Firewalking

A Traceroute-Like Analysis of IP Packet Responses to Determine Gateway Access Control Lists

Cambridge Technology Partners'
Enterprise Security Services

David Goldsmith
Senior Security Architect
dhg@es2.net

Michael Schiffman
Senior Security Architect
mds@es2.net

October 1998

TABLE OF CONTENTS

i. Terminology

ACL	Access Control List. A set of rules that enforce a security policy. In the scope of this paper, an Access Control List will solely apply to network policy.
Router/Gateway	Used interchangeably. In the scope of this report, they refer to a multi-homed host that is configured to forward IP datagrams. It may or may not have a packet filtering ACL in place that denies some network traffic.
Ingress traffic	Describes network traffic that originates from the outside of a network perimeter and progresses towards the inside.
Egress traffic	Describes network traffic that originates from the inside of a network perimeter and progresses towards the outside.
Firewall	Refers to a multi-homed host configured to forward IP datagrams which uses a packet filtering ACL to control network traffic.

ii. A note about examples

There are several sample traceroute dumps used in this report. The astute reader will note that the IP addresses are RFC 1918[1] compliant non-routable internal network addresses. The empirical data and traceroute dumps are taken directly from live Internet hosts[1], and in order to protect their identity, we have changed the addresses to anonymize the machines and networks involved.

[1] In fact, in the traceroute dumps, the original RTTs (round-trip times) are left in as they appeared.

I. Introduction

This paper describes Firewalking, a technique that can be used to gather information about a remote network protected by a firewall. The purpose of the paper is to examine the risks that this technique represents. This paper is intended for a technical audience with an advanced understanding of network infrastructure and TCP/IP packet structures.

Firewalking uses a traceroute-like IP packet analysis to determine whether or not a particular packet can pass from the attacker's host to a destination host through a packet-filtering device. This technique can be used to map 'open' or 'pass through' ports on a gateway. More over, it can determine whether packets with various control information can pass through a given gateway. Also, using this technique, an attacker can map routers behind a packet-filtering device. To fully understand how this technique works, we first need to understand how traceroute works. This paper provides an introduction to traceroute.

II. Traceroute

Traceroute [1] is a network debugging utility designed to map out all hosts en route to a particular destination. Traceroute works by sending UDP or ICMP echo (ping)[2] packets to a destination host and monotonically increasing the time to live (TTL) field in the IP header each successive round (by default, a round consists of three packets or probes). If the traceroute scan is done using UDP the destination port will be incremented with each probe sent.

The IP TTL field is used to limit the lifetime of datagrams across the Internet and is decremented just before a router forwards a packet. If this reduction would cause the TTL to be 0 or less, the router in question will send back an ICMP error message (time to live exceeded in transit) to the original host. This lets the original host know at which router the packet expired. By starting the TTL at one, routers between two given hosts can be found by increasing the TTL and monitoring the ICMP responses (provided there isn't any prohibitive filtering or any severe packet loss). To ensure that it gets a proper response from the ultimate destination host (an ICMP port unreachable or an ICMP echo reply) traceroute will either pick a high UDP port that is unlikely to be used by any application or use ping packets.

[2] Traceroute version 1.4a5 (available at ftp://ee.lbl.gov/traceroute1.4a5.tar.Z) allows for ICMP echo based traceroutes via the –I flag. Windows NT's version of traceroute '*tracert*' exclusively uses ICMP echoes.

III. Information gathering using traceroute

With an understanding of how traceroute works, we can now explore how this can this be used to leverage information about a particular network. This section will demonstrate two different ways of using traceroute to do some network reconnaissance. These following examples are contrived to show specific situations that may or may not be commonplace.

