www.robotstxt.org/wc/norobots.html]
[7] Richard Stevens, „Kosten-, Nutzen- und Risikoanalyse für den Einsatz von Honeynets“: Diplomarbeit an der FH Bonn-Rhein-Sieg, 2004
[8] Lutz Donnerhacke, „Teergruben-FAQ“: [http://www.iks-jena.de/mitarb/lutz/usenet/teergrube.html]
[9] Daniel Rehbein, „Gift für Harvester – aus meiner Entwicklung“: [http://www.daniel-rehbein.de/spamgift.html]
[10] Lance Spitzner, „Know your Enemy: Honeynets“: [http://project.honeynet.de/papers/honeynet/]
[11] Honeyd: [http://www.honeyd.org/]
[12] Labrea Tarpit: [http://labrea.sf.net]
[13] Laurent Oudot, „Honeyd vs. msblast.exe. Fighting worms with honeypots“: [http://citi.umich.edu/u/provos/honeyd/msblast.html]
[14] Lutz Donnerhacke, „Teergrubing Wrapper“: [http://www.iks-jena.de/mitarb/lutz/usenet/antispam.html]
[15] Niels Provos, „Honeyd Research: Honeypot against Spam“: [http://www.honeyd.org/spam.php]
[16] Laurent Oudot, „Fighting Spammers With Honeypots“: [http://www.securityfocus.com/infocus/1747]

Der Autor

Diplom-Wirtschaftsingenieur Tobias Eggendorfer ist in München als freiberuflicher IT-Berater und Dozent tätig. Die Bekämpfung von Spam – nicht nur mit technischen Mitteln – zählt zu seinen Hauptthemen.

User Friendly – der monatliche Comic im Linux-Magazin