• Protocol subterfuge

The first scenario involves a network protected by a firewall that is blocking all ingress traffic except for ping and ping responses (ICMP types 8 and 0 respectively). We can use the stock traceroute program to show us what hosts are behind this filter (which is presumably against the security policy). Instead of the default behavior of using UDP (*Figure 1*), we want to force traceroute to use ICMP packets (*Figure 2*). Notice that this time we are able to view hosts behind the firewall.

```
zuul:~>traceroute 10.0.0.10
traceroute to 10.0.0.10 (10.0.0.10), 30 hops max, 40 byte
packets
 1  10.0.0.1 (10.0.0.1)  0.540 ms  0.394 ms  0.397 ms
 2  10.0.0.2 (10.0.0.2)  2.455 ms  2.479 ms  2.512 ms
 3  10.0.0.3 (10.0.0.3)  4.812 ms  4.780 ms  4.747 ms
 4  10.0.0.4 (10.0.0.4)  5.010 ms  4.903 ms  4.980 ms
 5  10.0.0.5 (10.0.0.5)  5.520 ms  5.809 ms  6.061 ms
 6  10.0.0.6 (10.0.0.6)  9.584 ms  21.754 ms  20.530 ms
 7  10.0.0.7 (10.0.0.7)  89.889 ms  79.719 ms  85.918 ms
 8  10.0.0.8 (10.0.0.8)  92.605 ms  80.361 ms  94.336 ms
 9  * * *
10  * * *
```

Figure 1

```
zuul:~>traceroute –I 10.0.0.10
traceroute to 10.0.0.10 (10.0.0.10), 30 hops max, 40 byte
packets
 1  10.0.0.1 (10.0.0.1)  0.540 ms  0.394 ms  0.397 ms
 2  10.0.0.2 (10.0.0.2)  2.455 ms  2.479 ms  2.512 ms
 3  10.0.0.3 (10.0.0.3)  4.812 ms  4.780 ms  4.747 ms
 4  10.0.0.4 (10.0.0.4)  5.010 ms  4.903 ms  4.980 ms
 5  10.0.0.5 (10.0.0.5)  5.520 ms  5.809 ms  6.061 ms
 6  10.0.0.6 (10.0.0.6)  9.584 ms  21.754 ms  20.530 ms
 7  10.0.0.7 (10.0.0.7)  89.889 ms  79.719 ms  85.918 ms
 8  10.0.0.8 (10.0.0.8)  92.605 ms  80.361 ms  94.336 ms
 9  10.0.0.9 (10.0.0.9)  94.127 ms  81.764 ms  96.476 ms
 10 10.0.0.10 (10.0.0.10) 96.012 ms  98.224 ms  99.312 ms
```

Figure 2

• Nascent port seeding

The second scenario involves a more common example of a network protected by a firewall which blocks all ingress traffic except for UDP port 53 (Domain Name Service or DNS).

```
zuul:~>traceroute 10.0.0.10
traceroute to 10.0.0.10 (10.0.0.10), 30 hops max, 40 byte
packets
 1  10.0.0.1 (10.0.0.1)  0.540 ms  0.394 ms  0.397 ms
 2  10.0.0.2 (10.0.0.2)  2.455 ms  2.479 ms  2.512 ms
 3  10.0.0.3 (10.0.0.3)  4.812 ms  4.780 ms  4.747 ms
 4  10.0.0.4 (10.0.0.4)  5.010 ms  4.903 ms  4.980 ms
 5  10.0.0.5 (10.0.0.5)  5.520 ms  5.809 ms  6.061 ms
 6  10.0.0.6 (10.0.0.6)  9.584 ms  21.754 ms  20.530 ms
 7  10.0.0.7 (10.0.0.7)  89.889 ms  79.719 ms  85.918 ms
 8  10.0.0.8 (10.0.0.8)  92.605 ms  80.361 ms  94.336 ms
 9  * * *
10  * * *
```

Figure 3

As you can see from figure 3, the traceroute scan is blocked at the 8th hop because no traffic is allowed entrance into the network except for DNS queries. Armed with this knowledge, we can easily map hosts behind the gateway.

We can control the following:

- The starting source port of the traceroute (which, by default, increases monotonically as each probe is sent).
- The number of probes sent each round (by default this is 3).

We can determine the following:

- The number of hops in between our attacking host and the target firewall.

This information allows us to deterministically control the port number of the probe that will reach the firewall. Due to the fact that the firewall does no content analysis, we can fool it into thinking our packets are DNS queries, and therefore, we can bypass the ACL. We simply begin our scan with a starting port number of:

`(target_port - (number_of_hops * num_of_probes)) - 1`

If you are more then (target_port – 1) number of hops from your destination this method obviously will not work. For our above example this gives us:

`(53 - (8 * 3)) - 1 = 28`

The probe that reaches the filter will have an acceptable port number as dictated by the firewall's ACL and will be allowed to pass unmolested (*Figure 4*).

```
zuul:~>traceroute -p28 10.0.0.10
traceroute to 10.0.0.10 (10.0.0.10), 30 hops max, 40 byte packets
 1  10.0.0.1 (10.0.0.1)  0.501 ms  0.399 ms  0.395 ms
 2  10.0.0.2 (10.0.0.2)  2.433 ms  2.940 ms  2.481 ms
 3  10.0.0.3 (10.0.0.3)  4.790 ms  4.830 ms  4.885 ms
 4  10.0.0.4 (10.0.0.4)  5.196 ms  5.127 ms  4.733 ms
 5  10.0.0.5 (10.0.0.5)  5.650 ms  5.551 ms  6.165 ms
 6  10.0.0.6 (10.0.0.6)  7.820 ms  20.554 ms  19.525 ms
 7  10.0.0.7 (10.0.0.7)  88.552 ms  90.006 ms  93.447 ms
 8  10.0.0.8 (10.0.0.8)  92.009 ms  94.855 ms  88.122 ms
 9  10.0.0.9 (10.0.0.9)  101.163 ms * *
10  * * *
```

Figure 4

You will notice that the scan terminates immediately after the target port is passed. This is due to the fact that traceroute continues to increase the port numbers for each probe sent. The probe immediately after the successful one will be denied by the ACL on the firewall. To possibly get further, a simple modification to traceroute can be done to add a command line switch to stop port incrementation (*Figure 5*). This allows us to force every probe we send to be acceptable to the firewall's ACL (a side effect being that we might not get the normal ICMP unreachable message from the ultimate destination due to the fact that there might actually be something listening on the other end). See appendix A for the source code patch.

```
zuul:~>traceroute -S –p53 10.0.0.15
traceroute to 10.0.0.15 (10.0.0.15), 30 hops max, 40 byte
packets
 1  10.0.0.1 (10.0.0.1)  0.516 ms  0.396 ms  0.390 ms
 2  10.0.0.2 (10.0.0.2)  2.516 ms  2.476 ms  2.431 ms
 3  10.0.0.3 (10.0.0.3)  5.060 ms  4.848 ms  4.721 ms
 4  10.0.0.4 (10.0.0.4)  5.019 ms  4.694 ms 4.973 ms
 5  10.0.0.5 (10.0.0.5)  6.097 ms  5.856 ms  6.002 ms
 6  10.0.0.6 (10.0.0.6)  19.257 ms  9.002 ms  21.797 ms
 7  10.0.0.7 (10.0.0.7)  84.753 ms * *
 8  10.0.0.8 (10.0.0.8)  96.864 ms  98.006 ms  95.491 ms
 9  10.0.0.9 (10.0.0.9)  94.300 ms *  96.549 ms
10  10.0.0.10 (10.0.0.10)  101.257 ms  107.164 ms  103.318 ms
11  10.0.0.11 (10.0.0.11)  102.847 ms  110.158 ms *
12  10.0.0.12 (10.0.0.12)  192.196 ms  185.265 ms *
13  10.0.0.13 (10.0.0.13)  168.151 ms  183.238 ms  183.458 ms
14  10.0.0.14 (10.0.0.14)  218.972 ms  209.388 ms  195.686 ms
15  10.0.0.15 (10.0.0.15)  236.102 ms  237.208 ms  230.185 ms
```

Figure 5

• Taking it a bit further

Since the magic of traceroute is all happening at the IP layer, any transport protocol (UDP, TCP and ICMP) can be used. The foundation laid down by traceroute can extend to any other protocol on top on IP. If we attempt to traceroute to a machine behind a firewall and the probe reaching the firewall is prohibited by an ACL filter, the packet will be dropped on the floor (in most cases). All we can determine from the traceroute scan is the last gateway (in this case, a firewall) that responded. This is good entropic information. This firewall can then become a waypoint that we use to determine the success of future probes. If we traceroute to a machine behind this firewall with a different (protocol) traceroute probe, and we get a response, we know two things: 1) that particular kind of traffic is passed by the firewall, and 2) we know a host behind the firewall. If we only get as far as our waypoint, we know that traffic type is filtered. This is the basis for firewalking.

IV. Firewalking

In order to use a gateway's response to gather information, we must know two pieces of information:

- The IP address of the last known gateway before the firewalling takes place
- The IP address of a host located behind the firewall.

The first IP address serves as our metric (waypoint from the above example), if we can't get a response past that machine, then we assume that whatever protocol we tried to pass is being blocked[3]. The second IP address is used as a destination to direct the packet flow (*Figure 6*).

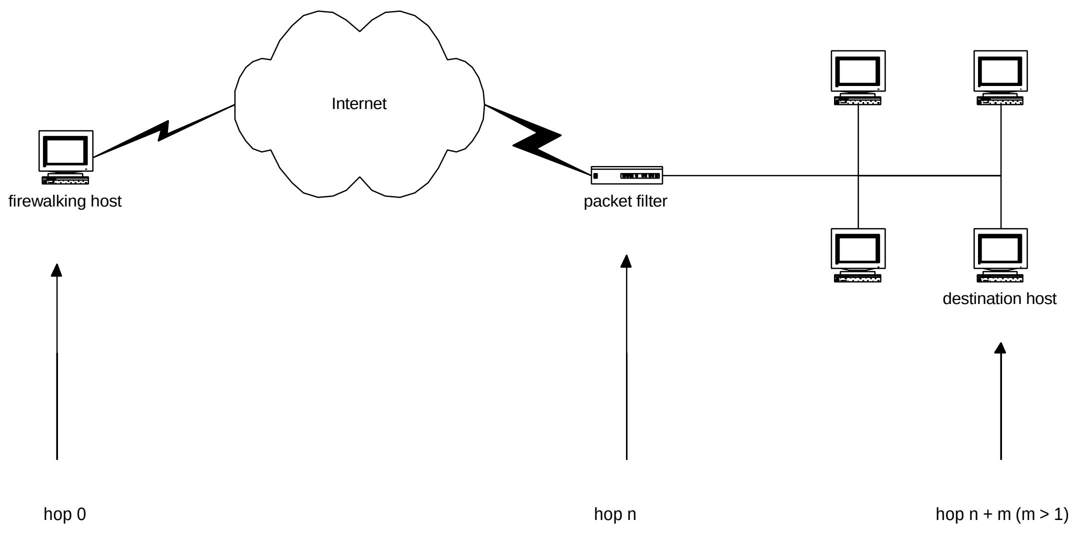

figure 6

Using this technique, we can perform several different information gathering attacks. One attack is a firewall protocol scan, which will determine what ports/protocols a firewall will let traffic through on from the attacking host. This would attempt to pass packets on all ports and protocols and monitor the responses. A second potential threat is advanced network mapping. By sending packets to every host behind a packet filter, an attacker can generate an accurate map of a network's topology.

[3]It should be noted that the assumption that it is our target gateway that is dropping the traffic may not be correct. There are several things that could cause a false positive in this case:
- A host could also be down or simply not responding.
- IP is unreliable. Packets can be dropped for any number of reasons.
- The packet could also be dropped by a previous filtering gateway before it ever reaches our target gateway host.

V. Firewalk – The tool

While traceroute is a useful application, it is not very extensible for any kind of serious reconnaissance scanning; to this end, the proof of concept tool, firewalk, was built.

• Fire, walk with me where?

Firewalk is a network-auditing tool that employs the techniques described above. It attempts determines what transport protocols a given gateway will let through. The firewalk scan works by sending out TCP or UDP packets with an IP TTL one greater then the targeted gateway. If the gateway allows the traffic, it will forward the packets to the next hop where they will expire and elicit a TTL exceeded in transit message. If the gateway host does not allow the traffic, it will likely drop the packets on the floor and we will see no response. By sending probes in a successive manner and recording which ones answer and which ones don't, the access list on the gateway can be determined.

• 2 Phases

To work its magic, firewalk has two phases, a network discovery phase, and a scanning phase. Initially, to get the correct IP TTL (that will result in expired packets one beyond the gateway) we need to 'ramp up' hop counts. We do TTL ramping in the same manner that traceroute works, sending packets out with successively incremented IP TTLs, towards the destination host[4]. Once we know the gateway hopcount (at that point the scan is 'bound') we can move onto the next phase, the actual scan.

The actual scan is simple. Firewalk sends out TCP or UDP packets and sets a timeout; if it receives a response before the timer expires, the port is considered open, if it doesn't, the port is considered closed (*Figure 7*).

```
zuul:#firewalk -n -P1-8 –pTCP 10.0.0.5 10.0.0.20
Firewalking through 10.0.0.5 (towards 10.0.0.20) with a maximum
of 25 hops.
Ramping up hopcounts to binding host...
probe:  1  TTL:  1  port 33434:  <response from> [10.0.0.1]
probe:  2  TTL:  2  port 33434:  <response from> [10.0.0.2]
probe:  3  TTL:  3  port 33434:  <response from> [10.0.0.3]
probe:  4  TTL:  4  port 33434:  <response from> [10.0.0.4]
probe:  5  TTL:  5  port 33434:  Bound scan: 5 hops <Gateway at
5 hops> [10.0.0.5]

port   1: open

port   2: open

port   3: open

port   4: open

port   5: open

port   6: open

port   7: *

port   8: open

13 packets sent, 12 replies received
```

Figure 7

• A Slow Walk

[4] It is significant to note that the ultimate destination host does not have to be reached. It just needs to be somewhere downstream, on the other side of the gateway from the firewalking host.

As noted above, packets on an IP network can be dropped for a variety of reasons. When a packet is dropped for any reason other then it being denied by a filter, it is *extraneous loss*. For our firewalk scan to be accurate, we need to limit this extraneous packet loss to the best of our ability. The best we can do in most cases is to be redundant with the number of probes we send. Unless there is severe network congestion some of the probes should get through. However, what if the probe we send is filtered or dropped by a *different* gateway while en route to the target gateway (*see figure 8*)?

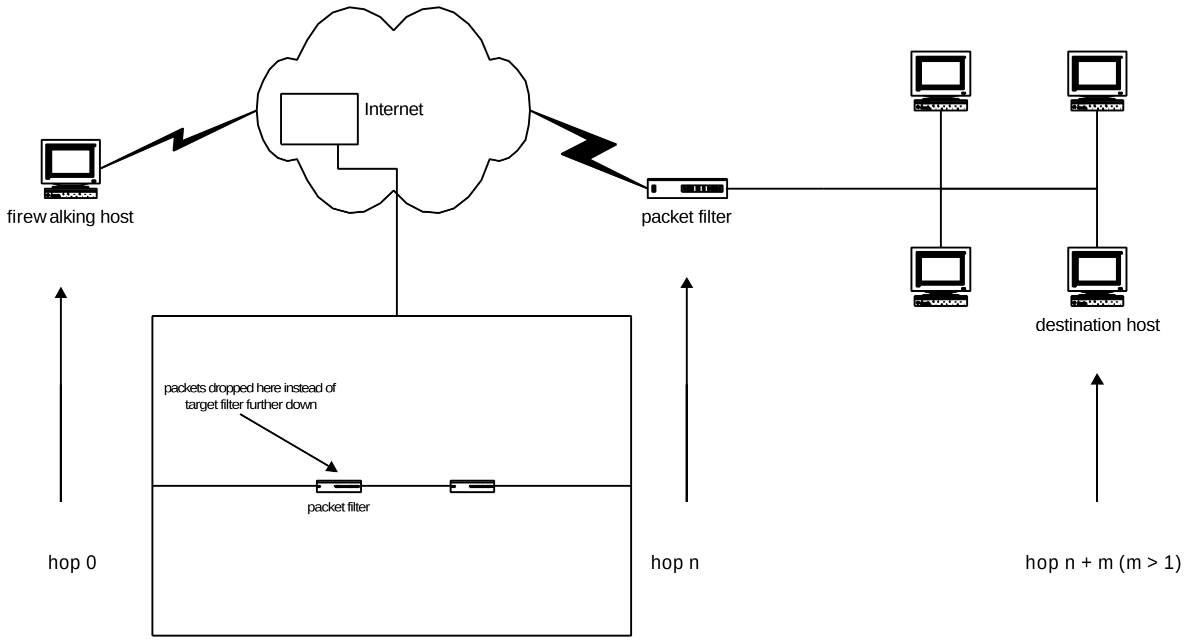

figure 8

To firewalk, this will look like the target gateway has denied the packet, which, in this case, is certainly a false negative. This is not extraneous loss, so simply sending more packets will not help. To prevent this, we must perform a `slow walk` or a `creeping walk`. This is akin to a normal scan, however we scan each hop en route to the target. We perform a standard firewalk ramping phase, and then scan each intermediate hop up to the destination. This allows prevents false negatives due to intermediate filter blockage and allows firewalk to be more confident in its report. The major benefit is that we can now determine if blocked ports are false negatives[5]. The drawback is that it is, as it's name states, slow.

More information about Firewalk (including the source) is available from http://www.es2.net/research/firewalk.

[5] If an intermediate filter is shown to drop packets, this prevents firewalk from scanning the actual target machine for the blocked packet type, on that route. This is annoying.

VI. Risk Mitigation

The easiest solution to this problem is to disallow ICMP TTL Exceeded messages from leaving an internal network. This will also have the affect of breaking valid uses of traceroute and may inhibit remote diagnostics of an internal network problem.

Another defense against Firewalking is the use of some form of proxy server. Network Address Translation (NAT) or any proxy server (both application level and circuit level) can prevent Firewalk from probing behind them. While network based intrusion detection tools could detect certain attacks [3]; it is possible to develop a version of Firewalk that would generate packets that would look like valid packets for each service that it is scanning. Currently, Firewalk only fills in the packet header and does not insert any data into a packet. A more sophisticated version could emulate various services in an attempt to masquerade as valid traffic and randomize the order and times that it scans services.

Appendix A. traceroute static port diff

Apply this diff to traceroute version 1.4a5 to add support for static destination ports. Apply the diff using the unix patch program from the traceroute source directory:

```
---------------------8<-------- traceroute.diff ------------------------------
--- traceroute.c.orig   Fri Aug 21 15:15:23 1998
+++ traceroute.c        Sun Aug 23 18:58:08 1998
@@ -289,6 +289,7 @@
 int nprobes = 3;
 int max_ttl = 30;
 int first_ttl = 1;
+int static_port = 0;
 u_short ident;
 u_short port = 32768 + 666;    /* start udp dest port # for probe packets */

@@ -352,7 +353,7 @@
                prog = argv[0];

        opterr = 0;
-       while ((op = getopt(argc, argv, "dFInrvxf:g:i:m:p:q:s:t:w:")) != EOF)
+       while ((op = getopt(argc, argv, "dFInrvxf:g:i:m:p:q:Ss:t:w:")) != EOF)
                switch (op) {

                case 'd':
@@ -406,6 +407,13 @@
                        options |= SO_DONTROUTE;
                        break;

+               case 'S':
+                       /*
+                        * Tell traceroute to not increment the destination
+                        * port, useful for bypassing some packet filters.
+                        * Useless without the -p option.
+                       static_port = 1;
+                       break;
                case 's':
                        /*
                         * set the ip source address of the outbound
@@ -744,7 +752,7 @@
                        register struct ip *ip;

                        (void)gettimeofday(&t1, &tz);
-                       send_probe(++seq, ttl, &t1);
+                       send_probe(static_port ? seq : ++seq, ttl, &t1);
                        while ((cc = wait_for_reply(s, from, &t1)) != 0) {
                                (void)gettimeofday(&t2, &tz);
                                i = packet_ok(packet, cc, from, seq);
@@ -1300,9 +1308,9 @@
        extern char version[];

        Fprintf(stderr, "Version %s\n", version);
-       Fprintf(stderr, "Usage: %s [-dFInrvx] [-g gateway] [-i iface] \
-[-f first_ttl] [-m max_ttl]\n\t[ -p port] [-q nqueries] [-s src_addr] [-t tos]
\
-[-w waittime]\n\thost [packetlen]\n",
+       Fprintf(stderr, "Usage: %s [-dFInrSvx] [-g gateway] [-i iface] \
+[-f first_ttl]\n\t[-m max_ttl] [ -p port] [-q nqueries] [-s src_addr] \
+[-t tos]\n\t[-w waittime] host [packetlen]\n",
            prog);
        exit(1);
 }

---------------------8<-------- traceroute.diff ------------------------------
```

Appendix B. References

[1] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot and E. Lear, “Address Allocation for Private Internets” RFC1918, February 1996

[2] Van Jacobson, traceroute documentation and source code, Lawrence Berkeley National Laboratory

[3] Thomas H. Ptacek and Timothy Newsham, “Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection”, Secure Networks, January 1998